Department of Computer Science

DRAFT PROCEEDINGS OF THE 24TH SYMPOSIUM ON IMPLEMENTATION AND APPLICATION OF FUNCTIONAL LANGUAGES (IFL 2012)

Ralf Hinze (Editor)

RR-12-06

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD

Foreword

It is my great pleasure to welcome you to IFL 2012, the 24th Symposium on Implementation and Application of Functional Languages — held in Oxford, UK, during 30 August – 1 September, 2012. The goal of the IFL symposia is to bring together researchers actively engaged in the implementation and application of functional and function-based programming languages. IFL is a venue for researchers to present and discuss new ideas and concepts, work in progress, and publication-ripe results related to the implementation and application of functional languages and function-based programming.

The call for papers generated 37 submissions, all of which were accepted for presentation and are contained in these draft proceedings. The submissions were screened by the programme committee chair to make sure they are within the scope of IFL. It should be stressed, however, that these contributions are not peer-reviewed publications. Following the IFL tradition, IFL 2012 will use a post-symposium review process to produce formal proceedings, which will be published by Springer Verlag in the Lecture Notes in Computer Science series. After the symposium, authors will be given the opportunity to incorporate the feedback from discussions at the symposium and will be invited to submit a revised paper for the formal review process. From the revised submissions, the programme committee will select papers for the formal proceedings considering their correctness, novelty, originality, relevance, significance, and clarity.

The programme consists of 37 presentations and one invited talk. Fritz Henglein, from the Department of Computer Science at the University of Copenhagen, is the invited speaker of IFL 2012. He will talk about generic sorting and partitioning in linear time and fully abstractly.

Putting together IFL 2012 was truly a team effort. I am grateful to the Department of Computer Science, Elizabeth Walsh in particular, for administrative support and to St Anne's college for hosting the event. I would like to thank the members of the programme committee for accepting my invitation and for their work in putting together the programme. Last but not least I would like to thank Kwok-Ho Cheung, Tom Harper, Daniel James, José Pedro Magalhães and Nicolas Wu, for their help with organizing the symposium and for distributing the call for papers.

Ralf Hinze
Chair of IFL 2012
University of Oxford
Oxford, UK, August 2012

Contents

Session 1 *Session Chair: José Pedro Magalhães*

1.1 Modular Monadic Reasoning, a (Co-)Routine
Steven Keuchel and Tom Schrijvers 4

1.2 A Notation for Comonads
Dominic Orchard and Alan Mycroft 19

1.3 On monadic parametricity of second-order functionals
Andrej Bauer, Martin Hofmann and Aleksandr Karbyshev 35

Session 2 *Session Chair: Clemens Grelck*

2.1 Iterating Skeletons - Structured Parallelism by Composition
Mischa Dieterle, Thomas Horstmeyer, Jost Berthold and Rita Loogen 51

2.2 Data Layout Inference for Code Vectorisation
Artjoms Šinkarovs and Sven-Bodo Scholz 71

2.3 The Design of a GUMSMP: a Multilevel Parallel Haskell Implementation
Malak Aljabri, Phil Trinder and Hans-Wolfgang Loidl 73

2.4 Specification of Extensible Sparse Functional Arrays
John T. O'Donnell 89

Session 3 *Session Chair: Peter Thiemann*

3.1 Data Change Notifications for Cooperative Web Applications
Bob van der Linden, Steffen Michels and Rinus Plasmeijer 98

3.2 Building JavaScript Applications with Haskell
Atze Dijkstra, Jurriën Stutterheim, Alessandro Vermeulen and Doaitse Swierstra 111

3.3 Push-Pull Signal-Function Functional Reactive Programming
Edward Amsden 126

3.4 Parameterized Parsers
Kathryn E Gray 141

Session 4 *Session Chair: Nicolas Wu*

4.1 Skew Generic Test Data Generation
Pieter Koopman and Rinus Plasmeijer 157

4.2 Advances in Lazy SmallCheck: Efficient testing of higher-order properties with mixed quantification
Jason S. Reich, Matthew Naylor and Colin Runciman 171

4.3 Functional Proxy Programming or Tinker Tailor Soldier Spy
David Wakeling 186

4.4 The Quintessential Neural Network Programming Language
Gene Sher 201

Session 5: Invited Talk *Session Chair: Ralf Hinze*

5.1 Have Your Cake And Eat It, Too: Generic sorting and partitioning in linear time and fully abstractly-simultaneously
Fritz Henglein 217

Session 6 *Session Chair: Tom Schrijvers*
6.1 Fusion in the Utrecht Haskell Compiler: Extended Abstract
Thomas Harper . . . 218
6.2 OCaml-Java: from OCaml sources to Java bytecodes
Xavier Clerc . . . 221
6.3 The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language
Neil Sculthorpe, Andrew Farmer and Andy Gill . . . 237
6.4 Optimisation of Generic Programs through Inlining
José Pedro Magalhães . . . 253

Session 7 *Session Chair: Wouter Swierstra*
7.1 The Nax Programming language (work in progress)
Ki Yung Ahn, Tim Sheard, Marcelo Fiore and Andrew M. Pitts . . . 269
7.2 Security Type Error Diagnosis
Jeroen Weijers, Jurriaan Hage and Stefan Holdermans . . . 307
7.3 A Type- and Control-Flow Analysis for System F
Matthew Fluet . . . 326
7.4 Verified and Executable Semantics in Coq
Ken Madlener and Sjaak Smetsers . . . 350

Session 8 *Session Chair: Jeremy Gibbons*
8.1 Dependently-typed Programming in Scientific Computing
Cezar Ionescu and P. Jansson . . . 368
8.2 Applications of Reflection in Agda
Paul van der Walt and Wouter Swierstra . . . 371
8.3 Agda Meets Accelerate
Peter Thiemann and Manuel Chakravarty . . . 382
8.4 On Binomial Expansions in Moessner's Theorem
Olivier Danvy and Moe Masuko . . . 398

Session 9 *Session Chair: Stephan Herhut*
9.1 Functional implementation of well-typings in Java
Martin Pluemicke . . . 401
9.2 User-Defined Shape Constraints in SAC
Fangyong Tang and Clemens Grelck . . . 414
9.3 Tyre-Check-I: Low-level Type Inference for Reduceron Code Safety
Marco Polo Perez and Colin Runciman . . . 433
9.4 An Embedded Type Debugger
Kanae Tsushima and Kenichi Asai . . . 447

Session 10 *Session Chair: Pablo Nogueira*
10.1 The Design of Scalable Distributed Erlang
Natalia Chechina, Phil Trinder, Amir Ghaffari, Rickard Green, Kenneth Lundin and Robert Virding . . . 459

10.2 A Concurrent Persistent Functional Language, Towards Practical Functional Databases
Lesley Wevers, Marieke Huisman and Ander de Keijzer 475
10.3 Detecting Process Relationships in Erlang Programs
Melinda Tóth and István Bozó 494
10.4 Skel: A Streaming Process-based Skeleton Library for Erlang
Archibald Elliott, Christopher Brown, Marco Danelutto and Kevin Hammond 509

Session 11 *Session Chair: Rinus Plasmeijer*
11.1 Pure and Lazy Lambda Mining
Nicolas Wu, José Pedro Magalhães, Jeroen Bransen and Wouter Swierstra 519
11.2 Decomposing Metaheuristic Operations
Richard Senington and David Duke 536
11.3 Rational Term Equality, Functionally
Tom Schrijvers and Bruno Oliveira 548

Organisation

Chair

Ralf Hinze, University of Oxford, UK

Programme Committee

Edwin Brady, University of St. Andrews, UK

Andrew Butterfield, University of Dublin, Ireland

Matthew Flatt, University of Utah, US

Andy Gill, University of Kansas, US

Stephan Herhut, Intel Labs, Santa Clara, US

Zhenjiang Hu, National Institute of Informatics, Japan

Patrik Jansson, Chalmers University of Technology, Sweden

Mauro Jaskelioff, Universidad Nacional de Rosario, Argentina

Gabriele Keller, University of New South Wales, Australia

Simon Marlow, Microsoft Research, UK

Pablo Nogueira, Technical Univerity of Madrid, Spain

Bruno Oliveira, Seoul National University, Korea

José Nuno Oliveira, University of Minho, Portugal

Rinus Plasmeijer, Radboud University Nijmegen, Netherlands

Tom Schrijvers, Ghent University, Belgium

Tim Sheard, Portland State University, US

Wouter Swierstra, University of Utrecht, Netherlands

Peter Thiemann, University of Freiburg, Germany

Simon Thompson, University of Kent, UK

Steve Zdancewic, University of Pennsylvania, US

Organising Committee

Nicolas Wu, University of Oxford, UK

José Pedro Magalhães, University of Oxford, UK

Modular Monadic Reasoning, a (Co-)Routine

Steven Keuchel and Tom Schrijvers

Universiteit Gent, Belgium,
{steven.keuchel,tom.schrijvers}@ugent.be

Abstract. Higher-order functions that are polymorphic in a monad make highly flexible modular components. Unfortunately, the combination of an unknown function parameter and a polymorphic monad are detrimental to reasoning. This paper shows how to eliminate both the function parameter and the polymorphism. The resulting characterization is amenable to reasoning.
The approach is based on a judicious combination of the coroutine monad transformer and monad morphisms.

1 Introduction

Modularity is one of the holy grails of software engineering. The dream is to be able build new software systems entirely from reusable components, that have been written independently and can (potentially) be reused in many different configurations for different applications. Increasingly more modularity demands are being made: without modifying components, it must be possible to augment or modify their behavior. At the same time, unprincipled copy-&-paste approaches, even performed by automated tools, are not acceptable. Components must have a meaning beyond their textual form and independently of a particular composition; they must support *modular reasoning*.

This paper considers modular reasoning in the purely functional setting of polymorphic monadic mixin components. Mixin components [4] employ a form of dynamic inheritance, called open recursion, to make a function's behavior modifiable. Monads [17] enable components to have side effects. Polymorphic monads do not fix the side effects up front, but enable different components in a composition to contribute their own side effects to the whole.

These polymorphic monadic mixin components are extremely flexible in their use. Yet, they prove to be highly challenging with respect to modular reasoning. Oliveira et al. [11] show how to do reasoning about non-interference of such components based on parametricity, i.e., based on the types of the components alone. However, modular reasoning about more involved, implementation-dependent properties is still an open problem.

Modular reasoning is difficult in this setting, because it combines two programming features that are independently difficult to reason about, but absolutely fiendish together: higher-order functions and polymorphic monads. Independently, they can be tackled with equational reasoning, free theorems and

monad laws, but these tools have only limited traction on higher-order functions over polymorphic monads.

Inspired by Hofmann et al.'s work [6,2] on characterizing pure monadic higher-order functions, this paper shows how to eliminate the polymorphism and the higher-order parameter for a range of side effects. Hence, we characterize polymorphic monadic mixin components with monomorphically-typed first-order representations. We believe that these representations are inherently more convenient for reasoning.

As this is work in progress, and deriving the first order representation is challenging in its own right, modular reasoning is not covered.

2 Motivating Example

This is the core infrastructure for mixin components:

```
type Open s = s → s
new :: Open s → s
new a = a (new a)
(⊕) :: Open s → Open s → Open s
a1 ⊕ a2 = a1 ∘ a2
```

A mixin component of type Open s has a hole of type s for the recursive occurrences. This hole is closed with fixpoint combinator new. Two mixin components are composed with ⊕.

Here is an example, the Fibonacci function rendered as a polymorphic monadic mixin component:

```
fib :: Monad m ⇒ Open (Int → m Int)
fib rec n | n < 2       = return n
fib rec n | otherwise = do x ← rec (n − 1)
                           y ← rec (n − 2)
                           return (x + y)
```

As m can be any monad, fib does not rely on any side effect. We say that fib is a *pure component*. The (slow) Fibonacci function is recovered by closing the open component and instantiating m to the identity monad:

```
slowfib n = runId (new fib n)
```

Note that this paper uses the Monatron library [8] for monads and monad transformers. Monatron provides superior lifting capabilities compared to other monad transformer libraries that we will use later in this paper.

A faster implementation of the Fibonacci function is obtained with the help of memoization. The memoization functionality can be captured orthogonally in its own component:

```
memo :: (Eq k, StateM [(k, v)] m) ⇒ Open (k → m v)
memo super n =
  do t ← get
     case lookup n t of
       Just v  → return v
       Nothing → do v ← super n
                    modify ((n, v):)
                    return v
```

This component uses the state side effect, which is documented in the constraint StateM [(k, v)] m. We obtain an $\mathcal{O}(n^2)$-time Fibonacci function by composing memo with fib, closing the open recursion and running the resulting function in the State monad.

```
fastfib n = evalState [] (new (memo ⊕ fib) n)
```

2.1 Modular Reasoning

To show the validity of applying memoization, the following property must be established:

$$\text{slowfib} \quad \equiv \quad \text{fastfib}$$

Oliveira et al. [14] prove this property in a non-modular fashion: they consider the composition memo ⊕ fib as a whole. Our goal is to reason about this composition modularly: We want to 1) establish properties about memo and fib based on their respective implementations but independent of one another, and 2) combine these properties to establish the above property without further reliance on the implementations.

The individual properties should be reusable for a establishing a range of properties:

1. The memo mixin is non-interfering for any pure component f :: Monad m ⇒ Open (i → m o):

$$\text{evalState}\ [\,]\ (\text{new}\ (\text{memo} \oplus \text{f})\ \text{n}) \quad \equiv \quad \text{runId}\ (\text{new}\ \text{f}\ \text{n})$$

2. The application of other mixins to fib, that implement alternative memoization techniques, is correct.
3. Adding particular additional effects, e.g., logging, does not affect the correctness of memoization:

$$\text{fst}\ \$\ \text{evalState}\ [\,]\ \$\ \text{runWriterT}\ (\text{new}\ (\text{log} \oplus \text{memo} \oplus \text{fib})\ \text{n}) \quad \equiv \quad \text{slowfib n}$$

This is very challenging, because different uses require different instantiations of the monad type parameter m, and the implementation-dependent property must cater to all at once. This rules out expanding the definitions of the monadic

operations for any particular instance. A second complicating factor is the function parameter for recursive or super calls. Its particular behavior (side-effects and returned value) depend on the composition. In the non-monadic setting, we can formulate preconditions on the input-output behavior of the parameter function. However, in this monadic setting, and due to the polymorphism, the function parameter may have access to more or other side-effects than the mixin itself. It is not clear how to impose any preconditions on these unknown side-effects or how to factor them into the overall behavior of the mixin.

3 Strategy Trees for Pure Functionals

The starting point of this paper are the results of Bauer et al. [2], who have studied the characterization of pure functionals of type Func i o a:

```
type Func i o a = ∀m.Monad m ⇒ (i → m o) → m a
```

They show that such functionals "can be seen as strategies in a question-answer game leading to the computation of" the result. These strategies are reified in *strategy trees*:

```
data Tree i o a  where
  Ans :: a → Tree i o a
  Que :: i → (o → Tree i o a) → Tree i o a
```

A strategy tree Tree i o a for a pure functional of type Func i o a is a sequence of queries Que resulting in an answer Ans. Bauer et al. show that there exists a strategy tree for every functional, and that it is constructed by fun2tree:

```
fun2tree :: Func i o a → Tree i o a
fun2tree f = runCont Ans (f (λi → cont (λk → Que i k)))
```

Moreover, tree2fun recovers the functional.

```
tree2fun :: Tree i o a → Func i o a
tree2fun t f = go t  where
  go (Ans a)   = return a
  go (Que i k) = f i >>= (go ∘ k)
```

Note that the fun2tree and tree2fun functions are each other's inverses:

$$\begin{aligned} \text{tree2fun} \circ \text{fun2tree} &\equiv \text{id} \\ \text{fun2tree} \circ \text{tree2fun} &\equiv \text{id} \end{aligned}$$

3.1 Strategy Trees for Monadic Mixins

It turns out that strategy trees are not restricted to functionals of type Func i o a, but are also suitable for encoding monadic mixins. For instance, fibt defines strategy tree of fib.

```
fibt :: Int → Tree Int Int Int
fibt n | n < 2 = Ans n
fibt n         = Que (n − 1) (λf1 →
                   Que (n − 2) (λf2 →
                   Ans (f1 + f2)))
```

More formally, we can state the following equivalence:

$$\mathsf{fib} \equiv \mathsf{tree2open}\ \mathsf{fibt}$$

where tree2open is a variant of tree2fun that is more convenient for open components:

```
tree2open :: Monad m ⇒ (i → Tree i o o) → Open (i → m o)
tree2open f super i = tree2fun (f i) super
```

More generally, any pure monadic mixin component of type ∀m.Monad m ⇒ Open (i → m o) can be encoded by a strategy tree of type Tree i o o:

```
open2tree :: (∀m.Monad m ⇒ Open (i → m o)) → (i → Tree i o o)
open2tree h i = fun2tree (flip h i)
```

This is great for reasoning, because the latter type is monomorphic, while the former is not.

Unfortunately, this observation far from solves our problems as it does not cover monadic mixins like memo. Unlike fib, memo is not a pure monadic component. On the contrary, it explicitly relies on state to implement its behavior. Thus it cannot be encoded as a strategy tree, and we are no step further to modularly reasoning about components with side effects.

4 Stateful Strategy Trees

In order to make progress on reasoning about memo, this section explores a variant of Bauer's pure strategy trees that caters for functionals of type:

```
type Func_S s i o a = ∀m.StateM s m ⇒ (i → m o) → m a
```

This stateful strategy tree type is:

```
data Tree_S s i o a  where
  Ans_S :: a → s → Tree_S s i o a
  Que_S :: i → s → (o → s → Tree_S s i o a) → Tree_S s i o a
```

A strategy is either an answer Ans_S or a query Que_S. The constructor Ans_S captures the final state alongside the answer. Similarly, Que_S exposes the intermediate state and its continuation takes an updated state. The stateful strategy tree for memo is:

```
memot :: Eq k ⇒ k → [(k, v)] → Tree_S [(k, v)] k v v
memot n t = case lookup n t of
  Just v  → Ans_S v t
  Nothing → Que_S n t (λv t' → Ans_S v ((n, v) : t'))
```

The function stree2fun recovers the functional from its strategy tree representation.

```
stree2fun :: (s → Tree_S s i o a) → Func_S s i o a
stree2fun t f = get ≫= go ∘ t  where
  go (Ans_S a s)   = put s ≫ return a
  go (Que_S i s k) = put s ≫ f i ≫= (λo → get ≫= go ∘ k o)
stree2open :: StateM s m ⇒ (i → s → Tree_S s i o o) → Open (i → m o)
stree2open f super i  = stree2fun (f i) super
```

For instance, we have the following relation between memo and memot:

$$\mathsf{memo} \equiv \mathsf{stree2open\ memot}$$

Unfortunately, this section's approach is far too ad-hoc. It is unclear how the stateful variants of Tree and tree2fun are actually derived, let alone how to do so for the stree2fun's inverse or to adapt the definitions to other kinds of effects. Clearly, we need a more systematic and structured way to derive stateful variants of Bauer et al.'s pure definitions.

5 Strategy Trees as Coroutines

The first step towards a more structured approach is the observation that strategy trees are effectively *coroutines*. In functional programming, they are better known as the coroutine *monad*, aka the resumption monad.

```
instance Monad (Tree i o)  where
  return        = Ans
  Ans x  ≫= f = f x
  Que i k ≫= f = Que i (λo → k o ≫= f)
```

In the co-routine interpretation, the Ans constructor denotes a completed computation, while the Que constructor denotes a suspended computation that can be resumed. A useful primitive operation is suspension:

```
suspend :: i → Tree i a a
suspend i = Que i Ans
```

which enables a very concise conversion from a pure functional to its strategy tree:

```
fun2tree′ :: Func i o a → Tree i o a
fun2tree′ f = f suspend
```

In words, the functional's monad type parameter m is instantiated to the coroutine monad and the function parameter is instantiated to the suspend primitive.

For example, the strategy tree for fib, derived as fun2tree′ fib, can be written in monadic style:

```
fibt′ :: Int → Tree Int Int Int
fibt′ n | n < 2 = return n
fibt′ n         = do f1 ← suspend (n − 1)
                     f2 ← suspend (n − 2)
                     return (f1 + f2)
```

6 Coroutines with Effects

The next step in our structured approach is to reconcile the coroutine monad with other side effects, like state for memo. The solution to combine two effects, co-routines and another one, is entirely standard: monad transformers.

Thus, the coroutine monad transformer replaces the coroutine monad. It is defined as follows:

```
newtype CorT i o m a = CorT {runCorT :: m (C i o m a)}
data C i o m a
  = Done a
  | Suspend i (o → CorT i o m a)
instance MonadT (CorT i o) where
  lift = CorT ∘ liftM Done
  tbind m f =
    CorT (do x ← runCorT m
             case x of
               Done y      → runCorT $ f y
               Suspend i k → return (Suspend i (λo → k o `tbind` f)))
susp :: Monad m ⇒ i → CorT i o m o
susp i = CorT $ return (Suspend i return)
```

From Polymorphic to Monomorphic Form We proceed in a two steps:

1. Use the coroutine transformer to transform a higher-order monadic function into a first-order one. This happens by partially instantiating the monad type parameter m to CorT n, where n takes care of other side effects.[1]

[1] Every Monad is also a Functor, though this is not explicit at the type class level. The Functor constraint is needed shortly.

```
step₁ :: (∀m.(Functor m, Monad m) ⇒ (i → m o) → m a)
      → (∀n.Monad n ⇒ CorT i o n a)
step₁ f = f susp
```

2. Eliminate the polymorphic type variable n by instantiating it to a concrete implementation. In the case the constraint on n is just Monad n, like for fib, the canonical instantiation is the identity monad – there are no side effects.

```
step₂ :: (∀n.Monad n ⇒ CorT i o n a) → CorT i o Id a
step₂ m = m
```

In summary, we obtain a monomorphic characterization as follows:

```
poly2mono :: (∀m.(Functor m, Monad m) ⇒ (i → m o) → m a)
          → CorT i o Id a
poly2mono m = step₂ (step₁ m)
```

For instance, here is the monomorphic characterization of the Fibonacci function:

```
fibt″ :: Int → CorT Int Int Id Int
fibt″ n = poly2mono (flip fib n)
```

From Monomoprhic to Polymorphic Form The opposite transformation is similar two step process.

1. Replace the concrete monad implementation by a type parameter, by means of a monad morphism. In the example, the monad morphism id2any does this:

```
type m ⇝ n = ∀a.m a → n a
id2any :: ∀n.Monad n ⇒ Id ⇝ n
id2any = return ∘ runId
```

The morphism is applied through the CorT transformer by means of Monatron's library function:

```
tmap :: (FMonadT t, Functor m, Functor n) ⇒ (m ⇝ n) → t m a → t n a
```

In summary, this step consists of:

```
step₃ :: CorT i o Id a → (∀n.(Functor n, Monad n) ⇒ CorT i o n a)
step₃ m = tmap id2any m
```

2. Eliminate the suspension transformer again, by means of:

```
step₄ :: (∀m.(Functor m, Monad m) ⇒ CorT i o m a)
      → (∀m.(Functor m, Monad m) ⇒ (i → m o) → m a)
```

```
step4 m f = go m  where
  go m = do x ← runCorT m
            case x of
              Done y      → return y
              Suspend i k → f i >>= go ∘ k
```

Hence, the reverse process is:

```
mono2poly :: CorT i o Id a → (∀m.(Functor m, Monad m) ⇒ (i → m o) → m a)
mono2poly m = step4 (step3 m)
```

Bringing everything together, we claim that poly2mono and mono2poly are each other's inverses:

$$\text{mono2poly} \circ \text{poly2mono} \equiv \text{id}$$
$$\text{poly2mono} \circ \text{mono2poly} \equiv \text{id}$$

which means that the monomorphic representation is isomorphic to the polymorphic one, and thus a useful characterization.

7 Coroutines with Algebraic Effects

This section adapts the two-step approach to two different side effects, state and non-determinism, that expose only *algebraic* operations. Jaskelioff [8] calls a monadic operation op *algebraic* if its type has the form ∀a.F a → M a, where F is some functor. Algebraic operations are easily lifted with lift∘op::∀a.F a → T M a.

7.1 Stateful Coroutines

The only changes in step1 and step2 are the replacement of Monad by StateM s, and of Id by State s:

```
step1,S      :: (∀m.(Functor m, StateM s m) ⇒ (i → m o) → m a)
             → (∀n.StateM s n ⇒ CorT i o n a)
step1,S f = f susp
step2,S :: (∀n.StateM s n ⇒ CorT i o n a) → CorT i o (State s) a
step2,S m = m
poly2monoS :: (∀m.(Functor m, StateM s m) ⇒ (i → m o) → m a)
  → CorT i o (State s) a
poly2monoS m = step2,S (step1,S m)
```

Note that step1,S works because Monatron's infrastructure lifts the get and put operations from n to CorT i o n.

Now poly2monoS yields a monomorphically typed variant of memo:

```
memot'' :: Eq k ⇒ k → CorT k v (State [(k, v)]) v
memot'' k = poly2mono_S (flip memo k)
```

Note that if we specialize the type CorT i o (State s) a, we indeed obtain s → Tree$_S$ i o a, i.e., the ad-hoc stateful strategy tree we postulated earlier.

In the other direction, memo is recovered from memo'' by means of step$_{3,S}$ and step$_{4,S}$, of which the former uses the state2any morphism and the latter differs from step$_4$ merely in its signature.

```
state2any :: StateM s m ⇒ State s ⇝ m
state2any m = do s_0 ← get
                 let (x, s_1) = runState s_0 m
                 put s_1
                 return x

step_{3,S} :: CorT i o (State s) a → (∀n.(Functor n, StateM s n) ⇒ CorT i o n a)
step_{3,S} m = tmap state2any m

step_{4,S} :: (∀m.(Functor m, StateM s m) ⇒ CorT i o m a)
           → (∀m.(Functor m, StateM s m) ⇒ (i → m o) → m a)
step_{4,S} m f = go m where
  go m = do x ← runCorT m
            case x of
              Done y      → return y
              Suspend i k → f i >>= go ∘ k

mono2poly_S :: CorT i o (State s) a
            → (∀m.(Functor m, StateM s m) ⇒ (i → m o) → m a)
mono2poly_S m = step_{4,S} (step_{3,S} m)
```

We claim again that poly2mono$_S$ and mono2poly$_S$ are each other's inverses:

$$\text{mono2poly}_S \circ \text{poly2mono}_S \equiv \text{id}$$
$$\text{poly2mono}_S \circ \text{mono2poly}_S \equiv \text{id}$$

The WriterM and ReaderM effects with their respective algebraic operations tell and ask are treated in a similar fashion.

7.2 Non-Deterministic Coroutines

Another interesting effect with algebraic operations is non-determinism, a control-flow effect. Monatron's type class for non-deterministic effects is ListM, which supplies operations mZero :: ListM m ⇒ m a and mPlus :: ListM m ⇒ m a → m a → m a for respectively no solutions and merging of solutions. The key ingredients we need for the approach are a canonical implementation of ListM, the list monad [], and a morphism:

```
list2any :: ListM m ⇒ [] ⇝ m
list2any []       = mZero
list2any (x : xs) = return x `mPlus` list2any xs
```

Following the two-step approach we obtain again two transformations $\mathsf{poly2mono}_L$ and $\mathsf{mono2poly}_L$, which we claim are each other's inverses.

8 Non-Algebraic Operations

Unfortunately, our two-step approach based on CorT breaks down when handling non-algebraic operations. As an example we study exceptions, with type class ExcM x and operations throw :: ExcM m ⇒ x → m a and handle :: ExcM x ⇒ m a → (x → m a) → m a for throwing and handling exceptions. While the throw operation is algebraic, the handle operation is not: it has a type of the form ∀a.F (M a) → M a rather than ∀a.F a → M a.

8.1 The Problem of Exceptions and Coroutines

Following the approach, Exception x[2] serves as the canonical implementation of ExcM x with the following morphism:

```
exc2any :: ∀x m.ExcM x m ⇒ Exception x ⇝ m
exc2any m = either2any (runException m)

either2any :: ∀x m.ExcM x m ⇒ Either x ⇝ m
either2any m = case m of
  Left x  → throw x
  Right a → return a
```

This results in the two operations:

```
poly2mono_X :: (∀m.ExcM x m ⇒ (i → m o) → m a)
            → CorT i o (Exception x) a
mono2poly_X :: CorT i o (Exception x a)
            → (∀m.ExcM x m ⇒ (i → m o) → m a)
```

which turn out not to be inverses because they do not preserve the semantics of handle. The following minimal example illustrates the issue:

```
f :: ∀m.ExcM () m ⇒ (() → m ()) → m ()
f g = g () `handle` return

test_1 = runException $ f (\_ → throw ())
test_2 = runException $ mono2poly_X (poly2mono_X f) (\_ → throw ())


> test_1
Right ()

> test_2
Left ()
```

The reason is twofold:

[2] isomorphic to Either x

1. CorT externalizes the call to the function parameter g, which now happens in $\mathsf{mono2poly}_X$. When an exception is raised during a call, $\mathsf{mono2poly}_X$ aborts and does not communicate it to the functional.
2. The definition of handle, uniformly lifted by Monatron from Exception x to CorT i o (Exception x), exhibits the following property:

   $$\mathsf{handle}\ (\mathsf{m} \gg\!= (\lambda \mathsf{x} \to \mathsf{susp}\ \mathsf{i} \gg\!= \mathsf{f}\ \mathsf{x}))\ \mathsf{h} \equiv (\mathsf{handle}\ \mathsf{m}\ \mathsf{h}) \gg\!= (\lambda \mathsf{x} \to \mathsf{susp}\ \mathsf{i} \gg\!= \mathsf{f}\ \mathsf{x})$$

   In words, susp extracts itself and any subsequent computations from under handle. Hence, any exceptions that arise in susp or later, cannot be caught.

Of the two problems, the latter might be amended by customizing the lifting of handle for CorT. However, the former problem is more fundamental: CorT leaves no room for communicating errors back, which makes it altogether usesless for our purpose.

8.2 Ad-hoc Solution

Fortunately, we can resort to a more ad-hoc solution again, and directly formulate a suitable monad that externalizes function calls, but preserves the desired semantics of handle.

```
data Tree_X x i o a
  = Return_X a
  | Suspend_X i (Either x o → Tree_X x i o a)
  | Raise x
```

This datatype has the two conventional constructors for: Return_X for returning an answer immediatly and $\mathsf{Suspend}_X$ for suspending the computation. Note that $\mathsf{Suspend}_X$'s continuation has a parameter of type Either x o rather than o. This way the external call can communicate its possible failure to the continuation. The reify_X function enables any external monad to reify its exception as an Either x o value

```
reify_X :: ExcM x m ⇒ m a → m (Either x a)
reify_X m = handle (m >>= return ∘ Right) (return ∘ Left)
```

and susp_X translates this communicated exception to the appropriate internal form:

```
susp_X :: i → Tree_X x i o o
susp_X i = Suspend_X i either2any
```

Raise x denotes throw x, a computation that results in exception x.

```
instance Monad (Tree_X x i o) where
  return = Return_X
  Return_X a    >>= f = f a
```

```
Suspend_X i k >>= f = Suspend_X i (λo → k o >>= f)
Raise x       >>= f = Raise x
```

We spare the reader the particular infrastructure-related details of ExcM in Monatron and summarize Tree_X x i o's implementations of throw and handle.

As indicated above, throw is represented by Raise:

```
throw x = Raise x
```

The following two laws cover the Return_X and Raise cases of handle:

```
handle (return a) h ≡ return a
handle (throw x) h ≡ h x
```

while $\mathsf{Suspend}_X$ defers handle to its continuation. Combined, we obtain the following definition of handle:

```
handle (Return_X a)     h = Return_X a
handle (Suspend_X i k) h = Suspend_X i (λo → handle (k o) h)
handle (Raise x)        h = h x
```

Now, the conversion from a functional to Tree_X is essentially the same as before:

```
fun2tree_X :: (∀m.ExcM x m ⇒ (i → m o) → m a)
           → Tree_X x i o a
fun2tree_X f = f susp_X
```

However, the inverse is more involved, interpreting the Tree_X constructors in the target monad:

```
tree2fun_X :: ∀m x i o a.ExcM x m ⇒ Tree_X x i o a → ((i → m o) → m a)
tree2fun_X m f = go m  where
  go :: ∀c.Tree_X x i o c → m c
  go (Return_X a)     = return a
  go (Suspend_X i k) = reify_X (f i) >>= go ∘ k
  go (Raise x)        = throw x
```

This approach does preserve the semantics, as we can observe in the following example:

```
test_{2,X} = runException (tree2fun_X (fun2tree_X f) (\_ → throw ()))
```

and, in contrast to test_2, we have:

```
> test_{2,X}
Right ()
```

Generally, we claim that $\mathsf{fun2tree}_X$ and $\mathsf{tree2fun}_X$ are inverses.

9 Related Work

Monadic Reasoning Hutton [7] advises to remove the abstraction layer of operations like $\gg=$, get and handle, and to reason in terms of their concrete implementations. Unfortunately, when the monad is a polymorphic type variable, we cannot apply this approach directly. However, the aim in this paper is to obtain a monomorphic characterization of the monadic code first, in order to be able to reason about its concrete implementation.

Both of *parametricity* [13,16] and algebraic laws of monadic operations enable reasoning about polymorphic monads. Voigtländer [15] shows how to derive parametricity theorems for type constructor classes such as Monad. Many people [10,5] introduced and studied various algebraic laws to enable reasoning about monadic operations directly. Oliveira et al. [11] even combine parametricity and algebraic laws to reason about the non-interference of polymorphic monadic mixin components. However, their approach is not strong enough to establish the correctness of memoization, which relies on non-trivial invariants, in a modular fashion.

Unfortunately, the standard form of parametricity and algebraic laws are not powerful enough for our purpose due to the presence of the unknown function parameter. Hofmann et al. [6,2] perform custom logical relations reasoning to characterize pure higher-order functions in terms of *strategy trees*. This paper extends their work to impure higher-order functions in order to enable reasoning about polymorphic monadic mixin components.

The Coroutine Monad Different variants of the coroutine monad (transformer) [3] have been studied under different names: resumption monad [12], free monad [1] and step monad [9]. For our purposes the name *coroutine* is most suitable because it emphasizes that the computation is split into two parts, the internal part of the mixin component and the external part of the function parameter.

10 Conclusion

We have shown how to characterize polymorphic monadic mixin components as monomorphically typed first order definitions for a range of different side effects. For effects with only algebraic operations we provide a particularly systematic two-step approach based on the coroutine monad transformer. In future work, we aim to show how the monomorphic characterizations facilitate modular reasoning and reuse of proofs. Moreover, we will investigate how to extend the systematic approach to non-algebraic operation like exception handling.

References

1. Awodey, S.: Category Theory, Oxford Logic Guides, vol. 49. Oxford University Press, Oxford (2006)

2. Bauer, A., Hofmann, M., Karbyshev, A.: On monadic parametricity of second-order functionals (July 2012), preproceedings of IFL'12
3. Blazevic, M.: monad-coroutine: Coroutine monad transformer for suspending and resuming monadic computations (2010), http://hackage.haskell.org/package/monad-coroutine
4. Cook, W.R.: A Denotational Semantics of Inheritance. Ph.D. thesis, Brown University (1989)
5. Gibbons, J., Hinze, R.: Just do it: Simple monadic equational reasoning. In: ICFP'11: 16th International Conference on Functional Programming. pp. 2–14 (2011)
6. Hofmann, M., Karbyshev, A., Seidl, H.: What is a pure functional? In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P. (eds.) Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 6199, pp. 199–210. Springer Berlin / Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-14162-1_17, 10.1007/978-3-642-14162-1_17
7. Hutton, G., Fulger, D.: Reasoning About Effects: Seeing the Wood Through the Trees. In: Proceedings of the Symposium on Trends in Functional Programming. Nijmegen, The Netherlands (2008)
8. Jaskelioff, M.: Monatron: an extensible monad transformer library. In: Proceedings of the 20th international conference on Implementation and application of functional languages. pp. 233–248. IFL'08, Springer-Verlag, Berlin, Heidelberg (2011)
9. Jaskelioff, M., Moggi, E.: Monad transformers as monoid transformers. Theor. Comput. Sci. 411(51-52), 4441–4466 (Dec 2010)
10. Liang, S., Hudak, P.: Modular denotational semantics for compiler construction. In: ESOP'96: European Symposium on Programming. pp. 219–234 (1996)
11. Oliveira, B.C.d.S., Schrijvers, T., Cook, W.R.: EffectiveAdvice: disciplined advice with explicit effects. In: AOSD'10: 9th International Conference on Aspect-Oriented Software Development. pp. 109–120 (2010)
12. Papaspyrou, N.S.: A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. Technical Report CSD-SW-TR-2-01, National Technical University of Athens (2001)
13. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress. pp. 513–523 (1983)
14. d. S. Oliveira, B.C., Schrijvers, T., Cook, W.R.: MRI: Modular Reasoning about Interference in Incremental Programming (2012), http://users.ugent.be/~tschrijv/Research/papers/mri.pdf
15. Voigtländer, J.: Free theorems involving type constructor classes. In: ICFP'09: 14th International Conference on Functional Programming. pp. 173–184 (2009)
16. Wadler, P.: Theorems for free! In: FPLCA'89: 4th International Conference on Functional Programming and Computer Architecture, pp. 347–359 (1989)
17. Wadler, P.: Monads for functional programming. In: Program Design Calculi: Marktoberdorf International Summer School on Program Design Calculi (1992)

A Notation for Comonads

Dominic Orchard and Alan Mycroft

Computer Laboratory, University of Cambridge
{firstname}.{lastname}@cl.cam.ac.uk

Abstract. The category-theoretic concept of a *monad* is applied widely as a design pattern for functional programs involving effects. The utility and ubiquity of monads is such that some languages provide syntax to simplify use of this pattern. Comonads, the dual of monads, can be similarly used as a design pattern in functional programming, yet remain relatively under-utilised compared with monads. There are several useful examples of comonads but a lack of syntactic sugar prevents wider adoption and indeed further understanding of comonads as an abstraction mechanism. We propose a lightweight syntax for programming with comonads in Haskell, analogous to the do-notation for monads, accompanied by examples of comonads in the notation.

1 Introduction

The flourishing approach of *categorical programming* applies concepts from category theory as design patterns for abstracting and structuring programs. For example, the category theoretic notion of a *monad* is widely used to structure programs with *side effects* [16, 17]. A *monadic* data type $M\,a$ has accompanying operations which provide composition of functions with *structured output*, *i.e.*, of type $a \to M\,b$, where effects are seen as *impure output* behaviour of a function, encapsulated or encoded in the data type M.

Monads are so effective as an abstraction technique that some languages provide a lightweight syntactic sugar simplifying programming with monads, such as the **do**-notation in Haskell and the *let!* notation in F# [12].

Comonads are the *dual* structure to *monads* and can be used to abstract *context dependence* in programs [14]. A *comonadic* data type $C\,a$ has accompanying operations for the composition of functions with *structured input*, *i.e.*, of type $C\,a \to b$, where context-dependence is seen as *impure input* behaviour of a function, encapsulated/encoded in the data type C. There are various examples of comonads in programming in the literature, for dataflow programming with streams [15], attribute evaluation [13], array computations [11], and more [6]. However, despite these example, comonads are less widely used than monads.

One reason for the relative underuse of comonads in programming is that the known examples appear less diverse compared to monads. Another reason is that, unlike monads, there is no language support to simplify programming with comonads, thus the use and experimentation of comonads as a design pattern in

programming is impeded. To remedy this situation, this paper proposes a syntax for programming with comonads in Haskell, called the **codo**-notation.
In Haskell, comonads are defined by the following class:[1]

```
class Comonad c where
  extract :: c a → a
  extend :: (c a → b) → c a → c b
```

If $c\ a$ types context-dependent computations of a value a, then *extract* defines the *current* context, extracting the value at this context; *extend* defines the range of *all possible contexts*, extending a *local operation* – which computes a value b from the context of $c\ a$ – to a *global* operation by applying it at all contexts. Thus comonads abstract boilerplate code for the iteration involved in extending an operation defined at one context to all contexts.

For example, the type of *arrays* paired with an array index, called the *cursor*, is a comonad, where *extract* accesses the cursor element of the array, thus the cursor denotes the current context, and *extend* applies its parameter function to the parameter array with its cursor set to each index in the domain of the array, computing an array of results [11]. Thus, an array is a value dependent upon its position in the array. Many applications in image processing, scientific computing, cellular automata, and graphics use this pattern of computation.

The **codo**-notation simplifies programming with operations defined on such arrays. For example, the following **codo**-block defines an operation for computing the contours in a 2D-image using a *difference of Gaussians*-style approach:

```
contours :: CArray (Int, Int) Float → Float
contours = codo x ⇒ y ← gauss2D x
                    z ← gauss2D y
                    w ← (extract y) − (extract z)
                    laplace2D w
```

where *CArray i a* models the cursored-array comonad, with index type i and element type a, and $gauss2D, laplace2D :: CArray\ (Int, Int)\ Float \to Float$ compute, at a particular context, the *discrete Gaussian* and *Laplace* operators. A contour image can thus be computed by applying (*extend contours*) to an image.

Section 2 introduces the **codo**-notation in more detail, continuing with arrays as an example. The notation has a simple translation into the operations of a comonad (described in Section 4) which provides an equational theory for the notation following from the laws of a comonad (Section 3).

The **codo**-notation is analogous to the **do**-notation for programming with monads in Haskell, but with some notable differences which will be explained from the point of view of *categorical semantics* in Section 5. Comonads and **codo**-notation are also related to concepts in natural language semantics, which will be discussed along with concluding remarks in Section 6.

The **codo**-notation is currently provided using macros in GHC as a library available at: `http://github.com/dorchard/codo-notation`.

[1] Available via Kmett's `Control.Comonad` package.

Array example The array example will be used throughout the next section to introduce **codo**. It is defined in Haskell by the following data type and instance:

$$\begin{aligned}
&\textbf{data}\ \mathit{CArray}\ i\ a = \mathit{CA}\ (\mathit{Array}\ i\ a)\ i\\
&\textbf{instance}\ \mathit{Ix}\ i \Rightarrow \mathit{Comonad}\ (\mathit{CArray}\ i)\ \textbf{where}\\
&\quad \mathit{extract}\ (\mathit{CA}\ a\ c) = a\,!\,c\\
&\quad \mathit{extend}\ f\ (\mathit{CA}\ a\ c) = \textbf{let}\ es' = \mathit{map}\ (\lambda i \to (i, f\ (\mathit{CA}\ a\ i)))\ (\mathit{indices}\ a)\\
&\qquad\qquad\qquad\qquad\ \textbf{in}\ \ \mathit{CA}\ (\mathit{array}\ (\mathit{bounds}\ a)\ es')\ c
\end{aligned}$$

where *extract* accesses the cursor element using the array indexing operation !, and, for every index i of the parameter array, *extend* applies f to the array with i as its cursor, returning an index-value pair list from which the result array is constructed. Note, the return and parameter arrays have the same size and cursor *i.e. extend* preserves the incoming context.

Many array operations can be defined as *local operations* – of type $c\ a \to b$ (hereafter *comonadic operations*) – using relative indexing *e.g.* the *laplace2D* operator, for approximating differentiation, can be defined:

$$\begin{aligned}
&\mathit{laplace2D} :: \mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float} \to \mathit{Float}\\
&\mathit{laplace2D}\ a = a\ ?\ (-1, 0) + a\ ?\ (1, 0) + a\ ?\ (0, -1) + a\ ?\ (0, 1) - 4 * a\ ?\ (0, 0)
\end{aligned}$$

where (?) abstracts relative indexing with bounds checking and default values:[2]

$$\begin{aligned}
&(?) :: (\mathit{Ix}\ i, \mathit{Num}\ a, \mathit{Num}\ i) \Rightarrow \mathit{CArray}\ i\ a \to i \to a\\
&(\mathit{CA}\ a\ i)\ ?\ i' = \textbf{if}\ (\mathit{inRange}\ (\mathit{bounds}\ a)\ (i + i'))\ \textbf{then}\ a\,!\,(i + i')\ \textbf{else}\ 0
\end{aligned}$$

Whilst *laplace2D* applied to an image computes the Laplacian operator at a single context (locally), *extend laplace2D* applied to an image computes the Laplacian at every context (globally), returning the resulting image.

2 Introducing *codo*

The **codo**-notation provides a form of *let*-binding for composing comonadic operations, of the form and type (here p ranges over patterns and e over expressions):

$$(\textbf{codo}\ p\ \Rightarrow\ \overline{p \leftarrow e};\ e) :: \mathit{Comonad}\ c \Rightarrow c\ a \to b$$

Compare this with the general form and type of the monadic **do**-notation:

$$(\textbf{do}\ \overline{p \leftarrow e}; e) :: \mathit{Monad}\ m \Rightarrow m\ a$$

Both comprise zero or more binding statements of the form $p \leftarrow e$, preceding a final result expression. A ***codo**-block* however defines a function, with a pattern for the function's parameter following the **codo** keyword. The parameter is essential since comonads describe computations with *structured input*. A **do**-block is instead a (nullary) expression. Section 5 compares the two notations in detail.

[2] There are many alternative methods for abstracting boundary checking and values; our choice here is for simplicity of presentation rather than performance or accuracy.

Comonads and ***codo****-notation for composition* Comonadic functions, with structured input, can be composed using *extend*:

$$\begin{aligned}&(\hat{\circ}) :: \mathit{Comonad}\ c \Rightarrow (c\ y \to z) \to (c\ x \to y) \to c\ x \to z\\ &g\ \hat{\circ}\ f = g \circ (\mathit{extend}\ f)\end{aligned} \tag{1}$$

The laws of a comonad are equivalent to requiring that this composition is *associative* and that *extract* is the identity (discussed further in Section 3).

The **codo**-notation abstracts use of *extend* in composition of comonadic operations. For example, we may wish to compose two array operations:

$$\mathit{lapGauss2D} = (\mathit{laplace2D} \circ (\mathit{extend}\ \mathit{gauss2D})) :: \mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float} \to \mathit{Float}$$

which can written equivalently in the **codo**-notation:

$$\begin{aligned}&\mathit{lapGauss2D} :: \mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float} \to \mathit{Float}\\ &\mathit{lapGauss2D} = \mathbf{codo}\ x \Rightarrow y \leftarrow \mathit{gauss2D}\\ &\qquad\qquad\qquad\qquad\ \ \mathit{laplace2D}\ y\end{aligned}$$

where $x :: \mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float}$ and $y :: \mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float}$.

Within a **codo**-block the *context* of the computation is set by the context of the incoming parameter; all subsequent local variables are *synchronised* at this incoming context. Thus, inside the above block, y is at the same context as x, *i.e.*, they have the same cursor.

For a variable pattern parameter, a **codo**-block is typed by the rule: (here typing rules are presented with a single colon : for the typing relation)

$$[\text{varP}]\frac{\Gamma,\ x : c\ t \vdash_c e : t'}{\Gamma \vdash (\mathbf{codo}\ x \Rightarrow e) : \mathit{Comonad}\ c \Rightarrow c\ t \to t'}$$

where $\vdash_c$ types binders of a **codo**-block, *e.g.*, a variable-pattern binding is typed:

$$[\text{varB}]\frac{\Gamma \vdash_c e : t \qquad \Gamma,\ x : c\ t \vdash_c e' : t'}{\Gamma \vdash_c x \leftarrow e;\ e' : t'}$$

Variables bound within the **codo**-block (the *local variables*) are the inputs of further expressions in the block. The presence of the top-level parameter means that every expression in the block has at least one input. The interpretation of binding an expression is thus the comonadic composition of this expression as a function from its inputs with the interpretation of the rest of the block. Thus in [varB] the binding $x \leftarrow e : t$ implies $x : c\ t$ in the rest of the block.

The typing rules for **codo**-notation are collected in **Figure 2**.

Multi-parameter operations The following defines a *pointwise difference* between two comonadic values, which is polymorphic in the comonad:

$$\begin{aligned}&\mathit{minus} :: (\mathit{Comonad}\ c, \mathit{Num}\ a) \Rightarrow c\ a \to c\ a \to a\\ &\mathit{minus}\ x\ y = \mathit{extract}\ x - \mathit{extract}\ y\end{aligned}$$

Multi-parameter comonadic functions can be composed with other operations naturally using **codo**-notation, *e.g.*:

$$\begin{aligned} \mathit{contours}' = \mathbf{codo}\ x \Rightarrow\ & y \leftarrow \mathit{gauss2D}\ x \\ & z \leftarrow \mathit{gauss2D}\ y \\ & w \leftarrow \mathit{minus}\ y\ z \\ & \mathit{laplace2D}\ w \end{aligned}$$

(equivalent to *contours* in the introduction which inlined the definition of *minus*).

Without **codo**, multi-parameter comonadic operations are significantly more difficult to compose. The equivalent program without **codo** is:

$$\begin{aligned} \mathit{contours}'\ x = \mathbf{let}\ & y = \mathit{extend}\ \mathit{gauss2D}\ x \\ & w = \mathit{extend}\ (\lambda y' \rightarrow \mathbf{let}\ z = \mathit{extend}\ \mathit{gauss2D}\ y' \\ & \qquad\qquad\qquad\qquad\ \mathbf{in}\ \ \mathit{minus}\ y'\ z)\ y \\ \mathbf{in}\ & \mathit{laplace2D}\ w \end{aligned}$$

where y' and z have the same cursor, thus $\mathit{minus}\ y'\ z$ is the pointwise difference.

The following is an incorrect use of *minus* for the pointwise difference:

$$\begin{aligned} \mathit{contour\_bad}\ x = \mathbf{let}\ & y = \mathit{extend}\ \mathit{gauss2D}\ x \\ & z = \mathit{extend}\ \mathit{gauss2D}\ y \\ & w = \mathit{extend}\ (\mathit{minus}\ y)\ z \\ \mathbf{in}\ & \mathit{laplace2D}\ w \end{aligned}$$

Applying the partially-applied *minus* y to z by *extend* means *minus* y is applied to z at all its contexts whilst the context of y is fixed. Thus w is not the pointwise difference, but the difference of y at a fixed context with the values of z. The **codo**-notation therefore simplifies the use of multi-parameter operations, avoiding incorrect programs caused by *un*synchronised cursors. The unusual behaviour of *contour_bad* could be encoded with **codo** using nested **codo**-blocks:

$$\begin{aligned} \mathit{contour}'' = \mathbf{codo}\ x \Rightarrow\ & y \leftarrow \mathit{gauss2D}\ x \\ & (\mathbf{codo}\ y' \Rightarrow z \leftarrow \mathit{gauss2D}\ y' \\ & \qquad\qquad\qquad\ \ w \leftarrow \mathit{minus}\ y\ z \\ & \qquad\qquad\qquad\ \ \mathit{laplace2D}\ w)\ y \end{aligned}$$

where y in $\mathit{minus}\ \ y\ \ z$ is bound in the outer **codo** and thus has its cursor fixed, whilst z is bound in the inner **codo** and has its cursor varying. Using a comonadic variable bound outside of the nearest enclosing **codo**-block means that the variable is unsynchronised with respect to the variables inside the block.

A **codo**-block may have multiple parameters via tuple patterns ([tupP], **Figure 2**) *e.g.*, the following Laplace-transforms and pointwise-adds two arguments:

$$\begin{aligned} & \mathit{lapPlus} :: \mathit{CArray}\ \mathit{Int}\ (\mathit{Float}, \mathit{Float}) \rightarrow \mathit{Float} \\ & \mathit{lapPlus} = \mathbf{codo}\ (x, y) \Rightarrow\ a \leftarrow \mathit{laplace2D}\ x \\ & \qquad\qquad\qquad\qquad\qquad\ \ b \leftarrow \mathit{laplace2D}\ y \\ & \qquad\qquad\qquad\qquad\qquad\ \ (\mathit{extract}\ a) + (\mathit{extract}\ b) \end{aligned}$$

The type of *lapPlus* shows that, instead of two parameters, it takes a single comonadic parameter with tuple elements, of the form $c\ (a, b)$. However, inside the block $x :: c\ a$ and $y :: c\ b$ as the desugaring of **codo** *unzips* the parameter (see Section 4). The intention is that x and y are synchronised in their cursors.

To apply *lapPlus* to a pair of arrays, its arguments must first be *zipped*, provided by the *czip* operation:

$$\textbf{class}\ \mathit{ComonadZip}\ c\ \textbf{where}\ \mathit{czip} :: (c\ a, c\ b) \to c\ (a, b)$$

For *CArray*, *czip* can be defined:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Eq}\ i, \mathit{Ix}\ i) \Rightarrow \mathit{ComonadZip}\ (\mathit{CArray}\ i)\ \textbf{where}\\ \quad \mathit{czip}\ (\mathit{CA}\ a\ c, \mathit{CA}\ a'\ c') =\\ \quad\quad \textbf{if}\ (c \not\equiv c')\ \textbf{then}\ \mathit{error}\ \texttt{"Cursors must be equal"}\\ \quad\quad \textbf{else let}\ es'' = \mathit{map}\ (\lambda i \to (i, (a\,!\,i, a'\,!\,i)))\ (\mathit{indices}\ a)\\ \quad\quad\quad\ \textbf{in}\ \mathit{CA}\ (\mathit{array}\ (\mathit{bounds}\ a)\ es'')\ c\end{array}$$

Thus only arrays of the same shape and cursor can be zipped together. In the contextual understanding, the two parameter arrays are synchronised in their contexts. The example of *lapPlus* can therefore be applied to two array parameters x and y by: *extend lapPlus* (*czip* (x, y)).

A tuple pattern can also be used in a binding statement, typed by rule [tupB] (**Figure 2**). For example, the following is equivalent to *lapPlus* by exchanging a parameter binding with a statement binding (see Section 3 for the general law):

$$\begin{array}{l}\mathit{lapPlus} = \textbf{codo}\ z \Rightarrow (x, y) \leftarrow \mathit{extract}\ z\\ \qquad\qquad\qquad\qquad\ \ a \leftarrow \mathit{laplace2D}\ x\\ \qquad\qquad\qquad\qquad\ \ b \leftarrow \mathit{laplace2D}\ y\\ \qquad\qquad\qquad\qquad\ \ (\mathit{extract}\ a) + (\mathit{extract}\ b)\end{array}$$

Example: labelled graphs Many graph algorithms can be structured by a comonad, particular compiler analyses and transformations on *control flow graphs* (*CFG*s). The following defines a labelled-graph comonad as a list of nodes which are pairs of a label and a list of their connected vertices:

$$\begin{array}{l}\textbf{data}\ \mathit{LGraph}\ a = \mathit{LG}\ [(a, [\mathit{Int}])]\ \mathit{Int}\\ \textbf{instance}\ \mathit{Comonad}\ \mathit{LGraph}\ \textbf{where}\\ \quad \mathit{extract}\ (\mathit{LG}\ xs\ c) = \mathit{fst}\ (xs\ !!\ c)\\ \quad \mathit{extend}\ f\ (\mathit{LG}\ xs\ c) = \mathit{LG}\ (\mathit{map}\ (\lambda c' \to (f\ (\mathit{LG}\ xs\ c'), \mathit{snd}\ (xs\ !!\ c)))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [0\,..\,\mathit{length}\ xs])\ c\end{array}$$

The *LGraph*-comonad resembles the array comonad where contexts are positions with a *cursor* denoting the current position. Analyses over *CFGs* can be defined using graphs labelled by syntax trees *Stmt*. For example, a *live-variable* analysis can be defined, using the **codo**-notation, as:

$$\begin{array}{lll}\mathit{lva} = \textbf{codo}\ g \Rightarrow \mathit{lv0} \leftarrow (\mathit{defUse}\ g, [\,]) & \text{-- compute definition/use sets, paired}\\ \qquad\qquad\qquad\ \ \mathit{lva}'\ \mathit{lv0} & \text{-- with initial empty live-variable set}\end{array}$$

$$
\begin{aligned}
lva' = \textbf{codo}\ & ((def, use), lv) \Rightarrow \\
& live\_out \leftarrow foldl\ union\ [\,]\ (successors\ lv) \\
& live\_in\ \ \leftarrow union\ (extract\ def)\ ((extract\ live\_out) \setminus\!\setminus (extract\ use)) \\
& lvp\ \ \ \ \ \leftarrow ((extract\ def, extract\ use), extract\ live\_in) \\
& lvNext\ \leftarrow lva'\ lvp \\
& \textbf{if}\ (lv \equiv live\_in)\ \textbf{then}\ (extract\ lv)\ \textbf{else}\ (extract\ lvNext)
\end{aligned}
$$

where *union* and set difference ($\setminus\!\setminus$) on lists have type $Eq\ a \Rightarrow [a] \to [a] \to [a]$ and $defUse :: LGraph\ \ Stmt\ \to ([Var], [Var])$ computes the sets of variables defined and used by each block in a CFG. The analysis is recursive, refining the set of live variables until a fixed-point is reached.

The live variables for every block of a *CFG* can be computed by *extend lva*.

Costate, trees, and zippers Arrays were used to introduce comonads and **codo** to aid understanding since the notion of *context* is made clear by the *cursor index*. The above graph example has a similar form. Both are instance of a general comonad, often called the *costate* comonad, whose data type is a pair of a function from contexts to values and a particular context: $Ca = (s \to a) \times s$.

For both arrays and labelled graphs the type of contexts is a finite domain of integer, or integer-tuple, indices. For labelled graphs, the costate comonad is combined with *product comonad* (see [14]) pairing the label of a node with the list of its successors, thus the type is isomorphic to $Ca = (s \to (a \times [s])) \times s$.

For *costate*, the notion of context is explicitly provided by a cursor acting as a *pointer* or *address*. This is not the only way to define a notion of context. Other data types encode the context structurally rather than using a cursor. For example, a comonad of labelled binary trees can be defined:

$$
\begin{aligned}
&\textbf{data}\ BTree\ a = Leaf\ a \mid Node\ a\ (BTree\ a)\ (BTree\ a) \\
&\textbf{instance}\ Comonad\ BTree\ \textbf{where} \\
&\quad extract\ (Leaf\ a) = a \\
&\quad extract\ (Node\ a\ l\ r) = a \\
&\quad extend\ f\ (Leaf\ a) = Leaf\ (f\ (Leaf\ a)) \\
&\quad extend\ f\ t@(Node\ a\ l\ r) = Node\ (f\ t)\ (extend\ f\ l)\ (extend\ f\ r)
\end{aligned}
$$

The action of *extend* is to apply its parameter function f to successive suffix trees, thus f can only access its children, not its parents. Thus *coextend* not only defines what it means for a local (comonadic) operation to be applied globally, but also which contexts are *accessible* from each possible context.

A tree comonad that has a structural notion of context but whose comonadic operations can access any part of the tree can be defined using Huet's *zipper* data type, where trees are split into a path to the current position and the remaining parts of the tree [5]. For a certain class of data types it has been shown that a *zipper* like structure can be automatically derived by *differentiation* of the data type [8]. All zippers are comonads, thus any data type with a corresponding zipper can be turned into a comonad where the notion of context is encoded in the structure of the type, rather than by a *pointer*-like cursor.

$$\boxed{\Gamma \vdash \textbf{codo}\ \ p \Rightarrow e : \mathit{Comonad}\ c\ t \to t'} \qquad \boxed{\Gamma \vdash_c \overline{p \leftarrow e};\ e : t}$$

$$[\text{varP}]\frac{\Gamma,\ x : c\ t \vdash_c e : t'}{\Gamma \vdash (\textbf{codo}\ \ x \Rightarrow e) : c\ t \to t'} \qquad [\mathit{varB}]\frac{\Gamma \vdash_c e : t \quad \Gamma,\ x : c\ t \vdash_c e' : t'}{\Gamma \vdash_c x \leftarrow e;\ e' : t'}$$

$$[\mathit{tupP}]\frac{\Gamma,\ x : c\ t,\ y : c\ t' \vdash_c e : t''}{\Gamma \vdash (\textbf{codo}\ \ (x, y) \Rightarrow e) : c\ (t, t') \to t''} \qquad [\text{tupB}]\frac{\Gamma \vdash_c e : (t_1, t_2) \quad \Gamma,\ x : c\ t_1, y : c\ t_2 \vdash_c e' : t'}{\Gamma \vdash_c (x, y) \leftarrow e_1;\ e' : t'}$$

$$[\text{wildP}]\frac{\Gamma \vdash_c e : t}{\Gamma \vdash (\textbf{codo}\ \ \_ \Rightarrow e) : c\ a \to t} \qquad [\mathit{letB}]\frac{\Gamma, \vdash_c e : t \quad \Gamma, x : t \vdash_c e' : t'}{\Gamma \vdash_c \textbf{let}\ x = e;\ e' : t'}$$

Fig. 1. Typing rules for **codo**

3 Equational Theory

As shown in Section 2, *extend* provides composition for comonadic operations (1). The laws of a comonad are exactly the laws that guarantee this composition is *associative* and has a *left* and *right unit*, provided by *extract*, summarised here:

$$\begin{array}{rlcll}
\text{(right unit)} & f \mathbin{\hat{\circ}} \mathit{extract} \equiv f & \rightsquigarrow & \mathit{extend}\ \mathit{extract} \equiv \mathit{id} & [C1] \\
\text{(left unit)} & \mathit{extract} \mathbin{\hat{\circ}} f \equiv f & \rightsquigarrow & \mathit{extract} \circ (\mathit{extend}\ f) \equiv f & [C2] \\
\text{(associativity)} & h \mathbin{\hat{\circ}} (g \mathbin{\hat{\circ}} f) & \rightsquigarrow & \mathit{extend}\ g \circ \mathit{extend}\ f & \\
& \equiv (h \mathbin{\hat{\circ}} g) \mathbin{\hat{\circ}} f & & \equiv \mathit{extend}\ (g \circ \mathit{extend}\ f) & [C3]
\end{array}$$

As there is no mechanism for enforcing such rules in Haskell the programmer is expected to verify the laws on their own.

Since **codo** is desugared into just the operations of a comonad, the comonad laws therefore imply equational rules on the syntax of **codo**-notation, shown in **Figure 3**(b). **Figure 3**(a) shows rules for the top-level **codo**-block which follow from its translation.

The operation: $\mathit{czip} :: (c\ a, c\ b) \to c\ (a, b)$ introduced in Section 2 corresponds to that of a *(semi)-monoidal functor* which may satisfy various laws with respect to the comonad (see the discussion of *(semi)-monoidal comonads* in [14]). The following property, which we call *idempotency* of a semi-monoidal functor, frequently holds of comonad/*czip* implementations:

$$\mathit{czip}\ (x, x) \equiv \mathit{cmap}\ (\lambda y \to (y, y))\ x \tag{2}$$

This property implies syntactic laws on **codo** which relate tuple patterns and *czip* (**Figure 3**(c)). For every rule involving a tuple pattern there is an equivalent rule derived from the *parameter/statement exchange* rule (**Figure 3**(a)).

Comonads are functors The category theoretic notion of a *functor* can be used to abstract *map*-like operations on parametric data types. In Haskell, functors are described by the *Functor* type class, of which *map* provides the list instance:

(a) Pure rules

$$\textbf{codo}\ x \Rightarrow f\ x \equiv f \qquad (\textbf{codo}\text{-}\eta)$$

$$\textbf{codo}\ x \Rightarrow z \leftarrow (\textbf{codo}\ y \Rightarrow e_1)\ x \equiv \textbf{codo}\ x \Rightarrow y \leftarrow \mathit{extract}\ x,\ z \leftarrow e_1,\ e_2 \qquad (\textbf{codo}\text{-}\beta)$$

$$\textbf{codo}\ z \Rightarrow (x, y) \leftarrow \mathit{extract}\ z \equiv \textbf{codo}\ (x, y) \Rightarrow e \quad \textit{(parameter/statement binding exchange)}$$

(b) Comonad laws

$$\textbf{codo}\ x \Rightarrow y \leftarrow \mathit{extract}\ x,\ f\ y \equiv \textbf{codo}\ x \Rightarrow f\ x \qquad [C1]$$

$$\textbf{codo}\ x \Rightarrow y \leftarrow f\ x,\ \mathit{extract}\ y \equiv \textbf{codo}\ x \Rightarrow f\ x \qquad [C2]$$

$$(\textbf{codo}\ x \Rightarrow z \leftarrow (\textbf{codo}\ y \Rightarrow w \leftarrow e_1,\ e_2)\ x,\ e_3) \equiv \textbf{codo}\ x \Rightarrow y \leftarrow \mathit{extract}\ x,\ w \leftarrow e_1,\ z \leftarrow e_2,\ e_3 \qquad [C3]$$

$$(\textit{iff } x \textit{ is not free in } e_1) \equiv \textbf{codo}\ y \Rightarrow w \leftarrow e_1,\ (\textbf{codo}\ x \Rightarrow z \leftarrow e_2,\ e_3)\ y$$

(c) Additional rules – if idempotency (2) holds

$$\textbf{codo}\ (b, c) \Rightarrow z \leftarrow (\mathit{extract}\ b, \mathit{extract}\ c),\ f\ z \equiv \textbf{codo}\ (b, c) \Rightarrow f\ (\mathit{czip}\ (b, c))$$

$$\textbf{codo}\ x \Rightarrow (a', b') \leftarrow \mathit{czip}\ (a, b),\ f\ a'\ b' \equiv \textbf{codo}\ x \Rightarrow f\ a\ b$$

Fig. 2. Equational laws for the **codo**-notation

$$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\ \mathit{fmap} :: (a \to b) \to f\ a \to f\ b$$
$$\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where}\ \mathit{fmap} = \mathit{map}$$

All comonads are functors by the following definition using *extend* and *extract*:

$$\mathit{cmap} :: \mathit{Comonad}\ c \Rightarrow (a \to b) \to c\ a \to c\ b$$
$$\mathit{cmap}\ f\ x = \mathit{extend}\ (f \circ \mathit{extract})$$

While *fmap* applies its parameter function to a single element, *extend* applies its parameter function to a subset (possibly the whole) of the parameter structure. Thus *extend* generalises *fmap*.

Shape preservation An interesting derived property of comonads is *shape preservation*. The *shape* of a data type is its structure without any values i.e.

$$\mathit{shape} = \mathit{cmap}\ (\mathit{const}\ ())$$

where $\mathit{const}\ x = \lambda\_ \to x$. For example, the *shape* of a list is just the *cons*-cells, without any values, or in this case *unit* values ().

A consequence of the comonad laws is that, for any comonadic function f, $(extend\, f)$ preserves the shape of the incoming structure in its result. For example, with the array comonad, *extend* preserves the size, cursor, and dimensions of the parameter array in the result. This *shape preservation* property is proved in Appendix A, and stated formally is that, for all $f :: c\ a \to b$:

$$shape \circ (extend\, f) \equiv shape \tag{3}$$

4 Desugaring *codo*

The translation of **codo** is based on Uustalu and Vene's semantics for a context-dependent λ-calculus [14] and has two parts: translation of binding into composition via *extend*, and management of the environment for *local variables* bound in a **codo**-block. The first part is explained by considering a restricted **codo**-notation, which only ever has one local variable, bound in the previous statement.

1). Single-variable contexts For some comonad C, consider the **codo**-block:

$$foo1 = (\mathbf{codo}\ x \Rightarrow y \leftarrow f\ x;\ \ g\ y) :: C\ x \to z$$

where $f :: C\ x \to y$, $g :: C\ y \to z$. The first statement $y \leftarrow f\ x$ can be interpreted as a function with parameter x and body $f\ x$, the second, which is the final result expression, can be similarly interpreted as a function from y to its expression:

$$(\lambda x \to f\ x) :: C\ x \to y \tag{4}$$

$$(\lambda y \to g\ y) :: C\ y \to z \tag{5}$$

Both (4) and (5) are functions with structured input, thus the semantics of *foo1* is the comonadic composition of (4) and (5):

$$[\![foo1]\!] = (\lambda y \to g\ y) \circ (extend\ (\lambda x \to f\ x)) : Cx \to z.$$

2). Multiple-variable contexts The **codo**-notation however provides multi-variable contexts, allowing the following example with binary function $h :: Cx \to Cy \to z$:

$$foo2 = (\mathbf{codo}\ x \Rightarrow y \leftarrow f\ x;\ \ h\ x\ y) :: C\ x \to z$$

The first statement cannot be interpreted as before since the second statement uses both x and y, thus the interpretation must return x with the result of $f\ x$:

$$(\lambda x \to (extract\ x, f\ x)) :: C\ x \to (x, y) \tag{6}$$

Applying *extract* to x means that *extend* (6), of type $C\ x \to C\ (x, y)$, returns the parameter x and the result of $f\ x$ synchronised in their contexts.

The interpretation of the second statement is a function taking a value $C\ (x, y)$ and *unzipping* it, binding the constituent values to x and y in the scope of the result expression, where x and y are synchronised at the same context since *cmap* preserves the contextual properties of the comonad:

$$\begin{aligned}(\lambda env \to \mathbf{let}\ x &= cmap\ fst\ env \\ y &= cmap\ snd\ env\ \mathbf{in}\ h\ x\ y) :: C\ (x, y) \to z\end{aligned} \tag{7}$$

The translation of *foo2* is therefore $[\![foo2]\!] = (7) \circ (extend\ (6))$.

4.1 General construction

The translation traverses the list of binding statements in a **codo**-block, accumulating a comonadic environment of the local variables bound so far. The accumulated environment is structured by right-nested two-tuples (pairs) terminated by an empty tuple. Thus, the actual translation of *foo2* is:

$$\begin{aligned}[\![\mathit{foo2}]\!] = (\lambda \mathit{env} \to \mathbf{let}\ & y = \mathit{cmap}\ \mathit{fst}\ \mathit{env} \\ & x = \mathit{cmap}\ (\mathit{fst} \circ \mathit{snd})\ \mathit{env}\ \mathbf{in}\ h\ x\ y) \\ \circ\ & (\mathit{extend}\ (\lambda \mathit{env} \to (\mathbf{let}\ x = \mathit{cmap}\ \mathit{fst}\ \mathit{env}\ \mathbf{in}\ f\ x, \mathit{extract}\ \mathit{env}))) \\ \circ\ & (\mathit{cmap}\ (\lambda \mathit{env} \to (\mathit{env}, ())))\end{aligned}$$

For *foo2*, the environment in the first statement contains just x and has type $C\ (x, ())$, and in the second statement contains x and y and has type $C\ (y, (x, ()))$.

The top-level translation of a **codo**-block is defined:

$$\begin{aligned}[\![\mathbf{codo}\ x \Rightarrow b]\!] &= [\![x \vdash b]\!]_c \circ (\mathit{cmap}\ (\lambda x \to (x, ()))) \\ [\![\mathbf{codo}\ \_ \Rightarrow b]\!] &= [\![\vdash b]\!]_c \circ (\mathit{cmap}\ (\lambda x \to (x, ()))) \\ [\![\mathbf{codo}\ (x, y) \Rightarrow b]\!] &= [\![x, y \vdash b]\!]_c\ \circ \mathit{cmap}\ (\lambda p \to (\mathit{fst}\ p, (\mathit{snd}\ p, ())))\end{aligned}$$

where $[\![\Delta \vdash b]\!]_c$ is the translation for binding statements b within a **codo**-block with the context of the local variables Δ.

The top-level translation generalises easily to arbitrary tuple patterns. In each case, $[\![-]\!]_c$ is pre-composed with a function converting tuples in the incoming parameter of the **codo**-block to right-nested tuples, terminated by the empty tuple $()$. Translation of binding statements yields a Haskell function of type:

$$[\![\Delta \vdash \overline{b};\ e]\!]_c : \mathit{Comonad}\ c \Rightarrow c\ (t_1, (\ldots, (t_n, ()))) \to t$$

where $e : t$ and $\Delta = v_1, \ldots, v_n$ where $v_i : t_i$. The definition of $[\![-]\!]_c$ is:

$$\begin{aligned}[\![\Delta \vdash e]\!]_c &= [\![\Delta \vdash e]\!]_{\mathit{exp}} \\ [\![\Delta \vdash x \leftarrow e; e']\!]_c &= [\![x, \Delta \vdash e']\!]_c\ \circ \mathit{extend}\ (\lambda \mathit{env} \to ([\![\Delta \vdash e]\!]_{\mathit{exp}}\ \mathit{env}, \mathit{extract}\ \mathit{env})) \\ [\![\Delta \vdash (x, y) \leftarrow e; e']\!]_c &= [\![x, y, \Delta \vdash e']\!]_c\ \circ \mathit{extend}\ (\lambda \mathit{env} \to (\lambda((x, y), \Delta) \to (x, (y, \Delta))) \\ &\qquad ([\![\Delta \vdash e]\!]_{\mathit{exp}}\ \mathit{env}, \mathit{extract}\ \mathit{env}))\end{aligned}$$

where $[\![\Delta \vdash e]\!]_{\mathit{exp}}$ translates expressions on the right-hand side of a binder or the last expression of a block. The last case translates binding to tuple patterns where $\lambda((x, y), \Delta) \to (x, (y, \Delta)))$ reformats results into the right-nested tuple format of the environment; this generalises in the obvious way to arbtirary tuples.

The translation of expressions unzips the incoming comonadic environment, binding the values to the variables in Δ with a local *let*-binding:

$$[\![v_1, \ldots, v_n \vdash e]\!]_{\mathit{exp}} = \lambda \mathit{env} \to \mathbf{let}\ [v_i = \mathit{cmap}\ (\mathit{fst} \circ \mathit{snd}^{i-1})\ \mathit{env}]_1^n\ \mathbf{in}\ e$$

where snd^k means k compositions of *snd* and $\mathit{snd}^0 = \mathit{id}$.

Section 5 compares **codo**-notation with **do**-notation, and explains why the translation of **codo**-notation is relatively more complex.

5 Comparing do- and codo-notations

While comonads and monads are dual, the **codo**- and **do**-notation (for programming with monads) do not appear dual. Both provide *let*-binding syntax, for composition of comonadic and monadic operations respectively. However, **codo**-blocks are parameterised, of type $c\ a \to b$ for a comonad c, whilst **do**-blocks are unparameterised, of type $m\ a$ for a monad m. Since comonads abstract functions with structured input the parameter to a **codo**-block is important. In the **do**-notation, expressions have implicit input via their free variables and Haskell's scoping mechanism is reused for handling local variables in a **do**-block.

The **codo**- and **do**-notation can be seen as internal domain-specific languages, for contextual and effectful computations respectively, with their semantics defined by translation to Haskell. This perspective is similar to the approach of *categorical semantics*, where typed programs are given a denotation as a *morphism*,[3] in some category, mapping from the inputs of a program to the outputs. The disparity between **codo**- and **do**-notation is illuminated by this approach.

Categorical semantics For the simply-typed λ-calculus, the traditional approach pioneered by Lambek and Scott recursively maps the *type derivation* of an expression to a morphism [7]:

$$[\![\Gamma \vdash e : \tau]\!] : ([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \longrightarrow [\![\tau]\!]$$

where $\Gamma = x_1 : \tau_1, \ldots x_n : \tau_n$. Thus, an expression $e : \tau$ with a *context* of free-variable typing assumptions Γ is modelled as a morphism from a *product* of the types for the free variables, as inputs, to the result type as the output. From now on, $[\![-]\!]$ brackets will be elided on types in morphisms for brevity.

Categorical semantics for effectful computations Moggi showed that effectful computations can be given a semantics in terms of a *Kleisli* category which has morphisms with *structured output* of type $a \to m\ b$ for a monad m [9, 10], with denotations (where again $\Gamma = x_1 : \tau_1, \ldots x_n : \tau_n$):

$$[\![\Gamma \vdash e : \tau]\!] : ([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \longrightarrow m\ [\![\tau]\!]$$

In Moggi's calculus, *let*-binding follows the traditional approach of internalising substitution (whether this is call-by-value or call-by-name with respect to effects depends on the particular monad) corresponding to composition of the denotations, provided by the *bind* operation of a monad. However, multi-variable environments requires a *strong monad* which has an additional operation called *strength*. The effectful semantics for *let*-binding is:

(where $f : a \to b$ is written $a \xrightarrow{f} b$ and composition is expressed by concatenation of two arrows)

$$\frac{[\![\Gamma \vdash e : \tau]\!] = g : \Gamma \to m\ \tau \qquad [\![\Gamma, x : \tau \vdash e' : \tau']\!] = g' : \Gamma \times \tau \to m\ \tau'}{[\![\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau']\!] : \Gamma \xrightarrow{\langle id, g \rangle} \Gamma \times m\,\tau \xrightarrow{strength} m\,(\Gamma \times \tau) \xrightarrow{bind\ g'} m\,\tau'} \quad (8)$$

[3] *Morphisms* generalise the notion of function. Readers unfamiliar with category theory may safely replace 'morphism' with 'function' here.

where $\langle f, g \rangle$ is the function pairing: $\lambda x \to (f\ x, g\ x)$, *bind* is the prefix version of Haskell's $\gg\!=$ operator and *strength* provides distributivity of $\times$ over m:

$$\begin{aligned} \mathit{strength} &: (a \times m\ b) \to m\ (a \times b) \\ \mathit{bind} &: (a \to m\ b) \to (m\ a \to m\ b) \end{aligned}$$

The **do**-notation does not provide a full semantics for a language with effects, but instead provides just a semantics for effectful *let*-binding embedded in Haskell. The translation is simplified by reusing Haskell's scoping mechanism since, in Haskell, all monads are *strong* with a canonical *strength* operator:

$$\begin{aligned} &\mathit{strength} :: \mathit{Monad}\ m \Rightarrow (a, m\ b) \to m\ (a, b) \\ &\mathit{strength}\ (a, mb) = mb \gg\!= (\lambda b \to \mathit{return}\ (a, b)) \end{aligned}$$

It is straightforwardly proved that this definition of *strength* satisfies the properties of a *strong monad* (see [9] for these properties). The standard translation of **do** can be derived from (8) by inlining the above definition of *strength* and simplifying according to the monad laws:

$$\frac{\Gamma \vdash e : m\ \tau \qquad \Gamma, x : \tau \vdash e' : m\ \tau'}{\Gamma \vdash [\![\mathbf{do}\ x \leftarrow e; e']\!] : m\ \tau' \ \equiv\ \Gamma \vdash e \gg\!= (\lambda x \to e') : m\ \tau'}$$

giving a translation using just the monad operations and Haskell's scoping mechanism to define the semantics of multi-variable contexts for effectful *let*-binding. Thus the inputs to effectful computations are handled implicitly and so a **do**-block is just an expression of type $m\ a$.

Categorical semantics for contextual computations The dual of Moggi's semantics interprets expressions in a *coKleisli category* whose morphisms have *structured input* ($c\,a \to b$ for a comonad c). The denotations are given by:

$$[\![\Gamma \vdash e : \tau]\!] : c\,([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \longrightarrow [\![\tau]\!]$$

Uustalu and Vene gave a semantics in this form for a *context-dependent* calculus [14]. For a comonadic semantics the context of free-variables is therefore not a product structure, but a comonadic structure over products, unlike the monadic semantics with just the product structure over the context (matching Haskell's semantics). Thus, the scoping mechanism of Haskell cannot be reused as Haskell does not provide *comonadic* structuring of the variables in the context. There is no therefore no general, correct notion of strength for a comonad that can be defined using just the operations of a comonad and Haskell's scoping.

Thus, for the **codo**-notation the multi-variable comonadic contexts must be modelled explicitly as seen in the translation of **codo**-notation (Section 4) resulting in the relatively more complicated translation of **codo**-notation compared with that of **do**-notation, equivalent to:

$$\frac{[\![\Gamma \vdash e : \tau]\!] = g : c\ \Gamma \to\ \tau \qquad [\![\Gamma, x : \tau \vdash e' : \tau']\!] = g' : c\,(\Gamma \times \tau) \to \tau'}{[\![\Gamma \vdash \texttt{let}\ x =\ e\ \texttt{in}\ e' : \tau']\!] :\ c\ \Gamma \xrightarrow{\mathit{extend}\langle \mathit{current}, g \rangle} c\,(\Gamma \times \tau) \xrightarrow{g'} \tau'}$$

6 Discussion and Conclusion

***Comonads, codo-notation, and* intensionality** In a pure language, *let*-binding traditionally internalises substitution, *i.e.*:

$$\textbf{let } x = e_1 \textbf{ in } e_2 \equiv \{e_1/x\}e_2$$

In the same way, the *let*-binding of the **codo**-notation describes substitution for context-dependent computations. This substitution has the property, related to *shape preservation*, that context-dependent computations are always substituted at the context determined by the top-most binding (the parameter to **codo**).

In natural language semantics, similar principles of substitution under context have been studied for centuries by logicians and philosophers. Notably, Frege's influential 1892 paper *Über Sinn und Bedeutung* (On Sense and Meaning), discussed problems of equality between terms that have the same *reference* (*denotation* à la Russell) but different *sense*, or *meaning*, for which *context* affects the validity of substitution for apparently equal terms [4].

Consider the following two true statements (under the ruling of the International Astronomical Union for the official number of planets in our solar system):

(1) *The number of planets is 8*
(2) *Kepler believed that 8 is the third power of two.*

In (1), *the number of planets* and *8* have the same *reference*. However, substituting *the number of planets* for *8* in (2) changes the truth of the statement:[4]

Kepler believed that the number of planets is the third power of two.

The problem is that *the number of planets* and *8* differ in meaning; the denotation of *the number of planets* is altered by the *context*: *Kepler believed that....*

Carnap used the terms *intension*, for the entire sense/meaning of a term, and *extension* for a reference or denotation, describing intensions as functions from states (contexts) to extensions [3]. Carnap's approach resembles the *exponent comonad* $C\ a = s \to a$ [14], mapping from contexts to values, where the operations of a comonad define composition/substitution for *intensions*.

Consider another statement:

Kepler believed the number of planets equals the number of popes since Gregory XIII.

Using Carnap's terminology, both *the number of popes since Gregory XIII* and *the number of planets* are *intensional* terms affected by the *context* of Kepler's belief at a particular time. In the temporal context of 1596, the sentence is true since there had been six popes since Gregory XIII (inclusive). Both intensional terms (number of planets/popes) can be substituted for their extensions at the same context of Kepler's beliefs in 1596, preserving the truth of the statement:

[4] Kepler's *Mysterium Cosmographicum*, published in 1596, describes the *six* known planets at the time.

Kepler believed 6 equals 6.

Substituting intensions at different contexts to the implicit context of Kepler's belief in 1596 likely changes the truth value of the statement *e.g.* substituting the first in 1596 and the second in 1605 renders the statement false:

Kepler believed 6 equals 7.

Thus the substitution is only valid when both are substituted under the same context of Kepler's belief, analogous to the principle of substitution of variables in the same *synchronised* context in the **codo**-notation.

Other applications There are many interesting comonads which have not been explored here. For example, the semantics of the Lucid dataflow language are captured by an *infinite stream comonad* [15], which was used by Uustalu and Vene to define an interpreter for Lucid in Haskell. Using **codo**-notation, Lucid can be embedded directly into Haskell as an internal DSL.

Many comonadic data types are instances of the general concept of *containers*. *Containers* comprise a set of *shapes* S and, for each shape $s \in S$, a type of *positions* Ps, with the data type $C\,a = \sum_{s \in S}(Ps \to a)$, *i.e.*, a coproduct of functions from positions to values for each possible shape [1]. Ahman et al. recently showed that all *directed containers* (those with notions of sub-shape) are comonads, where positions are contexts and sub-shapes define accessibility between contexts for the definition of *extend* [2]. The labelled binary-tree example in Section 2 can be described as a directed-container comonad. The *costate* comonad can be generalised to *cursored containers* with type $C\,a = \sum_{s \in S}(Ps \to a) \times Ps$.

Whilst the **codo**-notation was developed here in Haskell, it could be applied in other languages with further benefits. For example, a **codo**-notation for ML could be used to abstract laziness using a *delayed-computation* comonad with data type $C\ a = () \to a$, or defining lazy lists using the stream comonad [15].

Concluding remarks Comonads, and the **codo**-notation, essentially abstract boilerplate code for iteration, allowing succinct definition of operations defined *locally* abstracting their promotion to *global* operations.

The simple and natural notation provided by **codo** presented here considerably simplifies programming with comonads. It is our hope that this prompts the use of comonads as a design pattern and tool for abstraction and that it promotes further exploration of comonads yielding new and interesting examples.

Acknowledgements Thanks are due to Jeremy Gibbons, Ralf Hinze, Tomas Petricek, Tarmo Uustalu, and Varmo Vene for helpful discussions, and to anonymous reviewers for their comments on an earlier draft. This research was supported by an EPSRC Doctoral Training Award.

References

1. M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

2. D. Ahman, J. Chapman, and T. Uustalu. When is a container a comonad? *Foundations of Software Science and Computational Structures*, pages 74–88, 2012.
3. Rudolf Carnap. Meaning and necessity. 1947.
4. Melvin Fitting. Intensional logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2011 edition, 2011.
5. G. Huet. The zipper. *Journal of Functional Programming*, 7(05):549–554, 1997.
6. Richard B. Kieburtz. Codata and Comonads in Haskell, 1999.
7. J. Lambek and P.J. Scott. *Introduction to higher-order categorical logic*. Cambridge Univ Pr, 1988.
8. C. McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, 2001.
9. Eugenio Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS'89*, pages 14–23. IEEE, 1989.
10. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
11. Dominic Orchard, Max Bolingbroke, and Alan Mycroft. Ypnos: declarative, parallel structured grid programming. In *DAMP '10*, pages 15–24, NY, USA, 2010. ACM.
12. Tomas Petricek and Don Syme. Syntax Matters: writing abstract computations in F#. Pre-proceedings of TFP, 2012.
13. Tarmo Uustalu and Varmo Vene. Comonadic functional attribute evaluation. *Trends in Functional Programming-Volume 6*, pages 145–160, 2007.
14. Tarmo Uustalu and Varmo Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203(5):263–284, 2008.
15. Tarmo Uustalu and Varmo Vene. The Essence of Dataflow Programming. *Lecture Notes in Computer Science*, 4164:135–167, Nov 2006.
16. Philip Wadler. The essence of functional programming. In *Proceedings of POPL '92*, pages 1–14. ACM, 1992.
17. Philip Wadler. Monads for functional programming. *Advanced Functional Programming*, pages 24–52, 1995.

A Proof of shape preservation

To prove shape preservation we first prove the following lemma:

$$cmap\ g \circ extend\ f = extend\ (g \circ f) \tag{9}$$

$$\begin{array}{lll} & cmap\ g \circ extend\ f & \\ \equiv & extend\ (g \circ extract) \circ extend\ f & \text{definition of } cmap \\ \equiv & extend\ (g \circ extract \circ extend\ f) & [C3] \\ \equiv & extend\ (g \circ f) \quad \square & [C2] \end{array}$$

The proof of shape preservation (3) is then:

$$\begin{array}{lll} & shape \circ (extend\ f) & \\ \equiv & (cmap\ (const\ ()) \circ (extend\ f) & \text{definition of } shape \\ \equiv & extend\ ((const\ ()) \circ f) & (9) \\ \equiv & extend\ ((const\ ()) \circ extract) & (const\ x) \circ f \lambda equiv\ (const\ x) \circ g \\ \equiv & (cmap\ (const\ ())) \circ (extend\ extract) & (9) \\ \equiv & cmap\ (const\ ()) & [C1] \\ \equiv & shape\ \square & \text{definition of } shape \end{array}$$

On monadic parametricity of second-order functionals

Andrej Bauer[1], Martin Hofmann[2], and Aleksandr Karbyshev[3]

[1] University of Ljubljana, andrej.bauer@andrej.com
[2] Universität München, hofmann@ifi.lmu.de
[3] Technische Universität München, aleksandr.karbyshev@in.tum.de

Abstract. How can one rigorously specify that a given ML-functional, say, $f : (\texttt{int} \to \texttt{int}) \to \texttt{int}$ is pure, i.e., f produces no effects (e.g., changes in a store, raises of exceptions etc.) except those possibly produced by its functional argument? In this paper, we introduce a semantic notion of *monadic parametricity* (*purity*) for second-order functionals. We show that every monadically parametric f admits a question-answer strategy tree representation. We discuss possible applications of this notion, e.g., to the verification of generic fixpoint algorithms. The results are presented in two settings: a total set-theoretic setting and a partial domain-theoretic one. All proofs are formalized by means of the proof assistant Coq.

1 Introduction

The problem under consideration is: how can one rigorously specify that a given ML-functional, say, $f : (\texttt{int} \to \texttt{int}) \to \texttt{int}$ is pure, i.e., f produces no effects (like changes in a store, raises of exceptions etc.) except those possibly produced by its functional argument? Second-order functionals of this type may appear as inputs in different algorithms like generic fixpoint solvers [4, 5] or algorithms for exact integration [12, 18]. Such algorithms may apply a presumably pure input f to an effectful argument in order to observe the intentional behaviour of f and/or control the computation process.

In the paper [8], we addressed this question with regard to functionals of the type $\forall S.(A \to \mathit{State}_S B) \to \mathit{State}_S C$ polymorphic in states (here *State* denotes a state monad). The choice of class of functionals was imposed by our main application — rigorous verification of a generic fixpoint algorithm **RLD** [7]. In [8], we found that the standard notion of relational parametricity [15, 16] was too weak to exclude the snapback functional $f_{\mathrm{snap}} : \forall S.(A \to \mathit{State}_S B) \to \mathit{State}_S B$ defined by

$$f_{\mathrm{snap}}\, S\, k\, s = \mathbf{let}\ (b, s_1) = k\, a_0\, s\ \mathbf{in}\ (b, s)$$

from the class of relationally parametric objects. f_{snap} invokes k but discards the resulting state s_1 restoring the initial one instead, and only the resulting value b is kept. We have introduced the extensional notion of monadic relational

parametricity (purity) and have shown that for every function parametrical in that sense there exists a question-answer strategy tree representation. Intuitively, a strategy tree defines a second-order computation that comprises a sequence of queries to its functional argument resulting in a value while all the effects are propagated from the functional argument only. The functional f_{snap} is not pure in that sense.

Since the strategy tree reflects only a "skeleton" of computation and does not predefine a type of effects, such representation theorem may exist for other types of effectful computations. In this paper, we overcome the type limitation of [8] and prove the theorem for the class of second-order functionals polymorphic in monads from an arbitrary fixed set *Monad* (e.g., a set of monads presented in the programming language) that includes the continuation monad *Cont*, i.e., we focus on functionals of type

$$\text{Func} = \prod_{T \in Monad} (A \to TB) \to TC$$

It is known that every monad can be expressed in terms of continuation and state monads [6], but we do *not* require $State \in Monad$. Note that the two representation results are independent and do not imply each other directly. The interesting corollary from these two theorems is that given a pure functional F polymorphic in state monads in the sense of [8] there exists an implementation of F that does not make use of state monad explicitly and is polymorphic in monads from *Monad*. Such an implementation is defined by a strategy tree for F.

One possible application of the representation result is formal verification of the above mentioned algorithms. For example, when trying to prove correctness of the local fixpoint solver **RLD** we assumed without loss of generality that the input constraint system is given in the form of strategy trees. That allowed us to formulate sufficient pre- and post-conditions for the algorithm and complete the proof by induction. The fundamental lemma then allows us to argue that the functional input is indeed pure if it can be defined in some restricted programming language (with recursion) which is often the case in real-life program analyses.

In section 3, we give a semantical notion of monadic parametricity (purity) and formulate a fundamental lemma for the call-by-value lambda calculus with monadic semantics. In section 4, we define a notion of a strategy tree and show they represent pure functionals of type Func in the total setting. Section 5 provides a similar result in the partial setting. In section 6, we discuss generalizations of purity to other types. In section 7, we discuss application of purity to verification of fixpoint solvers.

All the proofs have been formalized by means of Coq theorem prover [19] and are available for download at `http://www2.in.tum.de/~karbyshev/purity.zip`. We used the development of constructive ω-cpos and inverse-limit construction for solution of recursive domain equations by Benton et al. [3] Our contribution takes around 1500 lines of Coq code.

Acknowledgements. We thank Alex Simpson, University of Edinburgh, for raising an interesting question and fruitful discussions on the topic. We thank Helmut Seidl for his comments on the paper. The third author was supported by GRK 1480.

2 Preliminaries

In the paper, we study a notion of purity for the total set-theoretic setting and the partial domain-theoretic one. We interpret types as sets in the former case and as cpos in the latter case. We use notations $a : X$ and $a \in X$ interchangeably. For sets (cpos) X and Y we write $X \times Y$ for the Cartesian product and $X \to Y$ for a function space of total functions (a cpo of continuous functions). We denote pairs by (x, y), and projections by *fst* and *snd*. We use λ, $\circ$ and juxtaposition for function abstraction, composition and applications, correspondingly. For a family of sets (cpos) $(X_i)_{i\in I}$ we write $\prod_{i\in I} X_i$ for its Cartesian product. If $F \in \prod_{i\in I} X_i$ then $F_i \in X_i$.

Definition 1. *Monad is a triple* $(T, \mathsf{val}_T, \mathsf{bind}_T)$ *where* T *is a* monad constructor *which for every* X *returns a type* TX *of computations over* X *(*TX *is a* cppo *in the partial case), and*

$$\begin{aligned} \mathsf{val}_T^A &: A \to TA \\ \mathsf{bind}_T^{A,B} &: TA \to (A \to TB) \to TB \end{aligned}$$

are monadic operators *such that the following holds:*

- $\mathsf{bind}_T^{A,B}(\mathsf{val}_T^A\, a)(f) = f\, a$*, for every* $a \in A$
- $\mathsf{bind}_T^{A,A}(t)(\mathsf{val}_T^A) = t$
- $\mathsf{bind}_T^{B,C}(\mathsf{bind}_T^{A,B}(t)(f))(g) = \mathsf{bind}_T^{A,C}(t)(\lambda x.\, \mathsf{bind}_T^{B,C}(f\, x)(g))$.

For the partial case, we assume that T *is* strict*, i.e.,*

- $\mathsf{bind}_T^{A,B}\, \bot_{TA} f = \bot_{TB}$.

We tend to omit indices A, B, C *if they are clear from context.*

We denote by $Cont_R$ the *continuation monad* with result type R defined by $Cont_R X = (X \to R) \to R$ and

$$\begin{aligned} \mathsf{val}_{Cont_R}\, x &= \lambda c.c\, x \ , \\ \mathsf{bind}_{Cont_R}\, t\, f &= \lambda c.t(\lambda x.f\, x\, c) \ . \end{aligned}$$

Given a type of states S, we denote by $State_S$ the *state monad* over S with $State_S X = S \to X \times S$, and monadic operations defined by

$$\begin{aligned} \mathsf{val}_{State_S}\, x &= \lambda s.(x, s) \ , \\ \mathsf{bind}_{State_S}\, t\, f &= \lambda s.\mathbf{let}\ (x_1, s_1) \leftarrow t\, s\ \mathbf{in}\ f x_1 s_1 \ . \end{aligned}$$

In the following, we assume that A, B, C, A_i, B_i are sets (cpos). Let *Monad* be a fixed set of monads such that $Cont \in Monad$. We denote

$$\mathrm{Func} = \prod_{T\in Monad} (A \to TB) \to TC \ .$$

3 Purity

We introduce the following abbreviations.

Definition 2. *If X, X' are types then* $\mathrm{Rel}(X, X')$ *denotes the type of binary relations between X and X'.*

- *if X is a type then $\Delta_X \in \mathrm{Rel}(X, X)$ denotes the equality on X;*
- *if $R \in \mathrm{Rel}(X, X')$ and $S \in \mathrm{Rel}(Y, Y')$ then $R \to S \in \mathrm{Rel}(X \to Y, X' \to Y')$ is given by*

$$f\,(R \to S)\,f' \quad \textit{iff} \quad \forall x\,x'.\,xRx' \implies (f\,x)\,S\,(f'\,x')\ ;$$

- *if $R \in \mathrm{Rel}(X, X')$ and $S \in \mathrm{Rel}(Y, Y')$ then $R \times S \in \mathrm{Rel}(X \times Y, X' \times Y')$ is given by*

$$p\,(R \times S)\,p' \quad \textit{iff} \quad \mathit{fst}(p)\,R\,\mathit{fst}(p') \wedge \mathit{snd}(p)\,S\,\mathit{snd}(p')\ .$$

Definition 3. *For cpos X, X' and $R \in \mathrm{Rel}(X, X')$, R is* admissible *if for any chains $\{c_i\}_{i\in\mathbb{N}}$, $\{c'_i\}_{i\in\mathbb{N}}$ such that $c_i\,R\,c'_i$, for all i, holds $(\bigsqcup c_i)\,R\,(\bigsqcup c'_i)$.*

Definition 4. *Fix $T, T' \in \mathit{Monad}$. For every X, X' and $Q \in \mathrm{Rel}(X, X')$ fix a relation $T^{\mathsf{rel}}(Q) \in \mathrm{Rel}(TX, T'X')$. We say that the mapping $(X, X', Q) \mapsto T^{\mathsf{rel}}(Q)$ is an* acceptable monadic relation *if*

- *for all $X, X', Q \in \mathrm{Rel}(X, X')$, $x \in X$, $x' \in X'$,*

$$xQx' \implies (\mathsf{val}_T\ x)\,T^{\mathsf{rel}}(Q)\,(\mathsf{val}_{T'}\ x')\ ;$$

- *for all $X, X', Q \in \mathrm{Rel}(X, X')$, $Y, Y', R \in \mathrm{Rel}(Y, Y')$, $t \in TX$, $t' \in T'X'$, $f : X \to TY$, $f' : X' \to T'Y'$,*

$$t\,T^{\mathsf{rel}}(Q)\,t' \wedge f(Q \to T^{\mathsf{rel}}(R))f' \implies (\mathsf{bind}_T\ t\ f)\,T^{\mathsf{rel}}(R)\,(\mathsf{bind}_{T'}\ t'\ f')\ .$$

In the domain-theoretic setting, we additionally assume that the monadic relation T^{rel} is

- admissible, *i.e., $T^{\mathsf{rel}}(Q)$ is admissible for every admissible $Q \in \mathrm{Rel}(X, X')$,*
- strict, *i.e., $(\bot_{TX}, \bot_{TX'}) \in T^{\mathsf{rel}}(Q)$.*

Definition 5. *A functional $F \in \mathrm{Func}$ is* pure (monadically parametric) *for the set Monad of monads iff*

$$(F_T, F_{T'}) \in (\Delta_A \to T^{\mathsf{rel}}(\Delta_B)) \to T^{\mathsf{rel}}(\Delta_C)$$

holds for all monads $T, T' \in \mathit{Monad}$ and acceptable monadic relations T^{rel} for T, T'.

In the following, we introduce the call by value lambda calculus with monadic semantics and provide a relational interpretation of types and terms. We establish the fundamental lemma of logical relations stating that every well-typed program respects any monadic relation, similar as done in [8].

Define simple types over some set of base types ranged over by o through the grammar

$$\tau ::= o \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \,.$$

Fix an assignment of a set (a cpo, in the partial case) $[\![o]\!]_T$ for each base type o and monad $T \in \mathit{Monad}$. We extend $[\![-]\!]_T$ to all types by putting

$$[\![\tau_1 \times \tau_2]\!]_T = [\![\tau_1]\!]_T \times [\![\tau_2]\!]_T \,, \quad [\![\tau_1 \to \tau_2]\!]_T = [\![\tau_1]\!]_T \to T[\![\tau_2]\!]_T \,.$$

Given a set of constants (ranged over by c) with their types τ^c and variables ranged over by x we define the lambda terms by

$$\begin{aligned} e ::= {} & x \mid c \mid \lambda x.e \mid e_1\, e_2 \mid e.1 \mid e.2 \mid \langle e_1, e_2 \rangle \\ & \mid \text{let } x \leftarrow e_1 \text{ in } e_2 \mid \text{let rec } f(x) = e \end{aligned}$$

with the last rule for recursive definitions in the partial case only. A typing context Γ is a finite map from variables to types. The typing judgement $\Gamma \vdash e : \tau$ is defined by the usual rules:

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \frac{}{\Gamma \vdash c : \tau^c} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.1 : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.2 : \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x \leftarrow e_1 \text{ in } e_2 : \tau_2} \quad \frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{let rec } f(x) = e : \tau_1 \to \tau_2}$$

The term $e : \tau$ is *closed* if $\emptyset \vdash e : \tau$.

For each $T \in \mathit{Monad}$ and c fix an interpretation $[\![c]\!]_T \in [\![\tau^c]\!]_T$. An *environment* for a context Γ and $T \in \mathit{Monad}$ is a mapping η such that $x \in \mathrm{dom}(\Gamma)$ implies $\eta(x) \in [\![\Gamma(x)]\!]_T$. If $\Gamma \vdash e : \tau$ and η is such an environment then we define $[\![e]\!]_T(\eta) \in T[\![\tau]\!]_T$ by the following clauses:

$$\begin{aligned}
&[\![x]\!]_T(\eta) && = \mathsf{val}_T(\eta(x)) \\
&[\![c]\!]_T(\eta) && = \mathsf{val}_T([\![c]\!]_T) \\
&[\![\lambda x.e]\!]_T(\eta) && = \mathsf{val}_T(\lambda v.[\![e]\!]_T(\eta[x \mapsto v])) \\
&[\![e_1\, e_2]\!]_T(\eta) && = \mathsf{bind}_T([\![e_1]\!]_T(\eta))\,(\mathsf{bind}_T([\![e_2]\!]_T(\eta))) \\
&[\![e.i]\!]_T(\eta) && = \mathsf{bind}_T([\![e]\!]_T(\eta))\,(\mathsf{val}_T \circ \pi_i),\ i = 1, 2 \\
&[\![\langle e_1, e_2 \rangle]\!]_T(\eta) && = \mathsf{bind}_T([\![e_1]\!]_T(\eta))(\mathsf{bind}_T([\![e_2]\!]_T(\eta)) \circ \mathit{curry}(\mathsf{val}_T)) \\
&[\![\text{let } x \leftarrow e_1 \text{ in } e_2]\!]_T(\eta) && = \mathsf{bind}_T([\![e_1]\!]_T(\eta))(\lambda v.[\![e_2]\!]_T(\eta[x \mapsto v]))) \\
&[\![\text{let rec } f(x) = e]\!]_T(\eta) && = \mathsf{val}_T(\mathit{fixp}(\lambda h.\lambda v.[\![e]\!]_T(\eta[f \mapsto h][x \mapsto v])))
\end{aligned}$$

where $\mathit{fixp} : \forall D.(D \to D) \to D$ is the least fixpoint operator for cppos, and curry is the currying function.

Definition 6. *Fix monads $T, T' \in Monad$ and an acceptable monadic relation T^{rel} for T, T'. Given a binary relation $[\![o]\!]^{\mathsf{rel}} \in \mathrm{Rel}([\![o]\!]_T, [\![o]\!]_{T'})$ for each base type o, we can associate a relation $[\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \in \mathrm{Rel}([\![\tau]\!]_T, [\![\tau]\!]_{T'})$ with each type τ by the following clauses:*

$$[\![o]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} = [\![o]\!]^{\mathsf{rel}}\,, \quad [\![\tau_1 \times \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} = [\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \times [\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\,,$$
$$[\![\tau_1 \to \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} = [\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \to T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,.$$

The following *parametricity theorem* is immediate from the definition of acceptable monadic relation and the previous one.

Theorem 1. *Fix $T, T' \in Monad$, and an acceptable monadic relation T^{rel} for T, T'. Suppose that $[\![c]\!]_T\, T^{\mathsf{rel}}([\![\tau^c]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![c]\!]_{T'}$ holds for all constants c. If $\emptyset \vdash e : \tau$ then*

$$[\![e]\!]_T\ T^{\mathsf{rel}}([\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\ [\![e]\!]_{T'}\ .$$

Proof. One proves the following stronger statement by induction on typing derivations. Given $\Gamma \vdash e : \tau$ and environments η for Γ and T and η' for Γ and T' then

$$\forall x.\, \eta(x)\, [\![\Gamma(x)]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\, \eta'(x) \quad \text{implies} \quad [\![e]\!]_T(\eta)\, T^{\mathsf{rel}}([\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e]\!]_{T'}(\eta')\ .$$

The assertion of the theorem follows. □

Every well-typed program $\emptyset \vdash e : \tau$ defines a truly polymorphic function of type $\forall T.[\![\tau]\!]_T$ by taking a product over a set of monads. From theorem 1, we obtain

Corollary 1. *Every truly polymorphic $F \in \mathrm{Func}$ implemented in the calculus is monadically parametric.* □

4 Total case

4.1 Strategy trees

Definition 7. *The set of strategy trees Tree is a minimal set generated by constructors:*

- $\mathsf{Ans} : C \to \mathit{Tree}$
- $\mathsf{Que} : A \to (B \to \mathit{Tree}) \to \mathit{Tree}$

Definition 8. *We define function fun2tree* : Func $\to$ *Tree as follows:*

$$\mathit{fun2tree}\, F = F_{Cont_{Tree}}(\mathsf{Que})(\mathsf{Ans})$$

Definition 9. *Given $T \in Monad$, we define the function $\mathit{tree2fun}_T$: Tree $\to$* Func *recursively by*

- $\mathit{tree2fun}_T(\mathsf{Ans}(c)) = \lambda k.\, \mathsf{val}_T\ c$
- $\mathit{tree2fun}_T(\mathsf{Que}(a)(f)) = \lambda k.\, \mathsf{bind}_T(k\, a)(\lambda b.\, \mathit{tree2fun}_T(f\, b)\, k)$.

Define $\mathit{tree2fun}\, t = \prod_{T \in Monad} \mathit{tree2fun}_T\, t$.

Intuitively, the computation defined by a strategy tree can by considered as a sequence of queries to the first-order argument function applied to elements from B followed by an answer in C. The definition ensures that all the monadic effects come only from the argument function. Moreover, $t \in \mathit{Tree}$ defines a pure (monadically parametric) computation.

Lemma 1. *For any $t \in \mathit{Tree}$, $\mathit{tree2fun}\, t$ is pure.*

Proof. Take monads T, T' and an acceptable monadic relation T^{rel} for T, T'. We prove the statement by induction on t:

- $t = \mathsf{Ans}(c)$. It suffices to show that $(\mathsf{val}_T\ c, \mathsf{val}_{T'}\ c) \in T^{\mathsf{rel}}(\Delta_C)$ which holds by the definition of acceptability of T^{rel}.
- $t = \mathsf{Que}(a)(f)$, and assume that

$$\mathit{tree2fun}(f\, b) \text{ is pure for every } b \in B \qquad \text{(IH)}$$

  Take $(k, k') \in \Delta_A \to T^{\mathsf{rel}}(\Delta_B)$. Since T^{rel} is acceptable, it suffices to show that

$$(k\, a, k'\, a) \in T^{\mathsf{rel}}(\Delta_B) \qquad \text{and}$$
$$(\lambda b.\, \mathit{tree2fun}_T(f\, b)\, k, \lambda b.\, \mathit{tree2fun}_{T'}(f\, b)\, k') \in \Delta_B \to T^{\mathsf{rel}}(\Delta_C).$$

  The former holds by the assumption on k and k', and the latter holds by (IH). □

4.2 Representation theorem

In what follows, we prove that the functions *fun2tree* and *tree2fun* are mutually inverse.

Lemma 2. *For any $t \in \mathit{Tree}$, $\mathit{fun2tree}(\mathit{tree2fun}\, t) = t$.*

Proof. By induction on t.

- $t = \mathsf{Ans}(c)$. By definition, $\mathit{tree2fun}(\mathsf{Ans}(c)) = \Lambda T.\lambda k.\, \mathsf{val}_T\ c$.
  Thus, $\mathit{fun2tree}(\mathit{tree2fun}(\mathsf{Ans}(c))) = (\lambda k.\, \mathsf{val}_{Cont_{Tree}}\ c)(\mathsf{Que})(\mathsf{Ans}) = \mathsf{Ans}(c)$.
- $t = \mathsf{Que}(a)(f)$. Assume that

$$\mathit{fun2tree}(\mathit{tree2fun}(f\, b)) = f\, b \quad \text{for all } b \in B \qquad \text{(IH)}$$

  We have $\mathit{tree2fun}(\mathsf{Que}(a)(f)) = \Lambda T.\lambda k.\, \mathsf{bind}_T(k\, a)(\lambda b.\, \mathit{tree2fun}_T(f\, b)\, k)$.
  Therefore,

$$\begin{aligned}
&\mathit{fun2tree}(\mathit{tree2fun}(\mathsf{Que}(a)(f))) = \\
&\quad = (\lambda k.\, \mathsf{bind}_{Cont_{Tree}}(k\, a)(\lambda b.\, \mathit{tree2fun}_{Cont_{Tree}}(f\, b)\, k))(\mathsf{Que})(\mathsf{Ans}) \\
&\quad = (\mathsf{bind}_{Cont_{Tree}}(\mathsf{Que}(a))(\lambda b.\, \mathit{tree2fun}_{Cont_{Tree}}(f\, b)\, (\mathsf{Que})))(\mathsf{Ans}) \\
&\quad = (\mathsf{Que}(a))(\lambda b.\, \mathit{tree2fun}_{Cont_{Tree}}(f\, b)\, (\mathsf{Que})(\mathsf{Ans})) \\
&\quad = (\mathsf{Que}(a))(\lambda b.\, \mathit{fun2tree}(\mathit{tree2fun}(f\, b))) \\
&\quad = \mathsf{Que}(a)(\lambda b.f\, b) \qquad \text{(by (IH))} \\
&\quad = \mathsf{Que}(a)(f)\ .
\end{aligned}$$

□

Our next goal is to prove the reverse statement.

Theorem 2. *For any pure $F \in \mathrm{Func}$,*

$$\mathit{tree2fun}_T(\mathit{fun2tree}\, F) = F_T$$

holds (extensionally) for an arbitrary monad $T \in \mathit{Monad}$.

We first prove that the statement holds for an arbitrary continuation monad.

Lemma 3. *Given F, $\mathit{tree2fun}_{\mathit{Cont}_S}(\mathit{fun2tree}\, F) = F_{\mathit{Cont}_S}$ holds for every S.*

Note that the statement follows from results of Plotkin and Abadi [14]. We give a direct proof below.

Proof. Given a set S and functions $q : A \to (B \to S) \to S$ and $a : C \to S$, we define the *conversion* function $\mathit{conv}_{q,a} : \mathit{Tree} \to S$ by

$$\mathit{conv}_{q,a} = \lambda t.\, \mathit{tree2fun}_{\mathit{Cont}_S}\, t\, q\, a \ .$$

We have:

$$\begin{aligned}
&\mathit{tree2fun}_{\mathit{Cont}_S}(\mathit{fun2tree}\, F) = F_{\mathit{Cont}_S} \\
&\quad\iff \forall q, a.\ \mathit{tree2fun}_{\mathit{Cont}_S}(\mathit{fun2tree}\, F)\, q\, a = F_{\mathit{Cont}_S}\, q\, a \\
&\quad\iff \forall q, a.\mathit{conv}_{q,a}(\mathit{fun2tree}\, F) = F_{\mathit{Cont}_S}\, q\, a \\
&\quad\iff \forall q, a.\mathit{conv}_{q,a}(F_{\mathit{Cont}_{\mathit{Tree}}}(\mathsf{Que})(\mathsf{Ans})) = F_{\mathit{Cont}_S}\, q\, a \\
&\quad\iff \forall q, a.(F_{\mathit{Cont}_{\mathit{Tree}}}(\mathsf{Que})(\mathsf{Ans}), F_{\mathit{Cont}_S}\, q\, a) \in \mathcal{G}_{\mathit{conv}_{q,a}} \ ,
\end{aligned}$$

where $\mathcal{G}_f$ is a *graph* of f, i.e., $(x, y) \in \mathcal{G}_f$ iff $y = f\, x$.

We prove the last proposition by constructing an appropriate monadic relation for $\mathit{Cont}_{\mathit{Tree}}$ and Cont_S and utilizing purity of F. Fix some q and a. For X, X' and $R \in \mathrm{Rel}(X, X')$, we define $T_1^{\mathsf{rel}}(R) \in \mathrm{Rel}(\mathit{Cont}_{\mathit{Tree}} X, \mathit{Cont}_S X')$ by

$$(H, H') \in T_1^{\mathsf{rel}}(R) \quad \text{iff} \quad \forall h, h'.(h, h') \in R \to \mathcal{G}_{\mathit{conv}_{q,a}} \implies (Hh, H'h') \in \mathcal{G}_{\mathit{conv}_{q,a}}$$

T_1^{rel} is an acceptable monadic relation. The proof is straightforward and omitted here. Since F is pure,

$$(F_{\mathit{Cont}_{\mathit{Tree}}}, F_{\mathit{Cont}_S}) \in (\Delta_A \to T_1^{\mathsf{rel}}(\Delta_B)) \to T_1^{\mathsf{rel}}(\Delta_C) \ .$$

Thus, it suffices to check that $(\mathsf{Que}, q) \in \Delta_A \to T_1^{\mathsf{rel}}(\Delta_B)$ and $(\mathsf{Ans}, a) \in \Delta_C \to \mathcal{G}_{\mathit{conv}_{q,a}}$. Indeed, take some $c \in C$. Then $\mathit{conv}_{q,a}(\mathsf{Ans}\, c) = a\, c$ and the latter holds. Take $a_1 \in A$ and $f : X \to \mathit{Tree}, f' : X' \to S$ such that $(f, f') \in \Delta_B \to \mathcal{G}_{\mathit{conv}_{q,a}}$. Then

$$\begin{aligned}
\mathit{conv}_{q,a}(\mathsf{Que}\, a_1\, f) &= \mathit{tree2fun}_{\mathit{Cont}_S}(\mathsf{Que}\, a_1\, f)\, q\, a \\
&= \mathsf{bind}_{\mathit{Cont}_S}(q\, a_1)(\lambda b.\, \mathit{tree2fun}_{\mathit{Cont}_S}(f\, b)\, q)\, a \\
&= (q\, a_1)(\lambda b.\, \mathit{tree2fun}_{\mathit{Cont}_S}(f\, b)\, q\, a) \\
&= (q\, a_1)(\lambda b.\mathit{conv}_{q,a}(f\, b)) \\
&= (q\, a_1)(\lambda b.f'\, b) \\
&= q\, a_1\, f'
\end{aligned}$$

and the former holds. □

By lemma 3, we have $\mathit{tree2fun}_{Cont_{TC}}(F_{Cont_{Tree}}(\mathsf{Que})(\mathsf{Ans})) = F_{Cont_{TC}}$. Using functions

$$\varphi_1 = \mathsf{bind}_T^{B,C} \colon TB \to Cont_{TC}B,$$
$$\varphi_2 = \lambda g.g\,(\mathsf{val}_T^C) \colon Cont_{TC}C \to TC$$

we construct $\Phi_T : ((A \to Cont_{TC}B) \to Cont_{TC}C) \to (A \to TB) \to TC$ as

$$\Phi_T F = \lambda h.\varphi_2(F(\varphi_1 \circ h)) = \lambda h.F(\mathsf{bind}_T^{B,C} \circ h)(\mathsf{val}_T^C) \ .$$

We prove

Lemma 4. *For any pure* $F \in \mathrm{Func}$, $\Phi_T(F_{Cont_{TC}}) = F_T$.

Proof. Again, the idea is to construct a suitable acceptable monadic relation and exploit the purity of F. For X, X', $R \in \mathrm{Rel}(X, X')$, we define $T_2^{\mathsf{rel}}(R) \in \mathrm{Rel}(Cont_{TC}X, TX')$ by

$$(H, H') \in T_2^{\mathsf{rel}}(R) \ \text{ iff } \ \forall h, h', (h, h') \in R \to \Delta_{TC} \implies (Hh)\,\Delta_{TC}\,(\mathsf{bind}_T\,H'\,h') \ .$$

It is straightforward to show that T_2^{rel} is an acceptable monadic relation. We omit the proof. Since F is pure, we have $(F_{Cont_{TC}}, F_T) \in (\Delta_A \to T_2^{\mathsf{rel}}(\Delta_B)) \to T_2^{\mathsf{rel}}(\Delta_C)$. Note that for any $g \colon A \to TB$,

$$\Phi_T(F_{Cont_{TC}})\,g = F_{Cont_{TC}}(\mathsf{bind}_T^{B,C} \circ g)(\mathsf{val}_T^C) \qquad \text{and}$$
$$F_T\,g = \mathsf{bind}_T^{C,C}(F_T\,g)(\mathsf{val}_T^C) \ .$$

First, we show that $(\mathsf{bind}_T^{B,C} \circ g, g) \in \Delta_A \to T_2^{\mathsf{rel}}(\Delta_B)$. Indeed, for any $a \in A$ and h, h' such that $(h, h') \in \Delta_B \to TC$ (and thus, $h = h'$) we have $(\mathsf{bind}_T^{B,C} \circ g)\,a\,h = \mathsf{bind}_T^{B,C}(g\,a)\,h'$. Therefore, we conclude

$$(F_{Cont_{TC}}(\mathsf{bind}_T^{B,C} \circ g), \mathsf{bind}_T^{C,C}(F_T\,g)) \in T_2^{\mathsf{rel}}(\Delta_C) \ .$$

Since $(\mathsf{val}_T^C, \mathsf{val}_T^C) \in \Delta_C \to \Delta_{TC}$, the lemma is proved. □

Proof (theorem 2). Finally, we get

$$\begin{aligned} F_T &= \Phi_T(F_{Cont_{TC}}) && \text{(by lemma 4)} \\ &= \Phi_T(\mathit{tree2fun}_{Cont_{TC}}(\mathit{fun2tree}\,F)) && \text{(by lemma 3)} \\ &= \mathit{tree2fun}_T(\mathit{fun2tree}\,F) && \text{(by lemmas 1, 4)} \end{aligned}$$

This proves the theorem. □

In the paper [8], we introduced a notion of pure functionals polymorphic in state monads. We notice that from the result of theorem 2 the following is true.

Corollary 2. *For any functional*

$$F : \forall S.(A \to State_S B) \to State_S C$$

pure in the sense of [8] there exists a truly polymorphic in $T \in Monad$ *pure implementation of* F *that does* not *make use of a state monad, i.e., there exists a monadically parametric functional* $G \in \mathrm{Func}$ *such that* $F_S = G_{State_S}$ *extensionally, for all* S.

Proof. Such an implementation is given by a corresponding strategy tree representation t_F of F. That is $G = \mathit{tree2fun}\, t_F$. □

5 Partial case

In this section, we generalize the characterisation of monadically parametric second-order functionals for the partial case in the domain-theoretic setting. In what follows, we will use the term *acceptable monadic relation* to refer to acceptable monadic relations which are strict and admissible as formulated in Definition 4.

5.1 Domain of strategy trees

We construct a cppo of "strategy trees" as a solution of a recursive domain equation $X \simeq \mathcal{F}(X)$ with a locally continuous functor $\mathcal{F} : \mathcal{C} \to \mathcal{C}$ for a suitable category $\mathcal{C}$ of domains.

Let $\eta_X : X \to X_\perp$ and $\mathit{kleisli}_X : (X \to X_\perp) \to (X_\perp \to X_\perp)$ be defined by

$$\eta_X\, x = x \qquad \mathit{kleisli}_X\, f\, x = x \ .$$

Define the lift monad $T_\perp$ over **Cpo** by

$$T_\perp X = X_\perp, \quad \mathsf{val}^X_{T_\perp} = \eta_X, \quad \mathsf{bind}^{X,Y}_{T_\perp}(t)(f) = \mathit{kleisli}_X\, f\, t \ .$$

Let $\mathcal{F}(X) = C + B \times (A \to X_\perp)$ be such a functor for the Kleisli category for $T_\perp$ over the category **Cpo** (category of cpos with continuous functions). Let *Tree* be a cpo such that $\mathit{Tree} \simeq \mathcal{F}(\mathit{Tree})$, together with two (continuous) isomorphism functions

$$\begin{aligned} \mathsf{fold} &: C + B \times (A \to \mathit{Tree}_\perp) \to \mathit{Tree}_\perp \quad \text{and,} \\ \mathsf{unfold} &: \mathit{Tree} \to (C + B \times (A \to \mathit{Tree}_\perp))_\perp, \end{aligned}$$

i.e., $\mathit{kleisli}(\mathsf{fold}) \circ \mathsf{unfold} = \eta_{\mathit{Tree}}$ and $\mathit{kleisli}(\mathsf{unfold}) \circ \mathsf{fold} = \eta_{\mathcal{F}(\mathit{Tree})}$ hold. For all isomorphisms in the Kleisli category for $T_\perp$, say, $f : X \to Y_\perp$ and $g : Y \to X_\perp$ that $\mathit{kleisli}(f) \circ g = \eta$ and $\mathit{kleisli}(g) \circ f = \eta$, f and g are total functions. Therefore, we can define total

$$\begin{aligned} \mathsf{roll} &: C + B \times (A \to \mathit{Tree}_\perp) \to \mathit{Tree} \quad \text{and} \\ \mathsf{unroll} &: \mathit{Tree} \to C + B \times (A \to \mathit{Tree}_\perp) \ . \end{aligned}$$

using their "partial" counterparts fold and unfold. Moreover, the *minimal invariance* property takes place

$$\mathit{fixp}\, \delta = \eta$$

for $\delta : (\mathit{Tree} \to \mathit{Tree}_\perp) \to (\mathit{Tree} \to \mathit{Tree}_\perp)$ defined by

$$\delta\, e = \mathsf{fold} \circ F(e) \circ \mathsf{unfold} \ .$$

For details on a Coq development of the reverse-limit construction and a formal proof of the minimal invariance, refer to [3].

It is well known that the morphism fold forms an initial F-algebra in the Kleisli category, i.e., for any other F-algebra $\varphi : F(D) \to D$ there exists the *unique* homomorphism $h : \mathit{Tree} \to D_\perp$ such that the $\varphi \circ F(h) = h \circ \mathsf{fold}$.
(Notice: Since the Kleisli category for the lift monad is classically isomorphic to the category $\mathbf{Cppo}_\perp$ (of cppos with strict continuous functions), $\mathit{Tree}_\perp$ is also a solution of the domain equation

$$X \simeq C_\perp \oplus A_\perp \otimes (B_\perp \multimap X)_\perp$$

in $\mathbf{Cppo}_\perp$, where $\oplus$ and $\otimes$ are the smash sum and the smash product respectively.)

Definition 10. *We call elements of* $\mathit{Tree}_\perp$ strategy trees. *Define continuous "constructor" functions* $\mathsf{Ans} : C \to \mathit{Tree}_\perp$ *and* $\mathsf{Que} : A \to (B \to \mathit{Tree}_\perp) \to \mathit{Tree}_\perp$ *by*

$$\mathsf{Ans} = \mathsf{fold} \circ \mathit{inl}, \quad \mathsf{Que} = \mathsf{fold} \circ \mathit{inr} \ .$$

Definition 11. *We define the function* $\mathit{fun2tree} : \mathrm{Func} \to \mathit{Tree}_\perp$ *by*

$$\mathit{fun2tree}\, F = F_{\mathit{Cont}_{\mathit{Tree}_\perp}}(\mathsf{Que})(\mathsf{Ans}) \ .$$

The definition is correct since $\mathit{Cont}_{\mathit{Tree}_\perp}$ is a strict monad. *fun2tree* is continuous and strict.

Definition 12. *Given* $T \in \mathit{Monad}$, *we construct*

$$\mathit{tree2fun}_T : \mathit{Tree}_\perp \to \mathrm{Func}_T = \mathit{fixp}\, G_T \ ,$$

where $\mathit{fixp} : \forall D.(D \to D) \to D$ *is a fixpoint operator for cppos and*

$$G_T : (\mathit{Tree}_\perp \to \mathrm{Func}_T) \to \mathit{Tree}_\perp \to \mathrm{Func}_T = \lambda f.\, \mathit{kleisli}([\phi_T, \psi_T^f] \circ \mathsf{unroll}) \,,$$
$$\phi_T : C \to \mathrm{Func}_T = \lambda c.\lambda h.\, \mathsf{val}_T\, c \,,$$
$$\psi_T^f : A \times (B \to \mathit{Tree}_\perp) \to \mathrm{Func}_T = \lambda p.\lambda h.\, \mathsf{bind}_T(h(\pi_1\, p))(\lambda b.(f \circ \pi_2\, p)\, b\, h)$$

Define $\mathit{tree2fun}\, t = \Lambda T.\, \mathit{tree2fun}_T\, t$.

$\mathit{tree2fun}_T$ is correctly defined (since Func_T is pointed) and is continuous and strict for every strict $T \in \mathit{Monad}$.

Lemma 5. *For any* $t \in \mathit{Tree}_\perp$, $\mathit{tree2fun}\, t$ *is pure.*

Proof. Fix pointed T, T' and acceptable T^{rel} for T, T'. We note that the relation $(\Delta_A \to T^{\mathsf{rel}}(\Delta_B)) \to T^{\mathsf{rel}}(\Delta_C)$ is admissible. It follows from admissibility of Δ_C and T^{rel}. Define an admissible $P \in \mathrm{Rel}(\mathrm{Func}_T, \mathrm{Func}_{T'})$ by

$$P(f, f') \equiv \forall t.(f\, t, f'\, t) \in (\Delta_A \to T^{\mathsf{rel}}(\Delta_B)) \to T^{\mathsf{rel}}(\Delta_C) \ .$$

To prove $P(\mathit{fixp}\, G_T, \mathit{fixp}\, G_{T'})$ it suffices to show that

1. $P(\perp, \perp)$ holds. Indeed, it follows from strictness of T^{rel}.

2. for all g, g', $P(g, g')$ implies $P(G_T\, g, G_{T'}\, g')$. Take g, g' such that $P(g, g')$ and a strategy tree t.
   Case $t = \bot_{Tree_\bot}$. Then $(G_T\, g\, t, G_{T'}\, g'\, t) = (\bot, \bot) \in (\Delta_A \to T^{\mathsf{rel}}(\Delta_B)) \to T^{\mathsf{rel}}(\Delta_C)$.
   Case $t = \mathsf{Ans}(c)$. Then $(G_T\, g\, t, G_{T'}\, g'\, t) = (\lambda h.\, \mathsf{val}_T\ c, \lambda h.\, \mathsf{val}_T\ c) \in (\Delta_A \to T^{\mathsf{rel}}(\Delta_B)) \to T^{\mathsf{rel}}(\Delta_C)$.
   Case $t = \mathsf{Que}(a, f)$. Then

   $$(G_T\, g\, t, G_{T'}\, g'\, t) = (\lambda h.\, \mathsf{bind}_T(h\, a)(\lambda b.g(f\, b)\, h), \lambda h.\, \mathsf{bind}_T(h\, a)(\lambda b.g'(f\, b)\, h)).$$

   Take $(h, h') \in \Delta_A \to T^{\mathsf{rel}}(\Delta_B)$. Using admissibility of T^{rel}, it suffices to check $(\lambda b.g(f\, b)\, h, \lambda b.g'(f\, b)\, h)) \in \Delta_B \to T^{\mathsf{rel}}(\Delta_C)$. It follows from $P(g, g')$. □

5.2 Representation theorem

Lemma 6. *For any $t \in Tree_\bot$, $fun2tree(tree2fun\, t) = t$.*

Proof. We note that $fun2tree \circ tree2fun$ is a homomorphism for *Tree*. Thus, the statement follows from initiality of fold. We give a direct formal proof using the minimal invariance property. □

Proofs of the following results are similar to the proofs in the total case.

Theorem 3. *For any pure $F \in$ Func,*

$$tree2fun_T(fun2tree\, F) = F_T$$

holds (extensionally) for any $T \in Monad$.

We first prove that the statement holds for an arbitrary continuation monad with a pointed result domain.

Lemma 7. *Given pure F, $tree2fun_{Cont_S}(fun2tree\, F) = F_{Cont_S}$ holds for any cppo S.*

Proof. The proof is similar to the proof of lemma 3. We construct a strict, admissible and acceptable monadic relation T_1^{rel} for monads $Cont_{Tree_\bot}$ and $Cont_S$ as in lemma 3 and utilize purity of F. □

As in the total case, for $T \in Monad$ we define

$$\Phi_T : (A \to Cont_{TC} B) \to Cont_{TC} C) \to (A \to TB) \to TC\,.$$

We prove

Lemma 8. *For any pure $F \in$ Func and $T \in Monad$, $\Phi_T(F_{Cont_{TC}}) = F_T$.*

Proof. The proof repeats the one of lemma 4. We only have to check that T_2^{rel} defined as in lemma 4 is a strict, admissible and acceptable monadic relation, which does hold. □

6 Generalizations

In this section, we argue that it is possible to extend the notion of purity to an arbitrary second-order type. Consider a general type n-Func of second-order functionals with n functional arguments

$$\begin{aligned} n\text{-Func} &= \forall T.(A_1 \to TB_1) \to \cdots \to (A_n \to TB_n) \to TC \\ &\simeq \forall T.(A_1 \to TB_1) \times \cdots \times (A_n \to TB_n) \to TC \ . \end{aligned}$$

Definition 13. *A functional $F \in n$-Func is* pure (monadically parametric) *iff*

$$(F_T, F_{T'}) \in (\Delta_{A_1} \to T^{\mathsf{rel}}(\Delta_{B_1})) \to \ldots \to (\Delta_{A_n} \to T^{\mathsf{rel}}(\Delta_{B_n})) \to T^{\mathsf{rel}}(\Delta_C)$$

holds for all $T, T' \in$ Monad and acceptable monadic relations T^{rel} for T, T'.

By theorem 1, any well-typed program of type n-Func is pure in this sense.

Definition 14. *The set of strategy trees n-Tree is a minimal set generated by constructors:*

- $\mathsf{Ans} : C \to n\text{-}Tree$
- $\mathsf{Que}_i : A_i \to (B_i \to n\text{-}Tree) \to n\text{-}Tree$, $i = 1, \ldots, n$

Similar to the case of one functional argument, one defines functions

$$\begin{aligned} &\mathit{tree2fun} : n\text{-}\mathit{Tree} \to n\text{-Func} \qquad \text{and} \\ &\mathit{fun2tree} : n\text{-Func} \to n\text{-}\mathit{Tree} \ . \end{aligned}$$

Now, the result of theorem 4 can be generalized for n-Func.

Theorem 4. *Given a pure $F \in n$-Func, $\mathit{tree2fun}_T(\mathit{fun2tree}\, F) = F_T$ holds (extensionally) for any $T \in$ Monad.* □

We provide formal Coq proofs for the case $n = 2$ in the total setting.

Characterization for the type n-Func with k parameters

$$\begin{aligned} n\text{-Func}_{D_1,\ldots,D_k} = \forall T.D_1 \to \cdots \to D_k \to \\ (A_1 \to TB_1) \to \cdots \to (A_n \to TB_n) \to TC \end{aligned}$$

is similar, with parameterized strategies of type

$$n\text{-}\mathit{Tree}_{D_1,\ldots,D_k} = D_1 \to \cdots \to D_k \to n\text{-}\mathit{Tree} \ .$$

For types of order higher than two it is not that clear what corresponding strategies should be let alone how one could characterise their existence by parametricity. It could be, however, that strategies in the sense of game semantics, like in [1, 2, 9], are the right generalization. This might be an interesting question for further investigation.

```
let rec solve (n:int) x s : Maybe S =
  match n with
  | 0 → None
  | _ →
    if is_stable x s then
      Some s
    else
      let s0 = add_stable x s in
      do p ← F (eval (n−1) x) s0;
      let (d, s1) = p in
      let cur = getval s1 x in
      let new = cur ⊔ d in
      if new ⊑ cur then
        Some s1
      else
        let s2 = setval x new s1 in
        let (w, s3) = extract_work s2 in
          solve_all (n−1) w s3

and solve_all (n:int) w s : Maybe S =
  match n with
  | 0 → None
  | _ →
    match w with
    | [] → Some s
    | x :: xs →
      (solve (n−1) x s) >>=
        solve_all (n−1) xs

and eval n x y : StateT_S Maybe D =
  match n with
  | 0 → fun s → None
  | _ → fun s →
      let s0 = add_infl y x s in
      do s1 ← solve (n−1) y s0;
      Some (getval s1 y, s1)
```

Fig. 1. The pure functional implementation of totalized **RLD**

7 Applications

The provided characterization of pure functionals of type Func can be used for verification of generic off-the-shelf fixpoint algorithms, which are used to compute a (local) solution of a constraint system $\mathbf{x} \sqsupseteq F_{\mathbf{x}}$, $\mathbf{x} \in V$, defined over a bounded join-semilattice $\mathbb{D}$ of abstract values and a set of variables V.

The local solver **RLD**, which relies on *self-observation*, applies F to a special stateful function to discover variable dependencies and perform demand-driven evaluations [7]. In order to reason about the algorithm formally, we implement **RLD** in purely functional manner and model side-effects by means of the state monad. Thus, the pure right-hand side F is assumed to be of type

$$F : \forall S.V \to (V \to \mathit{State}_S \mathbb{D}) \to \mathit{State}_S \mathbb{D} \ .$$

Figure 1 gives a pure functional implementation of a totalized version of **RLD**. Every main function of the algorithm has an extra natural parameter which limits the depth of recursion. Once the limit is reached, the solver terminates with None. Since F is pure, by corollary 2, a corresponding strategy tree provides a monadically parametric implementation, which can be used as

$$F : V \to (V \to \mathit{StateT}_S\ \mathit{Maybe}\ \mathbb{D}) \to \mathit{StateT}_S\ \mathit{Maybe}\ \mathbb{D} \ ,$$

where *Maybe* is an option monad, *StateT* is a state monad transformer, and S is a state structure managed by the solver. The total version can be implemented and proven correct in Coq with the certified code extracted in ML.

The characterization of 2-Func can be applied for verification of local fixpoint algorithms for *side-effecting* constraint systems [17] used for interprocedural analysis and analysis of multithreaded code. The main idea here is that in each constraint $x \sqsupseteq F_{\mathtt{x}}$ the right-hand side $F_{\mathtt{x}}$ is a pure function representable by a strategy tree with two kind of question nodes: QueR for which values of variables are queried using a stateful function `get` and QueW which, when accessed, update current values of some variables by means of a stateful function `set`. Thus, the strategy tree specifies a sequence of *reading* and *writing* accesses to some constraint variables. The general characterization enables the design of formally verified local solvers for side-effecting constraint systems. One version of such a solver although not verified presently is implemented in the program analyzer Goblint [20].

8 Conclusion

We have provided two equivalent characterisations of pure second-order functionals in the presence of nontermination; an extensional one based on preservation of relations and an intensional one based on strategy trees. All verifications have been formalized in Coq.

Our results can be applied to the verification of algorithms that take pure second-order functionals as input. Among these are generic fixpoint algorithms and algorithms for exact real arithmetic. It is generally easier to verify the correctness of such an algorithm assuming the intensional characterisation of purity for its input. On the other hand, for a concretely given input, e.g. in the form of a program in some restricted language it will be easier to establish the extensional characterisation.

We note that a closely related characterisation albeit in a rather different guise has already been given in O'Hearn and Reynolds landmark paper [13]. Our strategy trees appear there as an intensional characterisation of first-order Algol procedures which due to the call-by-name policy are in fact second-order functionals. New aspects of the present work are in particular the monadic formulation, the generalisation of the extensional characterisation to monads other than the state monad, and last not least the complete formalisation in Coq.

A natural question, albeit of mostly academic interest, is the extension of this work to higher than second order. Given that the strategy trees resemble winning strategies in game semantics it would seem natural to attempt to find extensional characterisations of the existence of a winning strategy. Care would have to be taken so as to sidestep the undecidability of lambda definability [11], thus the extensional property would have to be undecidable even if basic types receive a finite interpretation.

The techniques developed in this paper were extended to impure higher-order functions enabling modular reasoning about monadic mixin components [10].

References

1. Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for pcf. In Masami Hagiya and John C. Mitchell, editors, *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1994.
2. Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3, 1996.
3. Nick Benton, Andrew Kennedy, and Carsten Varming. Formalizing domains, ultrametric spaces and semantics of programming languages, 2010. Submitted to Math. Struct. in Comp. Science.
4. Baudouin Le Charlier and Pascal Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.
5. Christian Fecht and Helmut Seidl. A faster solver for general systems of equations. *Sci. Comput. Program.*, 35(2):137–161, 1999.
6. Andrzej Filinski. Representing monads. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *POPL*, pages 446–457. ACM Press, 1994.
7. Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. Verifying a local generic solver in Coq. In Radhia Cousot and Matthieu Martel, editors, *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 340–355. Springer, 2010.
8. Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. What is a pure functional? In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *ICALP (2)*, volume 6199 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2010.
9. J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for pcf: I, ii, and iii. *Inf. Comput.*, 163(2):285–408, 2000.
10. Steven Keuchel and Tom Schrijvers. Modular monadic reasoning, a (co-)routine. submitted to IFL'12, July 2012.
11. Ralph Loader. The undecidability of *lambda*-definability.
12. John Longley. When is a functional program not a functional program? In *ICFP*, pages 1–7, 1999.
13. Peter W. O'Hearn and John C. Reynolds. From algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):167–223, 2000.
14. Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In *Theoretical Computer Science*, pages 361–375. Springer-Verlag, 1993.
15. John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
16. John C. Reynolds and Gordon D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Inf. Comput.*, 105(1):1–29, 1993.
17. Helmut Seidl, Varmo Vene, and Markus Müller-Olm. Global invariants for analyzing multithreaded applications. *Proc. of the Estonian Academy of Sciences: Phys., Math.*, 52(4):413–436, 2003.
18. Alex K. Simpson. Lazy functional algorithms for exact real functionals. In Lubos Brim, Jozef Gruska, and Jirí Zlatuska, editors, *Proc. MFCS, LNCS 1450*, volume 1450 of *Lecture Notes in Computer Science*, pages 456–464. Springer, 1998.
19. The Coq Development Team. *The Coq proof assistant reference manual.* TypiCal Project (formerly LogiCal), 2012. Version 8.3pl4.
20. Vesal Vojdani and Varmo Vene. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, volume 30, pages 141–155, 2009.

Iterating Skeletons
Structured Parallelism by Composition

Mischa Dieterle[1], Thomas Horstmeyer[1], Jost Berthold[2], and Rita Loogen[1]

[1] FB Mathematik und Informatik, Philipps-Universität Marburg, Germany
{dieterle,horstmey,loogen}@informatik.uni-marburg.de
[2] Dept. of Computer Science, University of Copenhagen, Denmark
berthold@diku.dk

Abstract. Skeleton-based programming is an area of increasing relevance with upcoming highly parallel hardware, since it substantially facilitates parallel programming and separates concerns. When parallel algorithms expressed by skeletons involve iterations – applying the same algorithm repeatedly to successively improving data, the repeated instantiation of a skeleton incurs a certain overhead that could be saved by reusing existing processes, threads and communication structures. This is especially important when running parallel applications in a distributed environment.
However, customising a particular skeleton ad-hoc for repeated execution turns out to be considerably complicated, and raises general questions about introducing state into a stateless parallel computation. In addition, one would strongly prefer an approach which leaves the original skeleton intact, and only uses it as a building block inside a bigger structure.
In this work, we present a general framework for skeleton iteration and discuss requirements and variations of iteration control and iteration body. Skeleton iteration is expressed by synchronising a parallel iteration body skeleton with a (likewise parallel) state-based iteration control, where both skeletons offer supportive type safety by dedicated types geared towards stream communication for the iteration.
The skeleton iteration framework is implemented in the parallel Haskell dialect Eden. We use example applications to assess performance and overhead.

1 Introduction

Parallelism degree of modern hardware is currently increasing at multiple levels, e.g. in (GP)GPUs, CPUs and, combining all the parallel potential, in computer clusters. Increasing attention is paid to the problem of effectively programming such complex platforms at a sufficient abstraction level, especially when the programmer is not a parallelism expert. Therefore, research in parallel programming has developed a range of concepts and models for *skeleton-based parallel programming*, which encourages higher abstraction and separation of concerns. Algorithmic skeletons implement the parallel behaviour for applications of an algorithm class [Col89], represented directly in form of higher-order functions in

functional languages. Using skeletons, a concrete algorithm can be parallelised simply by applying the appropriate skeleton to function parameters which define the details of the algorithm in question. The skeleton proper describes algorithmic structure and solves independent sub-tasks in parallel - incurring a certain overhead such as thread and process creation, termination detection and communication/synchronisation. Repeatedly using one and the same skeleton leads to this parallel overhead for every skeleton instantiation, instead of reusing processes, initialisation data, and communication channels across the different skeleton invocations. As the parallel behaviour is encapsulated inside a skeleton's implementation, it is generally very hard to optimise the repeated use of a skeleton without modifying the skeleton itself. On the other hand, a solution that involves rewriting parallel skeletons for every concrete sequence of applications is infavourable; we seek for a more general method to compose skeletons for iterative computations, which we call *skeleton iteration* subsequently

Skeleton iteration should not be confused with *parallel for-loops or maps*, where a sequential block is executed in parallel by multiple threads, instead of several times. We focus on already parallel computations defined by algorithmic skeletons which will be executed several times in sequence. The contributions of this paper are as follows:

- We propose a general functional iteration scheme `iter`, a meta-skeleton (combinator) that uses an iteration control function and an iteration body skeleton. Specific control and body functionality can be freely combined to express a wide range of iterative algorithmic patterns.
- We show examples of iteration control and body skeletons and explain how to define efficiently iterable versions of ordinary skeletons – involving special types to describe iterative processing for programming comfort and safety.
- For the case of parallel (skeleton) iteration, we propose specialised solutions that work iteratively on distributed data. We show how to integrate distributed data in the iteration cycle and how to systematically adapt the interface of iteration body and iteration control for the case of parallel processing over distributed data.

Iterative algorithms in parallel computing have been discussed by researchers previously, yet solutions are usually limited to special cases and application classes. To our knowledge, our work is the first to investigate iteration as a general algorithmic pattern and its parallelism aspects and specific optimisations.

We use the parallel Haskell dialect Eden [LOMP05] to implement our skeletons. The functional approach makes it easy to precisely state interfaces and to identify conceptional requirements from our implementation. The proposed skeleton iteration framework allows for targeted optimisations of iterative algorithms, with respect to minimising data transfers and controlling dependencies.

2 Iterating Skeletons

Algorithmic skeletons are higher-order functions with a hidden parallel implementation, which provide tools for parallel programming at high abstraction.

A skeleton exposes an interface relating only to the algorithmic structure, and hides technical details of parallel execution, such as process creation, communication and synchronisation. As already outlined, this encapsulation becomes an obstacle when the actual algorithm is one that involves iterative application of the same skeleton to successively improve or approximate the final result.

2.1 Basic Idea

We start with a simple introductory example to clarify the basic idea of our approach. The Haskell prelude function `iterate` defines the iteration of a parameter function `f`, producing an infinite list (or: stream) of all intermediate results of the iteration: `[x,(f x),(f (f x)),...]`.

```
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)
```

This same stream can equally well be computed using the following alternative definition which uses the `map` function and a feedback of the result stream instead of direct recursion:

```
streamIterate :: (a → a) → a → [a]
streamIterate f x = xs
  where xs = x : map f xs
```

We are especially interested in the case where the parameter `f` of `map` is a skeleton with parallel implementation. In this case, evaluation of `f` involves the creation of threads and/or processes and communication of data between these parallel entities. The original `iterate` function would in this case lead to repeatedly constructing and destroying skeleton instances and their parallel process system, in every iteration step. The same happens when using the alternative `streamIterate` function (due to the recursion within the `map` function), but this version can be optimised more easily.

As an illustrative example, consider the case where the parameter function `f` of `map` is itself a parallel `map` skeleton (`parMap`), i.e. creates one parallel process per input list element to apply the parameter function to this element. The following specialised version of `streamIterate` implements this:

```
iterateParMap0 :: (a → a) → [a] → [[a]]
iterateParMap0 g xs = xss
  where xss = xs : map (parMap g) xss
```

The type of `(parMap g)` is `[a] → [a]`, thus this iteration creates a stream of lists (of type `[[a]]`) computed in parallel from an initial input of type `[a]`. Our goal is to not create new processes in `parMap` for each iteration, but to reuse them for all iterations. The simple approach of swapping `map` and `parMap` (to use `parMap (map g) xss`) would lead to a pseudo-parallelisation over the stream instead of over the lists. Instead, the stream of lists needs to be transposed into a list of streams and vice versa, as in the following definition:

```
iterateParMap1 :: (a → a) → [a] → [[a]]
iterateParMap1 g xs = xss
  where xss = xs : transpose (parMap (map g) (transpose xss))
```

Now the iteration via `map` takes place within the processes created by `parMap` only once. It is by virtue of streaming and the use of `map` to express the iteration that we can lift the body skeleton to work on streams and push the iteration inside the processes. In the following, we will propose special types and mechanisms to generalise this approach and make a clear distinction between the iteration stream and the list of inputs to the parallel processes. We will also add special control functions for the iteration to improve locality and performance.

2.2 Deriving Iterable Body Skeletons

Implementation Language. We use the parallel Haskell dialect Eden to present our language-independent concept. In Eden, the `parMap` skeleton

```
parMap :: (Trans b, Trans c) ⇒ (b → c) → [b] → [c]
```

creates a parallel process for every element of the input list, which eagerly evaluates the application of the parameter function (mapping input of type `b` to output of type `c`). Processes are distributed among the available machines; and their inputs (the list elements) and process outputs (elements of the result list) are sent implicitly to and from these processes. Communication-related properties of Eden processes are determined by internally used overloaded communication functions in the type class `Trans` for transmissible data. Data which is transferred will generally be evaluated to normal form prior to sending it. Eden processes thereby introduce strictness into Haskell's non-strict default evaluation. Furthermore, instances for `Trans` define different send modes: elements of a certain type will by default be evaluated and sent as a single item, but they can also be decomposed into components which are sent concurrently, or sent as a stream of elements. Such streams can also be infinite, but this is not important in the context of our work. The important aspect here is that the type of a process determines the communication mode for its in- and outputs.

Running Example: K-means Clustering is a heuristic method to partition a given dataset of n d-dimensional vectors into k clusters. In an iterative approximation, the method identifies clusters such that the average distance (a metric such as the euclidian or Manhattan distance) between each vector and its nearest cluster centroid is minimal [Mac03]. The algorithm proceeds as follows:

- Randomly choose k vectors from the dataset as starting centroids.
- Assign each vector to the cluster of the nearest centroid.
- Compute the centroids of the new clusters.
- Repeat the last two steps until the clusters do not change anymore.

```
vectorss:: [[Vector]]  --distrib. in sub-lists
body :: [Centr] → [[(Int, Centr)]]
body centroids = parMap centr $ zip vectorss (repeat centroids)

centr   :: ([Vector],[Centr]) → [(Int,Centr)]
combine :: [[(Int, Centr)]] → [Centr]
```

Fig. 1: Original version of iteration body for parallel K-means

The iterated function takes a list of cluster centroids as input and computes the list of new centroids as output. The iteration needs to continue until two subsequent iteration results are equal or their differences fall below a threshold. In Figure 1, the `parMap` skeleton serves to specify a single iteration step of the K-means example which can be executed in parallel. The vectors are distributed among the processes and every process receives the whole list of centroids as input. Function `centr` computes a list of weighted sub-centroids in every process, based on its subset of the vectors. Subsequently, all sub-centroids must be combined with the weighted centroids of the other processes, done in the recursive outer function (not shown here), which uses the function `combine`. The algorithm will re-iterate with these new centroids until convergence is detected.

Special Stream Type for Iteration. As discussed before, simply iterating this iteration body will trigger all internal side effects needed for the parallel evaluation over and over again, in every iteration step.

We would like to use a single skeleton instance for the whole iteration which is modified to work with input and output *streams*. In our example above, streams were modelled as lists, leading to a potential pseudo-parallelisation. Therefore we now define a special iteration type to clearly distinguish between these iteration streams and e.g. the input lists that the parallel map transforms in one step. The iteration type `Iter`, shown in Figure 2, is isomorphic to lists but different with respect to the communication mode. The benefits of distinguishing between iteration streams and ordinary lists are the following:

- We are able to identify iteration inputs and outputs in type signatures.
- The distinction of iteration streams and normal lists at type level increases type safety, as the type checker can distinguish them.
- Streaming behaviour can be defined in the `Trans` instance especially for `Iter`, while other lists can be communicated as single items.[3]

Aside from the new data type, Figure 2 shows auxiliary functions which define patterns that can be frequently used to define efficiently iterable skeletons.

[3] The original Eden definition specifies that top-level lists are communicated as streams. In this work, we use a modified `Trans` class which gives programmers more control of streaming through separate stream types.

```
newtype Iter a = Iter {fromIter :: [a]}

instance Functor Iter where
    fmap f = Iter ∘ map f ∘ fromIter

distribWith :: (a→[b]) → Iter a → [Iter b]
distribWith f = map Iter ∘ transposeRt ∘ map f ∘ fromIter

combineWith :: ([b]→a) → [Iter b] → Iter a
combineWith f = Iter ∘ map f ∘ transpose ∘ map fromIter
```

Fig. 2: Iter type and auxiliary functions

The functor instance of Iter provides `fmap` as an elegant way of lifting a function of type `a → b` to iteration streams, `Iter a → Iter b`. It is used to realise the iteration inside the skeleton body. Function `distribWith` splits one iteration stream into many iteration streams, where each i'th element of the resulting streams is generated from the i'th element of the originating stream. The function parameter `f` is used to produce lists for each element of the iteration stream, which are distributed into the list of streams using `map Iter ∘ transposeRt`. One subtle detail here is that `f` must produce lists of identical length for all its arguments (elements of the iteration stream). The implementation of `distribWith` thus needs to use a custom function `transposeRt` for rectangular matrices. `transposeRt` is mostly equivalent to the prelude function `transpose`, but fixes the length of its result list to the length of the first inner list of its input. In this way, the number of output streams is determined by the first incoming stream element. Finally, the function `combineWith` defines the inverse transformation (and does not impose restrictions on the transposition used).

With these tools at hand, it is easy to define a more efficiently iterable version of `parMap` (see Figure 3), which transforms inputs of type `Iter [b]` element by element to outputs of type `Iter [c]`. The input to the skeleton is a stream of lists (of equal length, see above), and the elements of each list should be sent to the mapper processes in each iteration step. Effectively, this is achieved by only using the type conversion and the `transposeRt` inside `distribWith` (but setting the transformation function to `id`), to generate a list of streams from the stream

```
simpleParMapIter :: forall b c. (Trans b, Trans c)
                    ⇒ (b→c) → Iter [b] → Iter [c]
simpleParMapIter f xss = yss where
  xss' = distribWith id xss    :: [Iter b]
  yss' = parMap (fmap f) xss'  :: [Iter c]
  yss  = combineWith id yss'
```

Fig. 3: Efficiently iterable parallel map variant

```
vectorss:: [[Vector]]  --distrib. in sub-lists
bodyIter :: Iter [Centr] → Iter [[(Int, Centr)]]
bodyIter centroids =
  simpleParMapIter centr $ zip vectorss (repeat centroids)

centr    :: ([Vector],[Centr]) → [(Int,Centr)]
```

Fig. 4: K-means iterative body function

of lists. The resulting stream of lists is transformed using `parMap (fmap f)`, and the results of type `Iter c` streamed back to the caller (bound by yss' :: [Iter c]), after reordering them again using function `combineWith`.

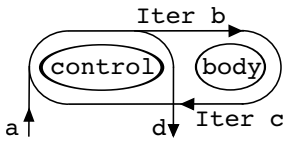
In our example, K-means, process creation overhead can now be saved, as every process of the body works on an iteration stream of input centroids – updated by the outer control function (again not shown).

This code exemplifies how to create an iteration *body* skeleton for the simple case of `parMap`. Before proceeding to more complex *body skeletons*, we discuss the *control* aspect of the iteration and the overall setup.

2.3 Iteration Scheme

Now we have to process the two "loose ends" of the iteration streams to decide termination and produce new input. The body skeleton's input stream has to be supplied with initial data, and the result stream of the skeleton must be conditionally fed back to the skeleton or terminated by closing the input stream and returning the final result. This can be defined in terms of a generic iteration scheme:

```
simpleIter :: (a → Iter c → (Iter b,d))  --control
           → (Iter b → Iter c)     --body
           → a → d                 --in/out
simpleIter control body a = d where
  (iterB,d) = control a iterC
  iterC     = body iterB
```



The meta-skeleton `simpleIter` takes as function parameters an *iteration control* function, which produces the initial input and handles the two loose ends of the iteration stream, also determining the final result, and an *iteration body* function. The latter typically is (but is not restricted to) an iterable skeleton such as `parMapIter`, which we will refer to as the *body skeleton*. All parallel side effects are encapsulated in the function parameters. Function `simpleIter` only takes care of their interconnection. The advantage is, that it defines an interface for control functions and body skeletons which can be freely combined. We allow that the body skeleton transforms input of type `Iter b` to `Iter c`. Thus, to feed the output of the body back to it, the control function elementwise transforms

a stream of type `Iter c` to `Iter b`. In general, the iteration control has the following tasks:

1. to determine the iteration body's input for further iterations,
2. to check the termination condition and
3. to produce the final output from the iteration body's output.

As examples for very simple control functions, we present the code for `loopControl`, which performs exactly n iterations by forwarding n inputs without any transformation:

```
loopControl :: Int → a → Iter a → (Iter a, a)
loopControl n a as = (Iter as', a') where
  (as',~(a':_)) = splitAt n $ a : fromIter as
```

and `untilControl`:

```
untilControl :: (c → Either a d) → a → Iter c → (Iter a, d)
untilControl cF a cs = (Iter $ a : lefts as, d) where
  (as,~((Right d):_)) = (break isRight ∘ map cF) $ fromIter cs
```

which feeds the iteration body until function parameter `cF` supplied with a bodies output yields `Right result`, the inputs for the body are the initial input and the `Left` part of the transformed bodies output `as`.

It is advantageous to use a stateful interface to the control function, which can be parameterised by a stateful transformation for single iteration steps. This ensures that the output of an iteration step is transformed to exactly one input for an iteration step, which is not enforced by the list interface. However, because of space limitations we will present our work based on the simpler list interface.

Running Example. KMeans can be efficiently iterated using the combinator `simpleIter`. In order to optimise the implementation, we use

```
simpleParMapIter' :: forall b c. (Trans b, Trans c)
                  ⇒ (a → b → c) → [a]
                  → Iter [b] → Iter [c]
```

for the iteration body, a slightly modified variant of `simpleParMapIter` presented in Figure 3, which uses additionally a static initial input for every process, and a map function working on the static *and* the iterated input. We need the additional input to keep the data vectors locally on every machine throughout the iteration, as they are dominating the communication overhead. In the definition of the iteration `body`, we partially apply the `simpleParMapIter'` skeleton to worker function `centr` and the static input `vectorss`. The stream of centroids will be supplied by the `simpleIter` combinator. Convergence of the centroids is decided based on a comparison to the centroids of the previous iteration, which are not available in the presented `untilControl` function. We therefore use a custom `control` function which directly works on the iteration streams. Using `combine` on the elements received from the body process, it merges the

```
vectorss:: [[Vector]]  --distrib. in sub-lists
body :: Iter [[Centr]] → Iter [[(Int, Centr)]]
body = simpleParMapIter' centr vectorss

control :: [Centr] → Iter [[(Int, Centr)]]
           → (Iter [[Centr]], [Centr])
control a cs = untilCmp hardlyChanged a (fmap combine cs)
hardlyChanged c1 c2 = all (< epsilon) (zipWith distance c1 c2)

kMeans :: [Centr] → [Centr]
kMeans = simpleIter control body

centr   :: [Vector] → [Centr] → [(Int,Centr)]
combine :: [[(Int, Centr)]] → [Centr]
untilCmp :: (a → a → Bool) → a → Iter a → (Iter [a], a)
```

Fig. 5: K-means: Optimised parallel version

partial centroids calculated by the processes, and then compares the new centroids to the ones obtained in the previous iteration (function `untilCmp`, using `hardlyChanged`). Function `simpleIter` combines control and body, which together implement the KMeans clustering in parallel.

2.4 Performance Tweaking

The main potential for optimisation of iteration steps lies in the reduction of communication overhead. One obvious bottleneck is the gathering and redistribution of data between the control function and the body skeleton. One approach to optimise communication is to *keep all data distributed* between the iterations. In Eden, this can be done using Remote Data [DHL10]. We can create a remote data handle from local data and fetch the data remotely using functions:

```
release :: Trans a ⇒ a → RD a
fetch   :: Trans a ⇒ RD a → a
```

The intermediate data handle of type `RD a` can be passed across processes with minor communication cost. The combination of `release` and `fetch` establishes a direct channel connection between the involved processes. When the body-skeleton's inputs and outputs are lifted to Remote Data, the actual data will be passed directly from the output of a process to its input of the following iteration step. We can define a variant of the `simpleParMapIter` skeleton simply by lifting its parameter function to the Remote Data interface:

```
parMapIterRD :: (Trans b, Trans c)
                ⇒ (b → c) → Iter [RD b] → Iter [RD c]
parMapIterRD f = simpleParMapIter (release ∘ f ∘ fetch)
```

We can now use `loopControl n` as control function to iterate `parMapIterRD` n times on input which is already supplied as Remote Data, without gathering and redistributing the data in between the iteration steps. In every iteration step, each process input will be fetched to process it locally using function `f` and afterwards released in order that it can be fetched on the same machine in the next iteration step. In the following we consider *parallel control skeletons*, which remove the conceptual bottleneck of a single manager process.

2.5 Parallel Iteration Control Skeletons

In many cases where the iteration body uses a skeleton to work on distributed data, a corresponding *control skeleton* with parallel processes can be used to inspect the distributed data, exchanging only the parts of it that are needed globally. In addition, corresponding processes of control and body skeleton can be placed on the same machine (i.e. core of a multicore, or node of a compute cluster) to avoid communication.[4]

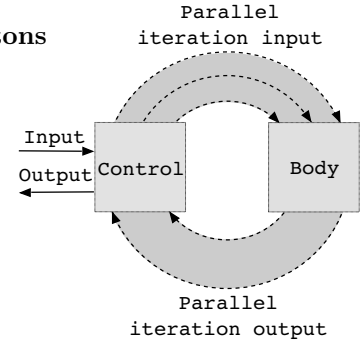


Fig. 6: Iteration scheme

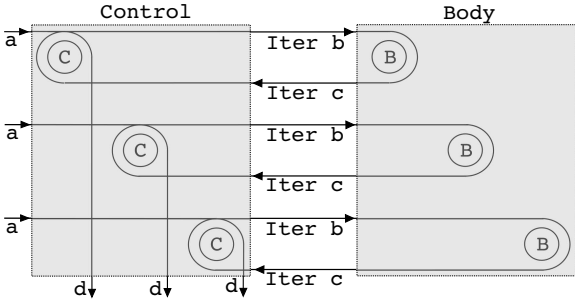


Fig. 7: Iteration body and local iteration control

Two different types of parallel iteration control can be distinguished: *local* and *global* iteration control, with respect to the data dependencies in each one of the control processes.

Local Iteration Control means that tasks of iteration control can be fulfilled without exchanging data with other control processes – data dependency is *local*, as depicted in Figure 7. Otherwise, a *global* data exchange is necessary.

[4] The parallel Haskell dialect Eden supports explicit placement of computations in a multi-node parallel system. We have omitted placement aspects from our code for simplicity throughout.

```
localControl ::
  forall a b c d. (Trans a, Trans b, Trans c, Trans d)
  ⇒ (a → Iter c → (Iter b, d))  -- ^process-local control
  → [RD a]                      -- ^initial Input
  → Iter [RD c]                 -- ^output of loops
  → (Iter [RD b], [RD d])       -- ^input for loops, final result
```

Fig. 8: Process-local iteration control skeleton

The type of a local iteration control skeleton for lists of Remote Data is given in Figure 8. The implementation is similar to the implementation of `parMapIterRD`, but takes the two input values and the tuple output into account. The control processes will connect both to their predecessor processes that produced the distributed list beforehand and to the processes of the body skeleton, fetching required data on-demand, or else passing on the `RD` handles. Functionality in each process is described by the process-local control function which transforms the initial input and the output of a process in the iteration body (stream-wise) in the respective control process. This skeleton can implement several common iteration control variants simply by partially applying the control skeleton to a suitable control function. E.g. a variant of `untilControl` where termination can be decided from local data would be:

```
localUntilCtrl :: (c → Either a d) →
                  [RD a] → Iter [RD c] → (Iter [RD a],[RD d])
localUntilCtrl checkNext = localControl (untilControl checkNext)
```

The control function `checkNext` works on the local part of a distributed list (of type `[RD a]`), and either produces input for the next iteration or the final output (again a distributed list).

Global Iteration Control If the control function needs information from multiple processes to calculate the next input for the body or to determine termination, the processes of the control skeleton need to exchange these data. As an example of this kind of control skeleton, consider an *all-gather* pattern where all processes gather data of all other processes in a distributed manner.

```
allGatherControl ::
  (a → Iter c → Iter sync)                          --t1
  → (a → Iter c → Iter [sync] → ((Iter b),d))       --t2
  → [RD a]→ Iter [RD c] → (Iter [RD b],[RD d])      --controlType
```

Fig. 9: Global control: the `allGatherControl` Skeleton

We only discuss the signature of the skeleton here, given in Figure 9. Aside from the iteration body output (distributed list of type `[RD c]`, iterated), the input for the next iteration and the final result (distributed lists `[RD b]` and `[RD d]`) depend on additional synchronisation data (of type `sync`, iterated). Function `t2` produces the local next input and result, but considers the entire list of synchronisation data (iterated). Function `t1` yields the local synchronisation data which will be communicated to all other control processes.

2.6 Inlining the Iteration Streams

In the previous two sections, we focussed on input/output data of type `[RD a/b]`, which is something like a distributed list type. For the iterated body and the control skeletons, we used the interface `Iter [RD a/b]` because we defined in the `simpleIter` scheme that data of type `a` will be passed during the iteration using type `Iter a`. There are two drawbacks of the implementations based on this signature:

1. The channel connections between the processes of the body and the control skeleton have to be rebuilt in every iteration step.
2. In the skeleton definitions, we have to drag the iteration stream from the outside of the iterated list to its elements.

In the definition of `simpleParMapIter`, we used function `distributeWith` and reversed its effects by `combineWith` when defining the skeleton's result. Similar transformations are necessary for other body skeletons as well as control skeletons. What is actually desired is a Remote Data connection list that itself carries iterated data, leading to type `[RD (Iter a)]`. If we had this type, a stream of data would be communicated over Remote Data connections established only once. The following `parMap` variant with modified interface implements these static Remote Data connections:

```
parMapIter :: (Trans b, Trans c)
              ⇒ (b → c) → [RD (Iter b)] → [RD (Iter c)]
parMapIter f = parMap (release ∘ fmap f ∘ fetch)
```

Notice that we can express the iterable skeleton simply by transforming the function parameter. We observed that the transformation of more complex topology skeletons, such as `allToAllRD` and `allReduceRD` (both developed in the context of remote data [DHL10]), are similarly easy, only involving the respective function parameters (all transformations done by the nodes are function parameters to these skeletons).

The `allToAllRD` skeleton can be adapted for Iteration streams essentially by lifting its parameter functions appropriately. Function `t1` generates the inputs for the all-to-all connection, `t2` combines the outputs from the all-to-all connections, and both are lifted using `distribWith` and `combineWith`:

```
allToAllIter :: (Trans b, Trans c, Trans i) ⇒
  (Int→b→[i]) → ([i]→c) → [RD (Iter b)] → [RD (Iter c)]
```

```
allToAllIter t1 t2 = allToAllRD t1Iter t2Iter where
    t1Iter p = distribWith $ t1 p
    t2Iter   = combineWith t2
```

Lifting skeleton `allReduceRD` to `allReduceIter` (which uses a butterfly scheme for a more efficient reduction than the former `allToAllRD`), is similarly easy.

```
allReduceIter :: (Trans b, Trans c) ⇒
  (b → c) → (c → c → c) → [RD (Iter b)] → [RD (Iter c)]
allReduceIter t r = allReduceRD (fmap t) (liftIter2 r)

liftIter2 :: (a → b → c) → Iter a → Iter b → Iter c
liftIter2 f (Iter bs1) (Iter bs2) = Iter $ zipWith f bs1 bs2
```

It uses two function parameters, function `t` transforms the initial input of each process to the reduction type. The reduce function `r` is then applied $log(n)$ times in all the nodes of the butterfly scheme. We lift `t` using `fmap`, and `r` using `liftIter2`. The latter is implemented similarly to `fmap` but uses `zipWith` instead of `map` because `r` takes two parameters.

The iteration streams to and from all body processes have to be processed by a control function or skeleton which exactly matches the particular distributed data shape. This constraint can be fulfilled by restricting the previous iteration meta-skeleton to a special type signature (`iterD`, with an implementation identical to the earlier `simpleIter`):

```
iterD :: (a → [RD (Iter c)] → ([RD (Iter b)],d))
         → ([RD (Iter b)] → [RD (Iter c)])
         → a → d
```

Further to using `iterD`, we need to define specialised versions of local and global iteration control for this interface, which is again a simplification of the existing implementations.

2.7 Unifying the Interface

The specialisation of the signature of `iterD` of the last section is not compatible to the `simpleIter` function, even though their implementations are identical. It is easy to specify a more general type for the iteration combinator,

```
type generalIter = (a → iterC → iterB,d))
                   → (iterB → iterC)
                   → a → d
```

but we lose type safety when dropping the type of the `Iter` streams. But this problem can be addressed using a type family which describes iteration types used to interconnect iteration control and iteration body skeleton. We want to have special instances for distributed data types. As an example we define a special for type distributed finite lists.

```
type family Iterated a :: *

newtype DList a = DList [RD a] --Distributed List
type instance Iterated (DList  a) = DList (Iter a)
```

The distributed list type `DList a` is defined, containing a list of Remote Data which represent the distributed elements of type `a`. Exchanging the iteration stream and the distribution by `[RD _]` is now done automatically in the type instance for `DList` of the `Iterated` type family, which yields `DList (Iter a)` – isomorphic to type `[RD (Iter a)]`. Other distributed data types and `Iterated` instances can be defined in the same way, e.g. distributed trees or distributed matrices.

We use the simple type mapping `type instance Iterated a = Iter a` to define the types of iterations for ordinary types. It is not possible to allow overlapping instances for type families, so we have to define these instances for every base-type separately. Quite advisedly, we have defined `DList a` as `newtype`, so an instance for lists can be defined without overlapping `Iterated DList a`:

```
type instance Iterated [a] = Iter [a]
```

The type family enables us to define a generic but type-safe iteration skeleton `iter` (see Figure 10) that works for both `DLists` and for any other reasonable type instance of `Iterated`. The small caveat is that two dummy parameters `b` and `c` are introduced in the control function, in order for the typechecker to check the types `Iterated b` and `Iterated c`. This is needed because the type family mapping might not be injective.

3 Evaluation

To assess and compare performance of the iteration framework and its variants, we carried out runtime measurements on a 32 node Beowulf cluster at the Heriot-Watt University Edinburgh with 8-core@2.00 GHz Intel Xeon E5504 processors. Several Eden runtime systems were co-located on nodes to make use of the multicore processors, in total up to 128, which we further refer to as processors. This is feasible because a single runtime system does not use more than one

```
iter :: (b → c          -- b/c to typecheck Iterated b/c
          → a → Iterated c → (Iterated b,d)) --control
        → (Iterated b → Iterated c)          --body
        → a → d                              --in/out
iter iterControl iterBody a = d where
  (iterB,d) = control undefined undefined a iterC
  iterC     = body iterB
```

Fig. 10: General iteration skeleton

core at once. All program versions where tested on 1, 2, 4, 8, 16, 32, 64 and 128 processors, all numbers given here are mean values of 5 program runs, in diagrams with logarithmically scaled axes.

3.1 K-means

For our measurements of the k-means algorithm, we used 600000 vectors and 25 clusters which took 142 iteration steps to terminate. The compared versions are

- *recursive parMap*
  This is the naïve setting defined in Figure 1
- *gather-distribute/simpleParMapIter*
  It uses the `simpleParMapIter'` as body, and a central control function which gathers all outputs and then distributes them for the next iteration as defined in Figure 5.
- *allGather/parMapIter*
  This uses the `parMapIter'`-body, a slight variant of `parMapIter'` using an additional static input for the data vectors, and skeleton `allGatherControl`, which lets every machine gather its input for the next iteration directly from all the others as described in Section 2.5.
- *monolithic iterUntil*
  The specialised monolithic iteration skeleton `iterUntil` described in [PR01].

Figure 11 shows the runtimes plotted against the number of processors. Our composed *gather-distribute/simpleParMapIter* version performs almost identically to the specialised monolithic `iterUntil`. Both scale well up to 128 processors, but it is notable that the sequential overhead begins to dominate in the last steps. This is mainly because of the sequential effort to evaluate and distribute the initial data.

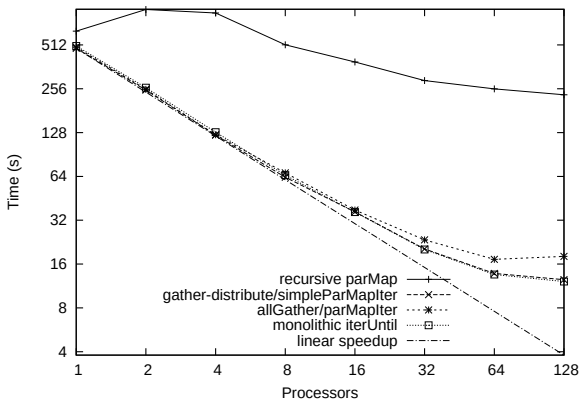


Fig. 11: Runtimes for k-means with 600000 Vectors, 25 Clusters, 142 iterations

The version *allGather/parMapIter* runs slightly slower. The amount of data that has to be communicated (the centroids) is too small to compensate for the overhead of establishing the additional communication channels in the all-to-all topology. By far the worst performance can be observed in the naïve version *recursive parMap*. Executed on a single processor where all communication is machine-local (i.e. cheap), performance is still comparable to other variants, but as soon as communication with other machines is involved, the overhead of distributing the vectors on every iteration slows down the computation enormously.

3.2 N-body

Our second example is an n-body simulation program. The n-body problem is to simulate the movement of n particles in 3-dimensional space, taking into account their mutual gravitational forces.

The set of n particles is described by the particles's mass positions and velocities. Computation proceeds in discrete time steps, each time computing the following:

- Compute a new velocity for each particle, considering the gravitational forces exerted on it by each of the other particles.
- Update each particle's position using its velocity and the time step length.

In the parallel versions of this algorithm, particles are distributed to processes and each process computes the new velocity and position for its own particles. To update its particles's velocities, each process needs information about position and mass (but not velocity) of all other particles. This information needs to be communicated in-between the iterations, leading to considerable communication between the parallel processes, in contrast to the parallel k-means algorithm described earlier.

We used variants of skeleton `allToAll` as iteration body. The processes possess a subset of the particles and exchange information about them in every iteration step in a distributed manner using the all-to-all topology. We distinguish the following versions:

- *recursive allToAllRD*
  Simply n recursive calls to `allToAllRD`. The particles are passed machine-local between the processes of the different skeleton instances. The overhead lies entirely in the repeated creation of the skeleton.
- *loopControl/allToAllIter*
  A `localLoopControl` skeleton with an `allToAllIter` body.

In the first setting, we ran the n-body simulation with 15000 bodies and 10 iterations. Workload as well as data volume to be moved in every iteration is relatively high. The runtimes against number of processes are plotted in Figure 12. While the *recursive allToAllRD* version performs better for medium numbers of processors, the *loopControl/allToAllIter* version outperforms the latter for bigger numbers of processors.

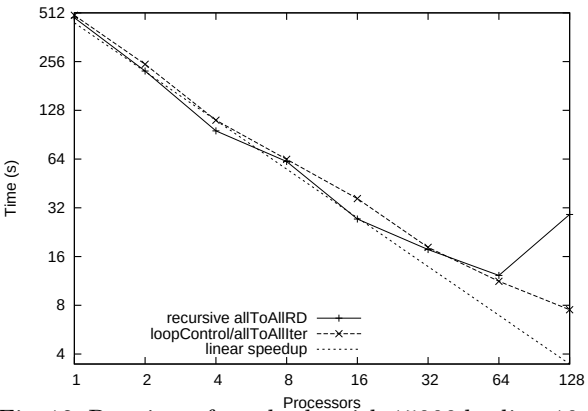


Fig. 12: Runtimes for n-body with 15000 bodies, 10 iterations

An analysis of runtime behavior revealed that the *recursive allToAllRD* has no disadvantage in the communication steps but has longer computation phases, while sharing the same code base and the same evaluation strategies, even similar communication structures (apart from the repeated creation of the skeleton in the latter version). We believe that the differences originate from the runtime system, maybe the garbage collection does not work as effectively for the former version. We plan to further investigate this. The more important observation is that the improved *localLoop/ allToAllIter* version scales much better.

In the second setting, we reduce the workload and data volume to be sent for every iteration step, in order to measure only the overhead. We use only 1500 bodies but increase the number of iteration steps to 100. This time, version *localLoop/allToAllIter* clearly performs better than the recursive version, its runtime on 128 processors is about 4 times faster than the runtime of the other version.

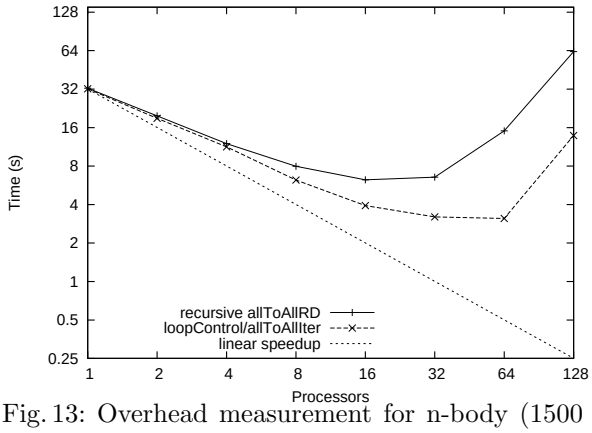


Fig. 13: Overhead measurement for n-body (1500 bodies, 100 iterations)

4 Related Work

The original skeleton work by Murray Cole [Col89] contains a chapter on an iterative completion, parallelised on a grid of processes, but does not generalise iteration as we do. Slightly more general is the iteration skeleton proposed in earlier Eden skeleton work [PR01], realising an iteration of a stateful parallel `map`. This work lays the grounds for our investigation, but does not generalise iteration bodies and types, nor does it consider parallel control skeletons.

Many skeleton libraries, especially those based on imperative programming languages, provide the constructs `while` for conditional iteration or `for` for fixed iteration, see e.g. the Scandium library [LP10], which uses Java as computation language. Scandium allows nesting of skeletons: in the `while` skeleton, another skeleton can be used to define the iteration body. No indications are made about whether the iterated body skeleton will be optimised with respect to process creation overhead, nor do the authors address or mention data transfer. The latter question is less important for performance, as Scandium targets multi-core architectures and may use update-in-place.

A slightly larger corpus of related work can be found in the cloud computing community, but usually restricted to map-reduce [DG08,BDL09] computations. iMapReduce [ZGGW12] provides an API to describe map-reduce jobs followed

by a "distance check" (Euclidian or Manhattan) and means to pass data directly from reducers to mappers (1:1 or 1:all). While elegant and addressing a key aspect, the framework appears limited in its restriction to map-reduce, and in disallowing stateful computation nodes for the sake of failover safety. A more liberal programming style is proposed by Twister [ELZ+10], whose extensions to the Map-Reduce programming model allow for general message-passing and publish-subscribe communication between all nodes. This approach, however, essentially breaks the desired functional guarantees of map-reduce. HaLoop [BHBE12] is another Map-Reduce extension, which adheres more cleanly to the original map-reduce paradigm, and maintains the functional properties. HaLoop mainly capitalises on caching mechanisms for unmodified data and reduction results across several iterations of one map-reduce computation over the same dataset. A small API extension is provided to specify how existing map-reduce (Hadoop) computations should be iterated.

None of these publications addresses parallel iteration as a general concept or distills out algorithmic patterns as we do. This generalising conceptual angle is present in very recent work in the data-flow framework Stratosphere [ETKM12]. The authors propose the concept of "incremental" iteration and "microsteps" to exploit sparseness of data dependencies and optimise read-only data accesses, but thereby break up the iterative nature of the computation.

5 Conclusions and Future Work

Iteration is one of the main building blocks of programming. In this work, we developed a general approach to describing iteration that works not only in the common sequential setting but also in the case where the iterated computation is highly parallel and executed in a distributed setting. We allow for arbitrary parallel body skeletons and supply some parameterised control functions including step counting and termination conditions on local and global data. We have shown how body skeletons can be transformed in such a way that the body processes will be re-used for all iterations, how to handle streams of input and output data, and how to optimise communication between distributed processes in a parallel execution.

The functional language setting helped us in the design phase to precisely state needed interfaces and to identify conceptual properties and requirements. Type families enabled us to non-intrusively introduce optimisations for distributed input and output types. Qualitative and quantitative examinations prove the effectiveness of these optimisations. Runtime measurements for two non-trivial example applications, k-means and n-body, clearly show that our framework similar to monolithic iteration skeletons and better than directly programmed iterations where the iterated skeletons are repeatedly instantiated.

In the future, we plan to make further efforts in the interesting field of skeleton composition. One missing piece is surely the extension of this work on iteration to other distributed data structures. Adequate type class support for such distributed data structures will be helpful. Some special issues need fur-

ther investigation. E.g. we need to find out why the `allToAllIter` skeleton performs worse than its heartbeat variant for some cases. In the context of the development of the iteration framework, we had to adjust parts of the Eden implementation, especially the `Trans` class to overload communication behaviour on the library level. Moreover, optimisations on runtime system level to shortcut messages directed to processes on the same machine turned out to be of general interest. A detailed investigation of the impact of these changes and of further improvements of the Eden system are left for future work.

References

[BDL09] Jost Berthold, Mischa Dieterle, and Rita Loogen. Implementing Parallel Google Map-Reduce in Eden. In H. Sips, D. Epema, and H.X. Lin, editors, *Euro-Par 2009*, volume 5704 of *Lecture Notes in Computer Science*, pages 990–1002. Springer-Verlag, 2009.

[BHBE12] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. The HaLoop approach to large-scale iterative data analysis. *VLDB Journal*, 21(2):169–190, 2012.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DHL10] Mischa Dieterle, Thomas Horstmeyer, and Rita Loogen. Skeleton composition using remote data. In Manuel Carro and Ricardo Peña, editors, *PADL*, volume 5937 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2010.

[ELZ+10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818, New York, 2010. ACM.

[ETKM12] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. In *Proceedings of VLDB 2012*, LNCS, 2012. To appear.

[LOMP05] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[LP10] Mario Leyton and José M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In Marco Danelutto, Julien Bourgeois, and Tom Gross, editors, *PDP*, pages 289–296. IEEE Computer Society, 2010.

[Mac03] David MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003. Chapter 20. An Example Inference Task: Clustering PDF, pages 284–292.

[PR01] R. Peña and F. Rubio. Parallel Functional Programming at Two Levels of Abstraction. In *PPDP'01 — Intl. Conf. on Principles and Practice of Declarative Programming*, pages 187–198, Firenze, Italy, September 5–7, 2001.

[ZGGW12] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. iMapReduce: A Distributed Computing Framework for Iterative Computation. *Journal of Grid Computing*, 10:47–68, 2012.

Data Layout Inference for Code Vectorisation — Extended Abstract —

Artjoms Šinkarovs and Sven-Bodo Scholz

Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, United Kingdom

In the last decade vectorisation became an important research topic again, as most of the modern CPUs grant vectorisation capabilities by means of SIMD instructions. Classical research in auto-vectorisation focuses on the optimisation of loop nestings. Data-independent operations within such loop nestings are identified and the loop-nestings as well as the order of operations within the loop nestings are reorganised to match pre-defined vectorisation patterns, typically sequences of identical arithmetic operations within loops. For vectorisation to be effective, such subsequent operations need to work on data that are adjacent in memory. Otherwise, loading/storing overheads would easily outweigh any possible performance gains from using SIMD operations. Furthermore, vectorisation only yields a substantial benefit if it can be applied within loop nestings, preferably within the innermost loops. As a consequence, classical auto-vectorisation fails to deliver substantial performance improvements whenever loop nestings cannot be re-arranged to match the layout of the data structures that are being computed on.

In this paper, we propose a radically different approach towards vectorising a given program. Rather than focusing purely on a reorganisation of loop nestings we suggest a reorganisation of data layouts to enable vectorisations. Based on an analysis of loop nestings, we infer a suitable memory layout and transform the given program to match that layout. Subsequently, we apply classical vectorisation techniques to achieve the overall goal.

The idea to modify data layouts by means of compiler transformations is not new. There has been quite some work in the context of optimisations for improved cache behaviour and, more recently, for improved streaming through GPUs. In that work, improvements of spatial and temporal locality are the key goals. While this may seem to be a goal very similar to what we propose here, spatial locality is not sufficient for an efficient vectorisation, as we will demonstrate at the example of a naive N-Body code. Furthermore, all prior work, at least to our knowledge, focuses on one individual loop nesting only rather than looking at the entire lifetime of a variable within a program.

Transforming the overall layout of data structures in memory is a quite challenging endeavour as it has far reaching implications.

First of all, semantical correctness of such a transformation is a non-trivial issue. Some languages guarantee a certain layout in memory as part of their semantics which precludes from the transformations we propose. To our knowledge, most functional languages are free from such constraints apart from some corner cases, where interfacing with the non-functional world is being supported. However, even in the functional setting semantical issues arise when it comes to

code reuse and separate compilation. If layout transformations are eligible in the context of modules either layout information needs to be stored or explicit layout conversions need to be inserted to preserve some standard layout at the interface. The latter comes with a potentially prohibitive runtime overhead and, thus, may annihilate any gains through vectorisation.

A second challenge arises from the layout inference itself. While one individual loop nesting may suggest a particular layout for all data structures involved, another loop nesting may lead to different layout suggestions for the very same data. Finding a beneficial performance trade-off between using one of the inferred layouts or inserting layout conversions constitutes a non-trivial challenge. For the same reason, reuse and separate compilation, again, become an issue. Whole program analyses are likely to be crucial for effective vectorisation.

Further challenges arise from the differences in the executing hardware. The quality of a vectorized code vastly depends on careful programing within a given hardware. Missing alignment specifications, inappropriate operation expansion, poor pipeline usage – all this can lead to serious slowdowns. SIMD instruction sets differ from architecture to architecture, and there is no uniformly agreed framework to address them. The straight-forward programming model is to express vector instructions in assembly, which is non-portable, hard to support, but very expressive. Alternatively we would like to consider a trade-off between expressiveness and portability, i.e. a programming abstraction for vector instructions, which is portable across major architectures, uses high-level language and is comparable with a hand-written assembly.

In this paper we propose a semantics preserving memory layout transformation technique for the functional array language SaC. SaC seems particularly well-suited for this attempt as (i) memory management is completely implicit in SaC, (ii) all data-structures are n-dimensional arrays which enables data layout changes in a high-level representation and (iii) data independent loop nestings are easily identifiable through the central data-parallel construct, the With-Loop.

The key idea of our approach is a bottom up layout inference that identifies ideal layouts (wrt SIMD vectorisation) for each individual loop nesting and then employs representational changes whenever necessary. We provide a solution to the separate compilation problem by a new program transformation that, in essence, generates two versions for each function, a vectorised version and a non-vectorised version. This enables code adaptations at the calling site without making representational changes inevitable. To cope with the potential variety of different SIMD hardware architectures, we use an extended version of GNU GCC, which we developed earlier. It enables the use of vector operations from C in a platform-independent way.

We use the N-Body problem as a case study throughout the paper. It nicely demonstrates the difficulties when attempting the classical approach to vectorisation and it also shows the effectiveness of our proposed approach. Finally, we present some performance measurements that show substantial speedups an different hardware, even in the presence of multi-threaded executions.

The Design of a GUMSMP: a Multilevel Parallel Haskell Implementation

Malak Aljabri[1], Phil Trinder[1], and Hans-Wolfgang Loidl[1]

Heriot-Watt University

Abstract. The most widely available high performance platforms today are multi-level clusters of multi-cores. GHC provides several parallel Haskell implementations. In particular, GHC-SMP supports shared memory, and GHC-GUM supports distributed memory. Both implementations use different but related runtime-environment (RTE) mechanisms. Good performance results can be achieved on shared memory architectures and on networks individually, but a combination of both, for networks of multi-cores, is lacking.
We present the design of the new multi-level parallel Haskell implementation, GUMSMP to better exploit such hierarchical platforms. It is designed to efficiently combine distributed memory parallelism, using a virtual shared heap over a cluster, with low-overhead shared memory parallelism on the multi-cores. Key design objectives in realising this system are even but asymmetric load balance, effective latency hiding, and mostly passive load distribution.

1 Introduction

Multi and many core architectures are the dominant general purpose hardware architectures. Moreover, the current trend in parallel architectures has shifted towards networks of multicores, in which several multicore CPUs with nodes sharing memory are connected via a network. A recent trend that has emerged in high-performance computing depends heavily on parallelism to efficiently exploit the hierarchy of these architectures. In particular, a hybrid parallel programming model is often used which combines a shared memory model to exploit parallelism within a multicore node and distributed memory model to exploit parallelism across the cluster of multicores.

Multilevel platforms are commonly programmed in multiple coordination abstractions, e.g. using MPI + OpenMP , where OpenMP (directive-based parallelism) is applied within a multicore node and MPI (message passing interface) is applied across the cluster of multicores. Thus this achieves a multilevel parallelism, combining the efficiencies, shared memory, and ease of programming of shared memory model and the scalability of distributed memory model. However, managing two abstractions is a burden for the programmer and increases the cost of porting to a new platform. In contrast, GUMSMP provides a uniform, semi-explicit high-level parallel programming model, with adaptive, automatic policies on both levels of the hierarchy. Therefore, this model relieves the programmer from the burden of explicitly controlling coordination on a multi-level hierarchy, delegating such control completely to the RTE.

Parallel functional languages are high level and well suited to exploit parallel architectures taking advantage of functional languages features, such as referential transparency, and the absence of side effects [8]. In particular, as the placement of parallelism

is not prescriptive, parallel functional languages make it relatively easy to exploit a cluster of multicores architectures using single programming model.

Glasgow Parallel Haskell (GpH) [23] is a widely-used parallel extension of Haskell, a lazy functional language. GPH is developed to facilitate parallel programming by limiting the programmer's work to a few key aspects of high-level coordination primitives supported internally by the language implementation. In GPH parallelism is expressed by two primitives added to the Haskell program, **par** and **pseq**. Evaluation strategies [16, 23] are polymorphic and higher order functions abstract over these primitives to provide a high level control of parallelism.

There are two different implementations of this semi-explicit programming model, namely GHC-SMP [18], a low overhead physical shared memory implementation integrated in GHC, and GUM [22], a virtual shared memory implementation on clusters built on top of explicit message passing.

A major difference between them lies in the work distribution model supported. While both implementations support a work-stealing approach, GUM distributes work in the form of sparks that are communicated by message passing and prefer coarse grain computations to be sent away. Whereas, spark pools are shared in GHC-SMP and therefore idle processors steal sparks from the spark pools of the busy ones.

In this paper we present the design of GUMSMP a multilevel parallel Haskell implementation which integrates the advantages of these two implementations. GUMSMP is motivated by the work distribution of the GHC-SMP as the shared memory model within a single multicore and by the work distribution of the GUM as the distributed memory model across a hierarchy of multicores.

The main benefits of this multi-level design of GUMSMP are:

- It provides a scalable model, which works on large distributed memory architectures.
- It efficiently exploits the specifics of distributed and shared memory on different levels of the hierarchy.
- It provides a single programming model, which makes programming easier and achieves performance portability.

Since the programming model is single, the implementation of the RTE is challenging as it has to make the decision on distributing the load and efficiently exploit the multilevel architecture.
The main contributions of this paper are as follows:

- We provide a detailed description of parallel Haskell languages and implementations (Section 2).
- We give a detailed description of the two GPH implementations GUM and GHC-SMP (Section 3).
- We present the design of GUMSMP, focusing on improved, hierarchy-aware scheduling and placement of light-weight threads (Section 4).
- We discuss the current status of the implementation for GUMSMP and give preliminary performance results (Section 5).

2 Related Work

There is a diversity of languages and implementations of parallel Haskell. At the language level, the diversity is based on the different abstractions supported which vary in how explicitly they control parallelism, e.g. implicit, semi-explicit, and fully explicit approaches. At the implementation level, the diversity is based on different classes of architectures with different characteristics, e.g. clusters, multicore etc. In this section we briefly outline different parallel Haskell languages and implementations.

2.1 Parallel Haskell Languages

Some important parallel Haskell languages are classified according to the abstraction level supported as follows:

- **Explicit**
  The Par Monad [17] is a new parallel Haskell programming model for pure deterministic parallel computations providing monadic control of concurrency.
  CloudHaskell [6] is a domain-specific language for developing programs for distributed memory systems. It emulates Erlang style message passing communication yet still benefits from Haskell features such as purity, types, and monads.
- **Semi-explicit**
  Eden [14] is a semi-explicit approach to functional parallel programming which extends Haskell with constructs to support parallelism. Processes are defined explicitly in Eden, but the communications are implicit, thus achieving a high level of abstraction. Eden supports distributed memory parallelism with message passing as a communication model. The programmer has some control over the load balancing as well as the granularity in order to specify expressions that have to be evaluated as parallel processes. Eden provides high level parallelism abstractions (libraries of skeletons) and therefore simplifies the task of parallelizing a program substantially.
  HdpH [15] a High-level Distributed-Memory Parallel Haskell is heavily influenced by the design and implementation of Cloud Haskell, targeting distributed memory architectures with multicore nodes. It supports high-level semi-explicit parallelism, dynamic load management, polymorphism, powerful coordination abstractions, and has the potential for fault-tolerance.

2.2 Parallel Haskell Implementations

Parallel Haskell Implementations are classified as distributed memory or shared memory implementations as follows:

- **Distributed Memory Implementation**
  GUM [22] is the distributed memory implementation for GPH which is discussed in further detail in Section 3.1 .
  Dream/EDI [14] is the distributed memory Eden implementation which extends GHC functionality by defining primitives for explicit remote task creation and

channel based communication mechanisms and supports the eager work distribution model.

CloudHaskell [6] is implemented entirely in Haskell as processes with explicit message passing and function closures serialisation. The overhead associated from using Haskell as a system language is acceptable as demonstrated by the initial performance results.

HdpH [15] is implemented entirely in concurrent Haskell. HdpH implementation is layered and moduler coded in Vanilla GHC Concurrent Haskell with independent modules for different coordination aspects, e.g. thread management, communication, scheduling etc., thus it preserves maintainability and facilitates development.

– **Shared Memory Implementation**

GHC-SMP [18] is the shared memory GpH implementations which is discussed in more detail in Section 3.2 .

Par Monad [17] provides implementation of the system-level functionality (work-stealing scheduler) as a Haskell library. The Par-Monad associated overhead is still low as indicated by performance results.

Many of the parallel Haskell language implementations depend on the sophisticated runtime systems implemented in low-level language to automatically manage parallelism, i.e. synchronization, communications, work scheduling etc. Examples include GUM, GHC-SMP, and Dream/EDI. The implementation of GUMSMP follows this approach.

Having a runtime system implemented in low level language resulted in a high application performance. However, the implementation maintenance is challenging and needs to be continuously updated.

The current trend for parallel Haskells is to use a concurrent Haskell to implement all functionality instead of modifying the GHC runtime system, thereby trading performance with maintainability and ease of development. Examples include CloudHaskell, Par Monad and HdpH.

Table 1. Parallel Haskell Comparison

	Distributed Memory				Shared Memory		
Property / Language	GUMSMP	GUM	Eden	CloudHaskell	HdpH	GHC-SMP	Par Monad
Fault Tolerance (isolated heaps)	-	-	(+)	+	+		
Polymorphic Closures	+	+	+	-	+	+	+
Pure, i.e. Non-Monadic API	+	+	+	-	-	+	-
Determinism	(+)	(+)	-	-	-	(+)	+
Implicit Task Placement	++	+	+	-	+	+	+
Automatic Load Balancing	++	+	+	-	+	+	+

Table 1 has been reproduced from [15] which compares the key features of different parallel Haskells. Shared memory implementations have limited scalability as they only

work on multicore. On the other hand, distributed memory implementations work on shared memory architectures as well as on distributed memory architectures. They can still give good performance on multicores as long as the tasks to be communicated are large and the communication rate is low [2, 3].

GUMSMP is also scalable, but it provides improvements for two aspects of the parallel implementations, namely implicit task placement and automatic load balancing. This resulted from integrating GHC-SMP which is tuned for multicore architectures and GUM which is tuned for clusters. Thus, GUMSMP is designed to provide an architecture-aware system tuned for a cluster of multicores architectures. As a result, if the computation is small it will remain in the multicore, but if it is large, it can be sent to different nodes in the clusters, thus reducing communication overheads associated with GUM.

All previously presented languages are dialects to Haskell. There are many other parallel functional languages. Most closely related to our approach is the Manticore [7] system implementing CML, a heterogeneous, statically typed, strict language. Most notably, it also takes a high-level and hierarchical view of designing large parallel applications, providing different mechanisms on different levels of the hierarchy. For large scale parallelism a more explicit approach drawn from the original CML [21] design is used, whereas for small-scale parallelism an implicit approach is used with support for data-parallelism, drawn from NESL [4] and Nepal [5], and support for task parallelism through light-weight, future-based synchronisation.

3 GPH Implementations

This section gives an overview of the design of GUM and GHC-SMP with special emphasis on thread management and load balancing.

3.1 GUM

GUM (Graph Reduction for a Unified Machine Model) [22] is the portable, message passing virtual machine for the parallel Haskell functional language. It extends the GHC (Glasgow Haskell Compiler) [10] runtime system by implementing a virtual shared memory abstraction. It is based on the parallel reduction of the graph representing the program, and the parallelism being exploited by the reduction of independent subgraphs being carried out in parallel [20].

The key concepts in GUM's design are supporting a virtual shared heap where the graph representing the program to be evaluated in parallel is stored and which is implemented on top of a distributed memory model as well as dynamically managing resources for work and data.

Several major components can be identified in the design of GUM:

1. **Initialization and Termination:** responsible for controlling start up and termination.
2. **Thread Management:** responsible for deciding when to generate a new thread and how to schedule the threads.

3. **Load balancing:** responsible for distributing the load in the parallel system so that the processing elements' idle time is minimized.
4. **Memory Management:** responsible for controlling access to remote data and in GUM, implementing a virtual shared heap.
5. **Communication:** responsible for transferring data and work between PEs.

Thread Management: The main component of GUM is the PE (CPU with Local Memory). Conceptually, one instance of the GUM RTE is used to represent one PE. The collection of communicating PEs implements the virtual machine. The thread management model used in GUM is called the "evaluate and die" thread management model, and it was originally developed for the parallel graph reduction machine(GRIP) [11], in which potential parallelism is represented as sparks (pointers to unevaluated graph structures). A spark is generated in the program code by explicit 'par' construct and maintained by the run time system in a flat distributed data structure "spark pool". Another pool is also maintained for runnable threads to be executed by the PE [12, 13].

The core of each PE's execution is the scheduling loop presented in Algorithm 1 which is executed by each PE until it receives a FINISH message. When a PE has no more work to do, it looks for local work in the runnable queue, and then it searches for a spark in its spark pool. If a spark is found, it is turned into a thread by creating TSO(thread state object)and the PE starts evaluating it. Otherwise, it searches for remote work by searching for a spark in other PE spark pools and hence an independent thread might execute the sparked expression.

Shared closures (nodes in the graph structure) can be either normal-form closures, representing data, or thunks, representing work (unevaluated data). Access to shared closures is implicitly synchronized to avoid two Haskell threads from evaluating the same thunk simultaneously.

There are three cases when acquiring the value of a spark:

1. If the value has been evaluated already (normal-form closure), the value is returned directly.
2. If it is under evaluation, the current thread will block on the structures and when the required data arrives, the blocked thread will be awakened and can continue.
3. If the expression has not been evaluated by another thread then the demanding thread will execute the computation itself. This behaviour is called "Thread Subsumption" because the potential parallel work is inlined by the parent thread. The potential parallelism represented by the spark is then fizzled. This principle of dynamically increasing the granularity of the threads by delaying the decision regarding whether a thread should be generated, is similar to the independently developed lazy task creation model [19].

Load Balancing: The load balancing model is designed specifically to achieve an efficient and effective distribution of the available sparks without generating an excessive amount of messages. Spark generation in GUM is cheap. It is simply the adding of a pointer to a thunk which is then added to the spark pool. This is essential to reduce the parallelism creation overhead, as well as to reduce the communication cost of sending sparks between PE. However, the cost of managing the thread pool is not as low as that for spark pool management. The reason for this is that additional information is needed

for a thread such as a live thread priority, which is essential if more flexible scheduling is to be achieved.

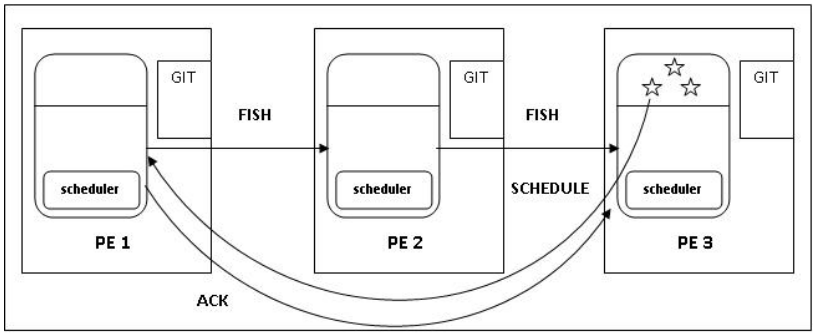


Fig. 1. Work Distribution in GUM

Figure 1 presents the work distribution in GUM which is explained as follows:

Searching for Local Work: In the current version of GUM, if there are no more threads to run in the thread pool, the scheduler searches for a spark in its spark pool. If a spark is found, it is activated by turning it into a thread and generating a TSO to hold essential information about the thread and start evaluating it. If the running thread is blocked for unevaluated values, it will be put in a queue and when the required data arrives the blocked thread will be awakened and transferred back to the runnable pool. The data becomes available when it is either reduced by a local thread in the same PE or its value is sent after being evaluated by another PE.

Searching for Remote Work: If there is no spark in the PE's spark pool, the scheduler requests work by sending a FISH message. The FISH message swims randomly from one PE to another searching for work. It includes the originating PE's id and age number representing the maximum number of PEs to visit. If the recipient PE has no spark in its spark pool, it forwards the message to another PE chosen at random after increasing its age. If the recipient has a spark, it sends it to the requesting PE as a SCHEDULE message. If no spark is found and the message limit is reached, the unsuccessful FISH is then returned to the originating PE, which then waits before sending another FISH message in order to avoid swamping the machine with FISH messages when there are only a few busy PEs. For the same reason, each PE only ever has a limited number of outstanding FISH messages (the default number is 1). This mechanism is called "work stealing", or passive work distribution, since the work is requested by the idle PE. Algorithm ScheduleFindWork presents the load balancing mechanism implemented in GUM.

```
1  void ScheduleFindWork(Capability *cap , Task *task)
2  if emptyRunQueue(cap) then
3      //Call ScheduleActivateSpark(cap)to get local work
4      if anySpark(cap) then
5          spark = tryStealSpark(cap);
6          if spark != NULL then
7              tso = createSparkThread(cap,spark);
8              pushOnRunQueue(cap,tso);
9          end
10     else
11         //Call Function ScheduleGetRemoteWork(cap)to get remote work
12         pe = choosePE();
13         sendFISH(cap,pe);
14     end
15 end
```

Function ScheduleFindWork(Capability *cap, Task *task)in GUM

3.2 GHC-SMP

is an optimized shared memory implementation for functional parallel Haskell integrated in GHC [9, 18]. It assumes a physical shared memory and uses mutexes for synchronization between local threads. GHC-SMP excels at the efficient handling of lightweight threads [1]. Millions of lightweight threads are supported by the GHC runtime system. To achieve this the threads are multiplexed onto a handful of operating system threads, approximately one for each physical CPU. A (TSO) thread state object is a heap allocated structure used to keep the Haskell thread's state together with its stack where it runs (same TSO structure as in GUM).

A set of operating system threads (worker threads, one worker thread per CPU) execute the Haskell threads. One Haskell Execution Context (HEC) is maintained for each CPU owing to the fact that the worker thread may frequently vary.

The HEC is the data structure where the data required by an OS worker thread in order to execute Haskell threads is contained. Each HEC has a spark, threads, and global black hole queues that are the same as GUM.

The state required by a HEC to perform ordinary execution of Haskell threads is local to the HEC. This means that a HEC requires no synchronisation, locks, or atomic instructions. Synchronisation is only needed for some situations such as load balancing, garbage collection etc.

Load Balancing HEC's spark pool is implemented as a bounded work-stealing queue in order to make spark distribution cheaper and more asynchronous. A work-stealing queue is a lock-free data structure where the owner can push and pop from one end of the queue without synchronization. Other threads can steal from the other end of the queue, meaning that only one atomic instruction is needed. In order to avoid a race between popping and stealing thread from the queue when it is almost empty, popping incurs an atomic instruction. On the other hand, when the queue is full, the new spark to be pushed is discarded, meaning potential parallelism may be lost.

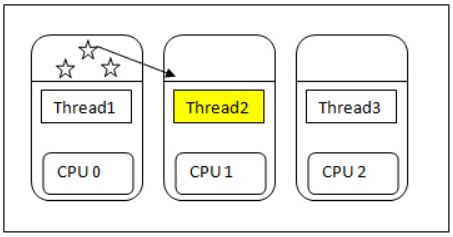


Fig. 2. Work Distribution in GHC-SMP

As shown in Figure 2, when an HEC has no assigned work, it searches for a spark, either in its HEC's spark pool or in any other HEC's spark pool. If a spark is found, then the HEC creates a 'spark thread' in order to reduce the thread overhead, which in turn steals the spark and starts evaluating it. Once this process has finished, it will steal another spark. Thus, the spark thread will evaluate sparks to WHNF sequentially until no more sparks are found, at which point it exits, allowing the TSO to be recovered by the GC.

```
1  void ScheduleFindWork(Capability *cap , Task *task)
2  if emptyRunQueue(cap) then
3      //Call ScheduleActivateSpark(cap) to get local work;
4      if anySpark(cap) then
5          for i ← 0 to num_capabilities do
6              if emptySparkPool(cap[i]) then
7                  continue;
8              end
9              spark = tryStealSpark(cap[i]);
10             if spark != NULL then
11                 break;
12             end
13         end
14         if spark != NULL then
15             tso = createSparkThread(cap,spark);
16             pushOnRunQueue(cap,tso_shell);
17         end
18     end
19 end
```

Function ScheduleFindWork(Capability *cap, Task *task)in GHC-SMP

It is necessary to create a spark thread in order to avoid creating a new thread and fresh TSO for every spark and to discard it after completing the evaluation for recovery by the garbage collector. In this way there will only be one thread executing multiple sparks. This also fixes the problem of latency between creating the parallel tasks and

being able to execute them in another CPU. Algorithm ScheduleFindWork presents the load balancing mechanism implemented in GHC-SMP .

3.3 Main Scheduling Loop

For both implementations, the core of each PE's execution is the scheduling loop presented in Algorithm 1 which is executed by each PE. The main difference between GUM and GHC-SMP is in the load balancing mechanism presented in the function **ScheduleFindWork** for both.

4 GUMSMP Design Model

GUMSMP is designed to be multilevel, integrating the work distributions of the two different parallel Haskell implementations to efficiently exploit a cluster of multicore architectures.

The main design objectives for GUMSMP can be summarized as follows:

- *Even but asymmetric load balancing:* The main objective is to balance the load between the multicores by restricting remote communications as well as controlling the placement of the new spark coming from a remote PE in response to work requesting. Whereas, within a multicore node, even load balancing is important due to the cheap communication.
- *Mostly passive load distribution:* It is essential to maintain a passive work distribution between multicore nodes, so work is only sent remotely when requested. On the other hand, within a multicore, it is preferred to maintain active work distribution as the communication is carried out locally within the same multicore.
- *Effective latency hiding:* the system must be designed in such a way that communication costs are not on the critical path of cooperative computations. This is can be achieved by providing a large pool of runnable threads, and executing another thread while another thread is waiting for the result of a communication.

In the remainder of this section we present the GUMSMP design, focussing on the work distribution algorithm and listing some of the important design alternatives.

4.1 Work Distribution Mechanism

GUMSMP integrates the two GPH implementations by combining the load balancing approaches for both. So, work distribution within the same node is achieved by GHC-SMP's work distribution mechanism, directly accessing the spark pools of other processors within the same physical shared memory machine. Across multicores, GUM's message-based work distribution is applied. This involves sending FISH messages to remote processors, in search of available sparks. This behaviour is summarised in Figure 3.

```
1  while True do
2      switch sched_state do
3          case SCHED_RUNNING
4              continue;
5          case SCHED_INTERRUPTING
6              performGC ;
7              shut_down;
8          case SCHED_SHUTTING_DOWN
9              Exit;
10
11     endsw
12     ScheduleCheckBlackHole(cap);
13     ScheduleSendPendingMessages(); //Send any messages
14     ScheduleFindWork(cap);
15     processMessages(cap);
16     ScheduleYield(cap);
17     if emptyRunQueue(cap) then
18         continue;
19     end
20     tso = popRunQueue(cap);
21     result = stgRun(tso);
22     switch result do
23         case out_of_heap
24             pushOnRunQueue(cap,tso); performGC;
25         case out_of_stack
26             enlargeStack(tso); pushOnRunQueue(cap,tso);
27         case time_expired
28             pushOnRunQueue(cap,tso);
29         case finished
30             if bound then
31                 return
32             else
33                 continue;
34             end
35
36     endsw
37 end
```

Algorithm 1: Main Scheduling Loop for GUM and GHC_SMP

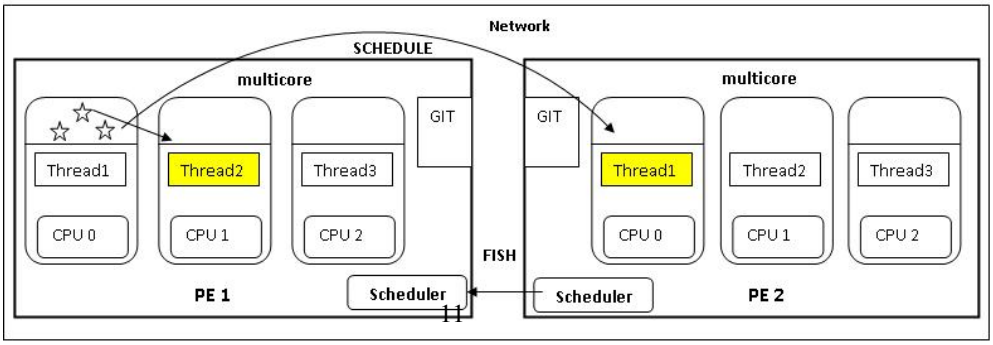


Fig. 3. Work Distribution in GUMSMP

The main novelty in the design of GUMSMP work distribution policy is its *hierarchy-aware* nature, built on efficient mechanisms that have been tuned for physical and virtual shared memory, respectively. It uses a work-stealing algorithm, through sending FISH message, on networks (inherited from GUM). Within a multicore it will search for a spark by directly accessing spark pools (inherited from GHC-SMP). If a spark found, a thread will be created to evaluate the associated piece of work. Preference is given to such local continue stealing of sparks. Only when no more local sparks are found the system will send a message to look for remote sparks. The concrete work balancing algorithm for GUMSMP is presented in Function ScheduleFindWork.

At the intersection of both levels, concrete design decisions need to be made to assure good utilisation without imposing too high communication overhead. We discuss the most important of these, namely *spark placement, fishing and work-offloading* and in the rest of this section.

```
1  void ScheduleFindWork(Capability *cap , Task *task)
2  if emptyRunQueue(cap) then
3      //Call ScheduleActivateSpark(cap)to get local work
4      if anySpark(cap) then
5          for i ← 0 to num_capabilities do
6              if emptySparkPool(cap[i]) then
7                  continue;
8              end
9              spark = tryStealSpark(cap[i]);
10             if spark != NULL then
11                 break;
12             end
13         end
14         if spark != NULL then
15             tso = createSparkThread(cap,spark);
16             pushOnRunQueue(cap,spark);
17         end
18     else
19         //Call Function ScheduleGetRemoteWork(cap)remote work ;
20         pe = choosePE();
21         sendFISH(cap,pe);
22     end
23 end
```

Function ScheduleFindWork(Capability *cap, Task *task) in GUMSMP

Spark Placement: An important decision is where to place a spark, that has been imported from another processor. An obvious choice would be to assign it to the spark pool of the first idle HEC. This would aim to keep utilisation high, but it might lead to higher fragmentation because imported sparks will be executed by different processor on a multicore. Keeping track of those processors that are executing remote sparks, and preferring these in the placement of newly imported sparks would help in reducing fragmentation, but impose additional overhead. Another design alternative would be to

use a separate spark pool, dedicated to imported sparks, from which other processors will steal work. This keeps related pieces of work together in one pool, but requires additional stealing steps in order to acquire external work. Such an additional spark pool would also be useful in situations where none of the processors are idle at the time of the arrival of a new spark (the processor originally requesting work, might have in the meantime found new work locally). Putting the imported spark into a dedicated spark pool would defer the placement decision to a later point, where idle processors are available. Committing too early, by eg. assigning the spark to the smallest spark pool, would not make best use of the dynamic information of the system.

Fishing: Another design decision is when to send a spark requesting message to a remote PE. One choice would be to send a message immediately as soon as the HEC becomes idle and there are no sparks in the local spark pools. This option might not be ideal in some cases when other HECs might produce more local sparks during the waiting time for the remote spark to arrive. An alternative option would be to delay for sometime waiting for local sparks to be generated locally. Implementing a Low-Watermark Mechanism [12] is another design alternative which shows the minimum number of sparks that should be kept local on a PE. If the number of sparks falls below this level, no sparks will be exported, and the PE will attempt to obtain new sparks from other PEs to maintain a minimum level.

Off-loading Work: How to process the received work-requesting message is another design decision. One option would be to select a spark from the HEC with the largest spark pool to send it as a response to the message. This would require traversing all HECs in order to find out the one with the largest spark pool and therefore impose additional overhead. Another alternative would be to apply the currently implemented mechanism in GUM in which the message swims randomly looking for a spark. This option is the most sensible one as it would be faster.

5 Current Implementation

The implementation of the design of GUMSMP, presented in this paper is ongoing. This section presents the current status of our implementation and focuses on initial measurements, comparing the overhead in an execution that only exploits the physical shared memory component of the combined system. The final version of the paper will present more details measurements, focussing on the performance of the combined system, with a standard benchmark suite for GPH.

This component has been tested on a common multi-core architecture, namely an eight-core machine comprising two Intel Xeon 5410 quad-core processors, running at 2.33 GHz, with a 1998 MHz front-side bus 6144 KB and 8GB RAM running under CentOS release 5.5.

Table 2 summarises our results in terms of runtimes for three test programs. Parfib which computes the Fibonacci number. Coins which computes the number of ways of paying a given value from a given set of coins. SumEuler which computes the sum of the Euler totient function on each list interval.

Table 2. Runtimes for Parfib, Coins, and SumEuler on an 8-core machine, comparing GHC-SMP and the shared memory component of GUMSMP

	Parfib		Coins		sumEuler	
No. Cores	GHC-SMP	GUMSMP	GHC-SMP	GUMSMP	GHC-SMP	GUMSMP
1	70.8	77.3	56.8	61.3	70.2	73.1
2	38.5	40.8	37.4	39.6	37.5	39.3
3	26.4	29.3	25.4	27.1	26.1	27.3
4	20.9	22.8	21.4	22.2	20.7	21.5
5	17.0	19.0	18.2	18.5	16.6	17.1
6	14.9	16.2	16.5	16.6	14.9	15.5
7	13.2	14.1	15.9	15.9	13.7	13.9
8	11.9	12.8	14.2	14.9	12.2	12.5

Preliminary results in Table 2 assess the relative performance of our merged system (GUMSMP) with the shared memory implementation (GHC-SMP). We note that on all cores of an 8-core processor, the GUMSMP version is within 7% of the GHC-SMP version. This indicates, that the overhead imposed by our implementation is low in this setup. Therefore GUMSMP is an efficient basis for larger scale distribution. The good speedups achieved for all three programs are mainly due to the efficient GHC-SMP implementation.

6 Conclusion

We have presented the design of the new multi-level parallel Haskell implementation GUMSMP, designed for high-performance computation on networks of multi-cores. Our design focuses on flexible work distribution policies. In particular, we aim for even but asymmetric load balancing, accepting that on the large scale clusters will exhibit significant differences in the relative loads on multicores, but assuring that work can be cheaply stolen between processors on one multi-core. We stress the importance of effective latency hiding, moving communication from the critical path of cooperative computations as much as possible. A high degree of available parallelism and low context switch costs are pre-requisites for this design. Finally, our system uses mostly passive load distribution, employing work stealing to obtain either local or remote work. However, we cater for computing patterns of hyper-active generators, that produce an abundance of parallelism within a short time-period, by allowing a switch to active load distribution in these cases.

Technically, GUMSMP represents a merge between two parallel Haskell implementations: GUM for distributed memory and GHC-SMP for shared memory systems. We build on the strengths of both systems to efficiently orchestrate parallel execution, and enhance the work distribution policies by accounting for the multi-level architecture by using different policies on different levels in the hierarchy. Additional benefits of our system are the high degree of scalability, covering large clusters, and the efficient support for multicore platforms. The combined system provides a single program-

ming model for such a multi-level architecture and therefore greatly simplifies parallel programming, compared to classical approach such as MPI with OpenMP. By using a high-level language model we avoid tying applications to one particular architecture, achieving higher performance portability.

The implementation of GUMSMP is still in early stages, without full support for the enhanced work distribution policies described in Section 4. We therefore focus on performance results on individual multi-cores and compare it to that of the existing GHC-SMP implementation, in order to assess the overhead of the combined system. The measurements in Section 5, on small test programs, show a very small runtime overhead. The shared memory component of the hybrid system shows performance within 7.1% of the original GHC-SMP implementation.

For the final version of this paper we will extend these measurements to assess the quality of the enhanced work distribution policy on hierarchical architectures, once completed. In particular, we are interested in test programs with aggressive generators, analysing how well our mechanisms distribute the work in this case.

Acknowledgements.

This work has been supported by the European Union grant RII3-CT-2005-026133 SCIEnce: Symbolic Computing Infrastructure in Europe, IST-2011-287510 RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software, and by the UKs Engineering and Physical Sciences Research Council grant EP/G055181/1 HPC-GAP:High Performance Computational Algebra and Discrete Mathematics and by the CALCIUM, a project in the EU Research Infrastructure project VenusC (RI-261565).

References

1. *The Computer Language Benchmarks Game*, Available at http://shootout.alioth.debian.org/u32/performance.php?test=threadring#about.
2. M. Aswad, P. Trinder, A. Al Zain, G. Michaelson, and J. Berthold, *Low Pain vs No Pain Multi-core Haskells*, Trends in Functional Programming, TFP 2009, Intellect, 2009, pp. 49–64.
3. J. Berthold, S. Marlow, K. Hammond, and A. Zain, *Comparing and optimising parallel haskell implementations for multicore machines*, Proceedings of the 2009 International Conference on Parallel Processing Workshops (Washington, DC, USA), ICPPW '09, IEEE Computer Society, 2009, pp. 386–393.
4. Guy Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha, *Implementation of a portable nested data-parallel language*, Journal of Parallel and Distributed Computing **21** (1994), 102–111.
5. Manuel M. T. Chakravarty and Gabriele Keller, *More types for nested data parallel programming*, Intl. Conference on Functional Programming, ICFP 2000, ACM Press, 2000, pp. 94–105.
6. J. Epstein, A. P. Black, and S. Peyton-Jones, *Towards Haskell in the Cloud*, Proceedings of the 4th ACM Symposium on Haskell (New York, NY, USA), Haskell '11, ACM, 2011, pp. 118–129.

7. Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao, *Manticore: a heterogeneous parallel language*, Proceedings of the 2007 workshop on Declarative aspects of multicore programming (New York, NY, USA), DAMP '07, ACM, 2007, pp. 37–44.
8. John Hughes, *Why functional programming matters*, The Computer Journal **32** (1984), 98–107.
9. Don Jones, Jr., S. Marlow, and S. Singh, *Parallel Performance Tuning for Haskell*, Proceedings of the 2nd ACM SIGPLAN symposium on Haskell (New York, NY, USA), Haskell '09, ACM, 2009, pp. 81–92.
10. S. L. Peyton Jones, C. Hall, K. Hammond, J. Cordy, H. Kevin, W. Partain, and P. Wadler, *The Glasgow Haskell Compiler: a Technical Overview*, 1992.
11. Simon L. Peyton Jones, Chris D. Clack, Jon Salkild, and Mark Hardie, *Grip - a high-performance architecture for parallel graph reduction*, FPCA, 1987, pp. 98–112.
12. H-W. Loidl, *Load Balancing in a Parallel Graph Reducer*, Trends in Functional Programming, TFP 2002, 2002, pp. 63–74.
13. H-W. Loidl, *The Virtual Shared Memory Performance of a Parallel Graph Reducer*, Intl. Symp. on Cluster Computing and the Grid, CCGrid 2002, IEEE Press, 2002, pp. 311–318.
14. R. Loogen, Y. Ortega-mallén, and R. Peña marí, *Parallel Functional Programming in Eden*, J. Funct. Program. **15** (2005), 431–475.
15. P. Maier and P. Trinder, *Implementing a High-level Distributed-Memory Parallel Haskell in Haskell*, Intl. Symposium on the Implementation of Functional Languages (IFL'11), LNCS, Springer, 2011, to appear.
16. S. Marlow, P. Maier, H-W. Loidl, M. K. Aswad, and P. Trinder, *Seq no more: Better Strategies for Parallel Haskell*, Proceedings of the 3rd ACM Symposium on Haskell (New York, NY, USA), Haskell '10, ACM, 2010, pp. 91–102.
17. S. Marlow, R. Newton, and S. Peyton Jones, *A Monad for Deterministic Parallelism*, Proceedings of the 4th ACM Symposium on Haskell (New York, NY, USA), Haskell '11, ACM, 2011, pp. 71–82.
18. S. Marlow, S. Peyton Jones, and S. Singh, *Runtime Support for Multicore Haskell*, Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (New York, NY, USA), ICFP '09, ACM, 2009, pp. 65–78.
19. E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., *Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs*, IEEE Trans. Parallel Distrib. Syst. **2** (1991), 264–280.
20. S. L. Peyton Jones, *Parallel Implementations of Functional Programming Languages*, Comput. J. **32** (1989), 175–186.
21. John Reppy, Claudio Russo, and Yingqi Xiao, *Parallel Concurrent ML*, Intl. Conference on Functional Programming (Edinburgh, UK), ICFP 2009, ACM Press, August 2009, pp. 257–268.
22. P. W. Trinder, K. Hammond, J. S. Mattson, A. S. Partridge, and S. L. Peyton Jones, *GUM: a Portable Implementation of Haskell*, Proceedings of Programming Language Design and Implementation (Philadephia, USA), PLDI 1996, may 1996.
23. P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones, *Algorithm + Strategy = Parallelism*, J. Funct. Program. **8** (1998), no. 1, 23–60.

Specification of Extensible Sparse Functional Arrays (Draft)

John T. O'Donnell

University of Glasgow
john.odonnell@glasgow.ac.uk

Abstract. A generalisation of pure functional arrays is defined, which supports extensibility and sparsity. A data parallel algorithm can implement the array operations efficiently.

1 Introduction

Imperative programming languages implement arrays use address arithmetic to locate an array element, and a store instruction to update an array element, thereby destroying the previous value of the element. Both operations — array lookup and update — take a small constant number of machine instructions.

Pure functional programming languages usually perform computations by calculating a result and allocating a new memory location to hold it, so previous values are not destroyed. This approach works well for computations on singleton data values, but it leads to inefficient arrays. If an array element is updated with a store instruction, as in imperative languages, the previous value of the array is destroyed and any existing references to it will have the wrong value.

Several techniques have been developed for overcoming this difficulty [4]. There are two broad approaches: (1) retain the generality of functional arrays but develop algorithms to improve the efficiency of array operations, and (2) implement functional arrays with destructive updates (store instructions) and provide guarantees that previous versions of the array are inaccessible to the program. Haskell supports such arrays, and this approach has been developed quite far, with efficient array accesses for parallel processors [1].

Rather than talking about "arrays in imperative/functional languages", it is clearer to use the terms *imperative array* and *functional array* for two distinct data structures. It is possible, although unusual, to implement functional arrays in an imperative language, and imperative arrays have come to be used widely in functional languages.

Since monads have made it straightforward to express algorithms using imperative arrays in a pure functional language [2], there has been less urgency to find improved implementations for functional arrays. In particular, when an imperative algorithm is translated into a functional language, imperative arrays are needed.

Nevertheless, functional arrays remain interesting in their own right, and they might have useful practical applications if they can be implemented efficiently. For example, some constraint solving algorithms use backtracking or coroutines to explore several alternative paths; functional arrays may prove useful for representing the constraint sets. Since unrestricted access to functional arrays is inefficient, there has been relatively little exploration of algorithms that rely on them, but this does not mean that such applications do not exist.

A novel implementation of a generalisation of functional arrays—called ESF arrays—appeared in 1993 [3]. It relies on fine grain data parallelism to perform both lookup and update on a functional array in a constant number of steps, without any restrictions on the past history of array operations. To achieve this performance, the algorithm requires extremely fine grain massive parallelism. The parallel hardware available at the time of publication was incapable of implementing the algorithm efficiently, but recent advances in FPGA and GPU technology have improved the situation, and the time is ripe to investigate this approach further.

The chief characteristic of the ESF array is that it is a pure functional data structure. There are two other extensions to basic arrays that are also supported: the arrays are extensible and may be sparse. Extensibility means that an array does not have fixed bounds, but may grow at any time. Sparseness means that an array that contains a large number of elements with a specific default value (such as 0) can be represented compactly, without requiring space for the 0 elements. The implementation discussed here for functional arrays also supports extensibility (this comes for free) and sparseness (which requires little extra effort).

The implementation presented here is a fine grain data parallel algorithm. The arrays are stored in a data parallel memory that is separate from the heap. The granularity is extremely fine: for best performance, there should be one processing element for each location in the array memory. The processing elements are not full scale processors; they need only the ability to perform comparisons and increments on natural numbers, and a few bit level operations. A processing element would contain on the order of 100 bits of memory and a few hundred logic gates. The algorithm is suited for implementation as a digital circuit or as an FPGA program.

The contributions of this paper are (1) a precise specification, using Haskell, of the ESF array algorithm, which was presented informally in the earlier paper [3]; (2) a precise specification correctness, in the form of a Quickcheck predicate; and (3) a discussion of the issues that arise in implementing ESF arrays on modern hardware. Practical parallel implementations, however, are left for future work.

This is a draft paper; further details will appear in the full paper.

2 Specification of ESF arrays

The ESF array data structure is defined as an abstract data type. Two versions are given: first a set of basic pure operations, which are available to the user, followed by a lower level set of operations that make the machine state visible.

2.1 User level operations

We begin by defining a basic version of the ESF array data structure. The operations defined below are suitable as an interface to the user programmer. In the following section the operations are redefined at a lower level, allowing for some essential implementation issues to be incorporated.

Arrays have type a, with index of type i and element values of type e. A basic ESF array system has a constant empty array called *empty*, and two access functions.

$$\begin{aligned} &\mathit{empty} :: a \\ &\mathit{update} :: a \to i \to e \to a \\ &\mathit{lookup} :: a \to i \to \mathit{Maybe}\ e \end{aligned}$$

The *lookup* function takes an array and an index and returns the array element defined at that index: thus $\mathit{lookup}\ a\ i$ corresponds to the imperative notation `a[i]`. The *update* function takes an array, index, and new value, and returns a new array with the given value at that index. Thus $\mathit{update}\ a\ i\ x$ corresponds roughly to the imperative notation `a[i] := x`, but there is a crucial difference: the update creates a new array without modifying the old one.

An array of undefined elements cannot be allocated all at once, as in Fortran. Instead, an array is built up incrementally through a sequence of updates, starting from the empty array:

$$\begin{aligned} &\mathit{a1} = \mathit{update}\ \mathit{empty}\ 1\ 101 \\ &\mathit{a2} = \mathit{update}\ \mathit{a1}\ 2\ 102 \\ &\mathit{a3} = \mathit{update}\ \mathit{a2}\ 3\ 103 \end{aligned}$$

The values of the arrays can be written

$$\begin{aligned} \mathit{a1} &= \{1 \rightsquigarrow 101\} \\ \mathit{a2} &= \{1 \rightsquigarrow 101, 2 \rightsquigarrow 102\} \\ \mathit{a3} &= \{1 \rightsquigarrow 101, 2 \rightsquigarrow 102, 3 \rightsquigarrow 103\} \end{aligned}$$

The chief characteristic of functional arrays is that update produces a new array, but does not change the old one. This allows for a tree-structured set of relationships among arrays. Consider the following definitions, with the previous definitions still in scope:

$$\begin{aligned} &\mathit{a4} = \mathit{update}\ \mathit{a2}\ 4\ 104 \\ &\mathit{a5} = \mathit{update}\ \mathit{a1}\ 5\ 105 \end{aligned}$$

The values of *a1* and *a2* have never changed, and the results are:

$$a4 = \{1 \rightsquigarrow 101, 2 \rightsquigarrow 102, 4 \rightsquigarrow 104\}$$
$$a5 = \{1 \rightsquigarrow 101, 5 \rightsquigarrow 105\}$$

If an update gives a new value to an index that has already been defined, the old value is shadowed: it does not appear in the new array but is still present in the old one.

$a6 = update\ a4\ 2\ 202$

$$a6 = \{1 \rightsquigarrow 101, 2 \rightsquigarrow 202, 4 \rightsquigarrow 104\}$$

The behaviour of arrays is specified by two laws that use equations to define the relationship between *empty*, *lookup* and *update*.

Law 1. (Empty array)

$lookup\ empty\ i = Nothing$

Law 2. (Nonempty array)

$$\begin{aligned} &lookup\ (update\ a\ j\ v)\ i \\ &\quad \mid i \equiv j = Just\ v \\ &\quad \mid i \not\equiv j = lookup\ a\ j \end{aligned}$$

2.2 Lower level operations on ESF arrays

In order to define the data parallel implementation, we need a lower level view of arrays and their operations. The machine state needs to be made explicit, because the implementation operates directly on the state. Furthremore, there needs to be an operation for deleting an array that has become inaccessible. Deletion would not be available to the user programmer, but would be used by the implementation to reclaim inaccessible arrays. An error mechanism is also needed; for example, if an update is performed when the array memory is full, the operation must fail.

Again, a is the type of an ESF array; i is the index type, and v is the element type. In addition, the ESF arrays are held in a separate memory with state of type s.

$$\begin{aligned} &\textbf{class}\ Eq\ i \Rightarrow ESFA\ s\ a\ i\ v \mid a \rightarrow i, a \rightarrow v\ \textbf{where} \\ &\quad empty :: a \\ &\quad update :: s \rightarrow a \rightarrow i \rightarrow v \rightarrow (s, Maybe\ a) \\ &\quad lookup :: s \rightarrow a \rightarrow i \rightarrow Maybe\ v \\ &\quad esfDelete :: s \rightarrow a \rightarrow s \end{aligned}$$

Both lookup and update can fail, giving Nothing. Lookup fails if the array a is not defined, or if a is defined but does not have a defined value at index i. Update fails if the array a is not defined, or if the memory does not have enough space to allocate a new element.

The laws for the array operations are updated to take account of the machine state and allow for failure.

3 Data parallel implementation

In an imperative array, each array element must be stored at a specific address that is calculated from the address of the array and the index of the element. Thus the position of a word in the memory determines the position of the contents of the word within an array.

The essential difficulty in implementing functional arrays is that we need to maintain full sharing of array elements—otherwise the cost in both space and time would be prohibitive—yet this means that there is no simple relationship between the location of an element in the memory and its location within all the arrays that contain it.

Consequently, we simply abandon the idea of using machine addresses to encode indices.

Instead, we develop a representation that decorates each memory location that contains an array elemeent with a representation of the set of arrays that contain the element. This is called the *inclusion set* of the element. In general, arbitrary sets cannot be represented in a small fixed amount of space. However, the inclusion sets that appear in the ESF memory are not arbitrary: they satisfy some structural properties that are forced by the fact that the memory state must be the result of a sequence of *update* operations.

Each array is identified by a unique natural number called an array code. An inclusion set can be represented by a pair of indices, low and high, such that an array is in the inclusion set if and only if its code lies between low and high. Every location in the array memory contains several fields, including an element value, and index value, an inclusion set, and a few more.

Suppose we are evaluating *lookup a i*. The algorithm begins by determining for each word in the memory whether a is a member of the inclusion set for that word. This calculation is performed in parallel in every word, and the result is used to set a mask bit in each word where the inclusion set contains a; this defines a set of words that might contain the right result; call these words the "candidates". We then compare the value of i with the index field in the candidates, and clear the mask where there is no match. It is possible that several candidates remain; this happens if an array has been calculated with several updates to the same index. The final step resolves the conflict and determines which location contains the correct value. (There are two ways to perform the resolution; the details are not given in this extended abstract.)

Notice that there are no loops in the loopup algorithm. The same is true for update and delete: each of the array operations requires a fixed number of steps.

Each step performs a lot of work—a small computation in each location—but these can all be performed in parallel.

The update and delete operations are largely similar; they all involve local computations that involve arithmetic on integers that can be performed in parallel on all the memory locations.

When an update is performed, the inclusion sets and array codes need to be adjusted. Many of the memory locations will need to modify one or more of their fields, but again these operations can all be performed in parallel. It is also necessary to modify some of the existing array codes (think of a memory allocation scheme that moves data and has to note the changed addresses of objects that have moved). Since the array codes change frequently, they are useful only inside the ESF array memory. Consequently the system maintains an association table between stable array names—which never change—and the rapidly changing codes. This association table is also maintained in the array memory. It requires two more fields in each location, and all of the operations on name/code translation are parallel.

(This brief explanation will be expanded, with examples, in the full paper.)

3.1 Representation

The representation is defined using Haskell notation. *The Haskell code that follows will not be compiled and executed to implement ESF arrays!* It is only a specification. The specification is executable, so it can be used as a reference implementation (by compiling and executing it!). It can also be used, along with correctness properties, for Quickcheck testing. To achieve a practical implementation, however, the specification can be translated to a lower level form, such as a digital circuit, an FPGA circuit, or a C+CUDA program for a GPU.

```
type StateDP1 = [CellDP1]
```

The following primitive types areused in the representation.

```
newtype EltIndex = EltIndex Int deriving (Eq, Read, Show)
newtype EltValue = EltValue Int deriving (Eq, Read, Show)
newtype AName = AName Int deriving (Eq, Read, Show)
newtype ACode = ACode Int deriving (Eq, Read, Show)
```

A cell corresponds to one memory location; it holds a complete array element.

```
data CellDP1 = CellDP1
  {low, high :: ACode,
   ind       :: EltIndex,
   val       :: EltValue,
   mapName   :: AName,
   mapCode   :: ACode}
  deriving Show
```

```
initCellDP1 :: CellDP1
initCellDP1 = CellDP1
  {low     = ACode 0,
   high    = ACode 0,
   ind     = EltIndex 0,
   val     = EltValue 0,
   mapName = AName 0,
   mapCode = ACode 0}
codeVal :: ACode → Int
codeVal (ACode x) = x
```

3.2 Data parallel algorithm

The algorithm is specified here, using Haskell. The full paper will explain how it works; some explanation is also given in the earlier paper [3].

The definitions below define precisely how the inclusion sets and array codes are handled. It does not give an accurate model of the process for allocating an empty cell to hold the result of an update; instead, this is modeled here by treating the memory as a list of cells. The full algorithm uses a parallel scan to locate an empty cell, and also for resolving index clashes.

```
encode :: StateDP1 → AName → ACode
encode [] (AName 0) = ACode 0
encode cs (AName n) =
  if n ≡ 0
    then ACode 0
    else selectUnique [mapCode c | c ← cs, mapName c ≡ AName n]


dp1update :: StateDP1 → AName → (EltIndex, EltValue) → (StateDP1, AName)
dp1update cs aname (aidx, aval) = (cs', aname')
  where
    ACode acode = encode cs aname
    acode' = acode + 1
    aname' = newName cs :: AName
    cs' = newc : map adjust cs
    adjust c =
      c {low = if codeVal (low c) > acode
               then ACode (codeVal (low c) + 1)
               else low c,
         high = if codeVal (high c) ⩾ acode
                then ACode (codeVal (high c) + 1)
                else high c,
         mapCode = if codeVal (mapCode c) > acode
                   then ACode (codeVal (mapCode c) + 1)
```

$$\textbf{else } ACode\ (codeVal\ (mapCode\ c))\}$$
$$newc = initCellDP1$$
$$\{ low = ACode\ acode', high = ACode\ acode',$$
$$ind = aidx, val = aval,$$
$$mapName = aname', mapCode = ACode\ acode'\}$$

4 Conclusion

A data structure for extensible sparse functional arrays has been specified, along with lookup and update functions for the user, and a delete function for use by the system.

The implementation relies on a fine grain data parallel host to hold the array memory. Each operation involves a small amount of calculation in every location in the entire memory. Each operation also requires a small constant number of steps, and there is no restriction on the past history of updates. That is, a lookup will always take the same time, regardless of how much sharing there is among all the existing arrays.

One way to think of the system is that it performs a lot of extra work—a little bit of arithmetic in every location—and then mitigates the extra work with massive parallelism—ideally, a processing element in every location.

However, there is a more insightful way to think of the algorithm. Consider a sequential program running on standard hardware, with a RAM memory. Programmers think of the RAM as just doing a little work on the word that is accessed (if they think of the RAM at all). However, a RAM is a digital circuit that actually has to perform an enormous amount of work on every access (not exactly a computation on every location, but that is a fair intuition). We think of the RAM as performing a small amount of work because most of its work is wasted. The ESF memory does more work than the RAM, but only by a constant factor, and it uses this work to enable it to support a data structure more efficiently than a RAM can.

The ideal host for the ESF memory would be an application specific integrated circuit (ASIC). The cost, however, would be prohibitive. FPGA chips (essentially chips that can be programmed to model an arbitrary circuit) would be almost as effective as an ASIC, at a far lower cost. Future work is planned on an FPGA implementation.

The reason that an ASIC or FPGA are well suited for ESF memory is that they can implement the algorithm at exactly the right level of granularity. As we move to coarser grain hardware, with smaller numbers of processors that are more powerful, the efficiency of the ESF algorithm diminishes. A GPU chip with several hundred processors might just about be useful for a small ESF memory; work is underway to evaluate this possibility. As we move to extremely coarse grain systems, such as traditional multicores, the amount of parallelism available is too low to overcome the extra work created by the ESF algorithm.

How useful is the data paralell ESF array? This is a data structure that allows for unlimited sharing and updates to arbitrary arrays at any time. It

may be useful for applications such as constraint solvers that perform searches through many alternative threads of possibilities, and that switch between different threads as their liklihoods of success are reevaluated. However, no such applications currently exist, and further work is needed. A final conclusion of this work is that algorithm designers do not necessarily need to shy away from functional arrays just because of their performance.

References

1. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonnell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Declarative Aspects of Multicore Programming*. ACM, January 2011.
2. S. L. Peyton Jones and P. L. Wadler. Imperative functional programming. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, 1993.
3. John O'Donnell. Data parallel implementation of Extensible Sparse Functional Arrays. In *Parallel Architectures and Languages Europe*, volume 694 of *LNCS*, pages 68–79. Springer-Verlag, 1993.
4. Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

Data Change Notifications for Cooperative Web Applications

Bob van der Linden, Steffen Michels, and Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
bobvanderlinden@gmail.com, s.michels@science.ru.nl, rinus@cs.ru.nl

Abstract. Currently, web applications are replacing traditional desktop software more and more. They have impact on the way people work by allowing people to collaborate in real-time. The way of collaboration ranges from delegating work and inspecting results of it, monitoring work of others while the work is going on, to working together on the same document in real-time. Our work on *Task-Oriented Programming (TOP)* [13], in particular the *iTask* system, aims to specifying such applications and in particular the usage of data sources on a high level of abstraction, using functional programming techniques.
One of the problems we have to solve is to keep everyone well-informed about the changes made to shared data. We propose a solution which is a novel interface combining callbacks with thread pools in a pure functional language. The technique fulfils the requirements given by the sketched application domain, but we believe it to be generally applicable to a wider range of application domains.

1 Introduction

Web applications are increasingly replacing traditional desktop software. They have several advantages, but the one that impacts the way people work most is that they allow people to collaborate in realtime. The type of collaboration ranges from delegating work and inspecting the results to working together on the same document.

Applications like Etherpad[1] and Google Wave[2] demonstrate that it is possible to achieve this kind of real-time collaboration using current web technology. These applications are however restricted to a single purpose and use a single data store exclusively. We are searching for a general way to deal with processes supporting collaborating users, which use a dynamic number of data sources with different characteristics. This is not only restricted to data storage exclusively controlled by the application server, but also includes external storages changed by other applications and even non-storage data sources like sensors.

[1] http://etherpad.org/
[2] http://wave.google.com/

This research is motivated by our work on *Task-Oriented Programming* [13], in particular the *iTask* system, which allows the processes supporting collaborating users to be defined in terms of tasks. Modelling real-world applications requires tasks to share data with other tasks and the outside world to keep users informed about the progress made by others. The description of the task completely abstracts from how and where data is stored. These kind of task descriptions create the need for a server architecture that allows all clients to be notified instantaneously about changes in an efficient way.

The problem of efficient notifications is already solved for the communication between a server and its clients by new web standards. Efficient change notification mechanisms also exist for different kinds of data storages. This paper discusses general solutions for offering an interface between arbitrary processes and data source abstractions and the consequence of different choices for the entire architecture.

Defining tasks with this high level of abstraction relies on the power of functional programming, so the solution finally discussed must fit in this framework. The proposed solution is a novel interface combining callback messages with thread pools in a purely functional language. This technique fulfils the requirements imposed by the application domain outlined above, but we believe it to be generally applicable to a wide ranger of application domains.

The remainder of this paper is organised as follows. A motivating example showing why such a general way to deal with changes is needed is given first (Section 2). A short introduction to the *iTask* system is given in Section 3. In Section 4 we then discuss the problem in more detail and define a number of requirements, serving as a basis to evaluate solutions. We then discuss the drawback of the most straightforward solution which is a blocking API in Section 5. Our solution based on a non-blocking API is described finally in Section 6. Sections 7 and 8 discuss related work and conclude the paper, respectively.

2 Motivating Example

In this section we give an example showing why a general way to deal with changes is needed. The scenario consists of an operator monitoring some machine. For this he makes use of an application server providing an application supporting this task.

It is crucial that the view of the operator is updated as the situation changes as soon as possible. We concentrate on information that might be useful, which data sources they come from and how changes of this data are reported to the application server.

A first interesting piece of information might be the operational state of the machine. Assume there is some interface between the machine and the server, the machine can then send a notification to the server when the state changes. This requires a custom implementation of the notification mechanism.

Other important information might be the current temperature of the machine measured by some sensor connected to the server. If the server can only

ask the sensor for the current temperature a proper update interval has to be defined. For instance, the temperature might be asked every 10 seconds.

Then, the operator might be interested in how long the machine is already in a certain operational state. This is a combination of the already discussed reported state, a stored timestamp and the current time. When the current time is determined, it can be predicted when it will change again. For example, the view has to be updated in one minute again for sure, at moment the view is updated and tells that the machine is already running for 24 minutes,

Finally, a maintenance log is maintained for the machine. The maintenance personnel uses the application server to log what has been done after inspecting or repairing the machine. The server can detect that the operator's view has to be updated, since the log is stored by the application server itself.

3 *iTask*

In this section we give a short overview of how a cooperative application, like the example given before, can be defined in a task-oriented way, using *iTask* [13]. The aim is not to explain the details of the system, but to describe the unique properties and architecture requiring a very general solution.

3.1 API

One wants to abstract from implementation details as much as possible, in a task-oriented programming setting, in particular *iTask*. All of the data sources discussed are handled by the same abstraction in the definition of the actual view shown to the operator. This creates the need for a uniform interface for handling all kinds of data sources.

We quickly introduce some concepts here. The goal is not to explain all details but to give an impression of the level of abstraction we want to achieve. Data sources in *iTask* are represented by an abstraction called *shared data sources* (SDS) [11]:

```
:: SDS a
```

In the actual system `SDS` has to type parameters making a difference what one can read and write, in order to achieve access control. In this way it can for instance statically be enforced that one cannot write to the current temperature. This is however not relevant for the problem discussed in this paper and we abstract from that.

The maintenance log and the measured temperature of the motivating example could for instance be represented by the following SDSs:

```
log         :: SDS [(Timestamp, String)]
temperature :: SDS Int
```

The main building block of *iTask* applications are tasks, some of them operate on those SDS. For instance, a task showing the log is represented by the following one-liner:

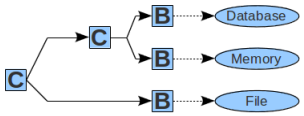


Fig. 1: Composed sources (C) consisting of basic sources (B) using different media

```
viewSharedInformation "maintenance log" [] log
```

SDSs can be combined to build new ones. One example is the following combinator:

```
(>*<) infixl 6 :: (SDS x) (SDS y) → SDS (x,y)
```

Viewing the log together with the temperature can be realised by combining two SDS:

```
viewSharedInformation "temperature & maintenance log" [] (temperature >*< log)
```

We use two kinds of powerful abstractions here. We abstract from the actual updated data store by using an SDS. The actual data could be stored in, for instance, files or some kind of other database. Additionally, we use the abstraction provided by the task having the goal of updating a data source, abstracting from how this is done. The system automatically generates a webform and ensures that all fields are filled in correctly, such that type-safety is remained, using *generic programming* [1] techniques.

Composition can be applied repeatedly to build arbitrary large composed data sources. Composition makes it possible to combine different kinds of data sources. An example of a composed source using different kinds of storages is given in Figure 1.

Composition is however not always static as with the >*< combinator. It is possible to dynamically determine the data sources involved in the composition. Details are described elsewhere [11]. The important point is that it makes thread-safe operations, like the one described in Section 6.1, more tricky.

A task such as `viewSharedInformation` has to deal with all possible kinds of data sources, but the view it provides should be updated as soon as possible when the data of these data sources changes. The mechanisms of how changes of SDSs are detected depends on the concrete SDS implementation while an SDS might even be a combination of other ones. The definition of the task should completely abstract from that.

3.2 Architecture

We start with describing the general architecture depicted by Figure 2. Each active client of the server runs a process, which is actually an instantiation of a task description. We abstract from sub-processes under the control of this main process. Computation of an *iTask* process is at the time of writing initiated only by a *client request*, which can be data filled in by the user or just a request to

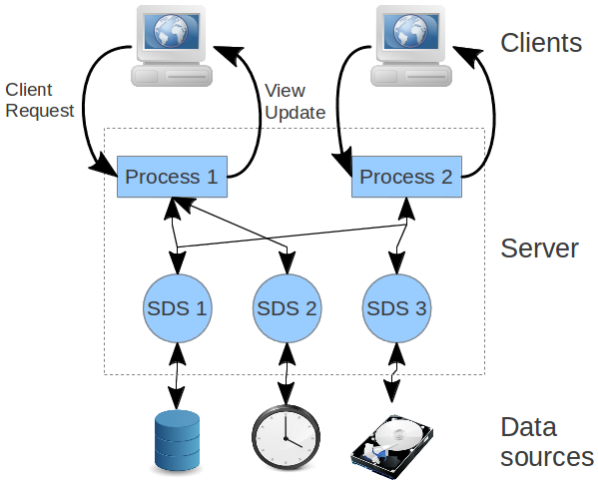


Fig. 2: General web application server architecture

refresh. After recomputation of the process, the server sends update instructions to the client. Details are described elsewhere [10].

Each process uses a number of data sources. Which sources are used can change each time the process is recomputed. They are accessed through the already discussed SDSs, finally giving access to the physical data source. We abstract from the fact that the structure of SDS can be more complex, a SDS can be composed out of other ones. Each time a process is evaluated however a number of basic SDS is read.

3.3 Implementation

The implementation of *iTask* is too complex to be discussed here, we only shortly explain the low-level interface of the SDS used by the implementation of the actual tasks, such as `viewSharedInformation`. The only operations tasks have to perform is to read and to write SDSs:

```
read  ::   (SDS a) *IWorld → (a, *IWorld)
write :: a (SDS a) *IWorld →     *IWorld
```

The *iTask* API itself is monadic, but its implementation is based on uniqueness typing, which *Clean* uses to obtain access to the outside world [2]. In this paper we assume all side-effecting functions work on a *iTask* specific uniquely attributed (`*`) type `IWorld`. We further abstracted from error handling, too.

The rest of the paper deals with extending this API, to allow tasks being recomputed when the value of the SDS has changed.

4 Problem Overview & Requirements

Our basic assumption is that the view of a user only changes when one of the SDS used by a task changes its value. Notifications that something is changed is already solved for the communication between the server and the client. New web-standards solve the need for direct communication from server to client[3]. Both of these APIs are implemented in most modern browsers. We therefore consider this problem as solved and for the rest of the paper.

Consequently, we reduce the problem to the problem of keeping the process up-to-date with the data sources. This means concretely to design an interface between the processes and the data sources abstractions, and an interface between the data source abstraction and the actual data source implementation.

We characterise different kind of data source by the way a change of their value is caused. The first kind of source behaves like data storage. It only changes if a new value is written to it. We distinguish between *internal sources*, under exclusive control of the server, and *external sources* which can be changed by other systems not under the server's control. There are then sources which change without a system changing them. Examples are the current time or sensors. The difference is that for some sources it is *predictable* when they change the next time, while some sources have to be asked whether they have changed.

We define some requirements making it possible to evaluate different possible designs. Changes should be propagated as soon as possible by using as few as possible server resources and it should allow the server process to scale. The synchronisation overhead if computing processes in parallel should remain within reasonable bounds. It should further work for all different kinds of data sources, as illustrated by the example in the previous section. This leads to some basic requirements:

Req 1 Changes of the data sources should be propagated to process depending on them as soon as possible.

Req 2 As few as possible server resources should be used.

Req 3 The synchronisation overhead when processes are running in parallel should remain within reasonable bounds.

Req 4 The solution should be very flexible. It should work for all possible change detection mechanisms.

These requirements are rather vague and one cannot define proper absolute numbers for them. However, they can be used to compare solutions. At this stage no concrete measurements have been performed, as this paper is about designing a suitable architecture avoiding obviously inefficient solutions. In the following we give a number of related requirements which should minimally hold.

We now define a number of additional requirements derived from to the general ones given before. They are more concrete and focus on the identified sub-problems.

[3] WebSocket APIWebSocket API (http://dev.w3.org/html5/websockets/), EventSource API (http://dev.w3.org/html5/eventsource/)

The actual time between the change of a data source and recomputing the process, depends on the actual server hardware and load. The server should however never miss a change. Missing a change means that a data source has changed, but the server is unaware of this fact and idles instead of undertaken actions.

Req 1.1 The server should only idle, if the effect of changed data sources is reflected in the state of all processes and vice versa.

However, it may happen that after a data source has been changed, it changes again before the change could be propagated to all processes. This is a fine, as long as the last value will be sent to the client before the server idles. The goal of web applications we deal with is to give users the most recent view on the current situation. There is no need to send all intermediate results. It is even against the goal of giving the most recent view as soon as possible to compute and communicate a situation which is known to be not up-to-date anymore.

Req 1.2 For the recomputation of processes always the most recent values available should be used.

It is always better to avoid data transfer if possible, no matter how efficient it is. There should be only minimal data transfer when nothing is changed. This requires that the transfer is initiated by the components that have to report a change. In our architecture this means that processes should not superfluously be recomputed.

Req 2.1 A process should only be recomputed if it is not up-to-date with the latest data, relevant for it.

To make **Req 4** more concrete we categorise the ways changes of data sources can be detected. The solution should work for all the mechanisms discussed. *Internal sources* do guaranteedly not change as long as no write operation is performed by the application server itself. We call this mechanism **change on write**. Data sources not exclusively controlled by the server, can notify the application about changes in several ways. This requires a custom implementation of **notification** mechanisms for each such kind of data source. For example, *Microsoft SQL Server*, *PostgreSQL* and *CouchDB* have different facilities to notify an application when changes have occurred.

How changes of **predictable** sources can be detected is obvious. Source which can only be asked whether they have changed can only be dealt with by **polling**. Poling will work for all imaginable data sources and can therefore serve as a fallback for sources not supporting a more sophisticated mechanism. A sensible polling interval together with a method to check for changes has to be defined.

5 Blocking API

The most straightforward solution to wait for changes is to wait for them using a blocking API. Waiting basically means that the execution of a thread blocks until

a data source changes. One can either wait for a single source alone (Section 5.1) or for a source in a collection of sources (Section 5.2) to change. In this section we show the limitations of this approach.

5.1 Waiting for a Single Change

The first approach is to define a function which waits for changes that happen in data sources. When we call the function, the function blocks until the data source has actually been changed. Such a function could look like this:

```
wait :: (SDS a) *IWorld → *IWorld
```

This method works well when we want to wait for a single change and continue processing when the change happened, but has problems in the context we want to use it.

First, is would require one thread per data source. Threads use quite a number of resources, since they need a stack, OS related memory and language specific support like a garbage collector. For this reason we want to restrict the number of threads (**Req 2**). The actual number giving optimal performance depends on the server's CPUs.

The most severe problem is that the semantics of the function depends on operations performed by other threads and their order. We miss a change, if a write operation to a data source is performed by another thread just before a call to `wait`.

5.2 Waiting for Multiple Changes

Waiting for multiple changes allows one thread to wait for more than one source to change. A function for this could look like this:

```
wait :: [WaitForSDS] *IWorld → *IWorld

:: WaitForSDS =∃ a: WaitForSDS (SDS a)
```

This makes it possible to use one thread per process waiting for a change of any of the data sources the process depends on. This still gives too many threads.

The other extreme is to have only a single thread waiting for any change of any of the data sources. This has several drawbacks. Such a wait function requires a list of data sources on which we want to wait. This list can become quite big. It results in quite some processing to handle the list every time a single data source changes. Also, to only recompute the process for which this is necessary (**Req 2.1**), the function has to return which sources have changed. This mechanism can become quite tricky given the fact that during the handling of a change data sources can be changed in the meantime. The problem is basically the same as with the semantics for the wait function waiting for a single SDS to change. A final problem is that the entire mechanism works in one thread only and is therefore not scalable.

From this we conclude that waiting functions are not suited to solve the problem in a way fulfilling our requirements. We need a solution allowing for a variable number of threads.

6 Non-Blocking API

To allow for a variable number of threads a non-blocking operation seems more suited. An option is to have an operation on an SDS allowing to register which process depends on it. During computation each process registers itself to the SDS. To avoid unnecessary computation (**Req 2.1**) a process should to able to unregister itself again. To make the mechanism generally applicable, for instance for handling network request as well, we use *callback messages* instead of registering something specific like process identifiers.

Callback messages can be of any user defined type. In contrast to *callback function* commonly used in other languages, message have a more functional-style solution. They can contain function, but one hast to be explicit about on which state they work and where they are applied. Additionally, serialising messages not containing functions and transferring them between threads can be more efficient than doing the same for arbitrary functions in a functional language. In the case of *iTask* for instance it is more efficient to use the identifiers of processes which have to be recomputed as messages than serialising and transferring a function on `*IWorld` recomputing a process.

So we can actually use an SDS to register *work* to be done when the corresponding data source changes. This raises the problem of by whom and in which context the work is performed. To solve this problem we will introduce a queue which contains user-defined callback messages. When we register for changes of a data source we store such a message. When a change happens, we put all callback messages of that particular data source into the queue. This queue can be shared by a variable number of threads, which retrieve the callback messages, remove them from the queue and execute them. The queue makes it possible to distribute the work among threads. As soon as a change occurs a new message is put on the queue and executed before the server may idle (**Req 1.1**). The behaviour is visualized in Figure 3.

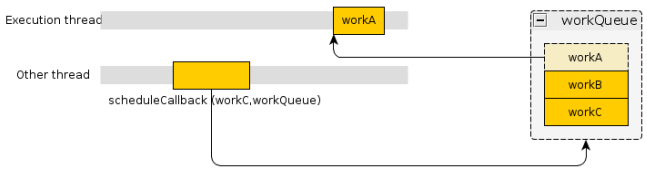


Fig. 3: Work queue for distribution callback functions to execution thread

There are three perspectives of different components involved. The task implementation has to indicate on which SDSs it depends such that the current process is re-evaluated when necessary (Section 6.1). The implementation of a data source has to provide the needed operations to support one of the discussed ways of detecting changes (Section 6.2). The application environment finally has

to provide the infrastructure for keeping tracks of the dependencies and queuing processes which have to be recomputed (Section 6.3).

6.1 The Perspective of a Task Implementation

All high-order function, like the discussed task `viewSharedInformation`, are implemented using basic functions for reading and writing SDSs. If a process wants to register a message the following extended read operation is used:

```
readRegister :: msg (SDS a) *IWorld → (a, *IWorld)
```

The operation reads the current value from a data source (`SDS a`) using the environment `*IWorld`. It is crucial that the read operation is performed together with the registration of the message. It makes sure that the message is evaluated as soon as the value is changed and does not equal the returned read value any more. In this way we achieve a clear semantics of the function.

The alternative to seperate reading and registering for changes with different API calls has the drawback that its semantics is unclear, similar to that of the discussed blocking functions. Further registering for changes without reading is of little use, since are virtually no cases in which one wants to wait for a value to change without knowing the value.

The function `readRegister` is invoked on all basic sources a SDS is composed of from left to right. Each basic implementation has to make sure that the callback message is registered in such a way that no changed value is missed. This is dicussed later.

6.2 The Perspective of a Basic Data Source Implementation

The implementation of an SDS can choose how to pass the message to the state. This flexiblity is needed in order to support the different ways of deteting changes, as discussed. The function for registering a message becomes:

```
registerSDSMsg :: (CallbackMsg msg) *IWorld → *IWorld

:: CallbackMsg msg  = EvalOnNotify   BasicShareId msg
                    | EvalAtTime     Timestamp    msg
                    | CheckForChange Timestamp BasicShareId (*IWorld → (CheckRes, *IWorld))

:: CheckRes = Changed | CheckAgain Timestamp
```

The type `CallbackMsg` represents possible ways to register messages in order to enable dealing with the different ways of detecting changes, previously discussed. The first constructor `EvalOnNotify BasicShareId msg` represents a message evaluated when a SDS notifies about a change. This can happen when a write operation is performed (**Change on write**), but the implementation of a data source can implement a custom mechanism for this as well (**Notification**). To be able to identify a SDS each one – at least if it reports changes – has a unique identifier. The write operation or some custom event listener of the SDS then has to inform the state about a change later. For this the following function can be used:

```
reportSDSChange :: BasicShareId *IWorld → *IWorld
```

`CheckForChange` can be used to schedule a check whether the value of an SDS with given identifier has been changed (**Polling**). For this a function on the state is used. Data sources which certainly change at a predictable moment of time (**Predicatble change**) can register a message to be evaluated with `EvalAtTime Timestamp msg`. There is no need to first check if the value has been changed since it certainly did, therefore the process can be recomputed immediatelly.

This scheme introduces a problem for write operations performed after read operations. A process would be informed about their own changes, but they are aware of the latest value they have written. To prevent write operations causing certain messages to be evaluated, a variable using an additional filter function is available:

```
reportSDSChangeFilter :: BasicShareId (msg → Bool) *IWorld → *IWorld

writeFilterMsg  :: a (msg → Bool) (SDS a) *IWorld → *IWorld
```

iTask processes would have to filter out messages being their own identifier.

The basic data source implementation performs basically two operations: reading the current value and registering for changes. In a concurrent environment, in the case `EvalOnNotify` is used, this can lead to missed changed, if the value changes between reading the value and registering for changes. To prevent this the basic implementation has to lock the source for write operations during registration for changes.

6.3 The Perspective of the Server State

The environment `*IWorld` consequently has to support the discussed function:

```
registerSDSMsg        :: (CallbackMsg msg)          *IWorld → *IWorld
reportSDSChange       :: BasicShareId               *IWorld → *IWorld
reportSDSChangeFilter :: BasicShareId (msg → Bool) *IWorld → *IWorld
```

It has to provide a mapping from `BasicSharedId`s to a set of messages for all messages registered using `EvalOnNotify`. When `reportSDSChange` or `reportSDSChangeFilter` is called this mapping is used to determine which messages to evaluate. No message should however be evalauted twice, since this would be a waste of resources, so possible references from other SDS to the same message should be removed when a message is evaluated. In a concurrent environment this mapping has to be thread safe as well.

For messages triggered by a timestamp it also has to support a queue for elements to which a timestamp is attached.

7 Related Work

In this paper we touched a number of topics: noticing changes in a reliable way, notifying user-code about changes and reacting to the changes efficiently. We only focused on a general solution for the particular domain of web applications

supporting users to collaborate. Many contributions have been made about notifications for changing data, but they are either dealing with a domain with different requirements or give solutions for particular implementations.

There are different database systems that support change notifications in different ways [12, 6, 9, 3]. We showed in this paper how we defined an abstraction for the change notifications so that change notifications can be implemented across different data-systems, like databases or files. This is different from the mentioned works. These explain how to define change notifications specifically for one database system. These contributions can help to apply our concept of change notifications to the different databases, since one still has to implement the specific functionality to make change notifications work for a specific type of shared data source.

The solution for change notifications that we have shown in this paper is similar to a messaging-system called publish-subscribe. There are many contributions about this technique [4, 7, 8]. It is used often in distributed systems where multiple systems want to know when new information is published. These contributions describe network protocols to distribute messages in a networked publish-subscribe system. This paper describes how to handle change notifications, that can be described as messages, on the side of the consumer. The contributions could be used for implementing such messaging-systems over the network. Using our abstraction these systems could be used for change notifications the same way as any other change notification, independent of its source.

Apart from these network protocols, there is also work on how to handle publish-subscribe mechanisms in general. In [5] filtering method is described. This method can be used to filter messages in publish-subscribe systems efficiently. In this paper we showed how to subscribe for changes of any data in data sources. When however we are only interested in parts of the data, like a section of a text-file, it would be more efficient to only be notified of changes we are interested in. In such case efficient filtering of notifications is needed. This work on filtering messages can be helpful to give insight into such problems.

8 Conclusions

We discussed several solutions to the problem of notifying clients of changes in the context of collaborative web applications which are defined in a task-oriented way using functional programming techniques. The description of tasks makes it possible to use different kinds of data sources, but to abstract from their implementation. Consequently, the notification mechanism should allow to abstract from the actual implementation of notifications as well. Furthermore, changes should be propagated as soon as possible and with using as few as possible server and network resources.

We specified the requirements in more detail and used them to evaluate and compare different possible solutions. Finally, we proposed a solution which is a novel interface combining callbacks messages with thread pools in a pure functional language. This technique fulfils the requirements given by the sketched

application domain, but we believe it to be generally applicable to a wider range of application domains.

References

1. Artem Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, Radboud University Nijmegen, 2005. ISBN 3-540-67658-9.
2. Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In Rudrapatna Shyamasundar, editor, *Proceedings of the Conference on the Foundations of Software Technology and Theoretical Computer Science, FSTTCS '93, Bombay, India*, volume 761 of *LNCS*, pages 41–51. Springer-Verlag, 1993.
3. H.T. Chou and W. Kim. Versions and change notification in an object-oriented database system. In *Proceedings of the 25th ACM/IEEE design automation conference*, pages 275–281. IEEE Computer Society Press, 1988.
4. A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. *Advances in Database Technology-EDBT 2006*, pages 627–644, 2006.
5. F. Fabret, H.A. Jacobsen, F. Llirbat, J. Pereira, K.A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD Record*, volume 30, pages 115–126. ACM, 2001.
6. PostgreSQL Global Development Group. PostgreSQL: Documentation: Manuals: NOTIFY. http://www.postgresql.org/docs/8.1/static/sql-notify.html.
7. A. Gupta, O.D. Sahin, D. Agrawal, and A.E. Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273. Springer-Verlag New York, Inc., 2004.
8. Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. *Wireless Networks*, 10(6):643–652, 2004.
9. Jan Lehnardt J. Chris Anderson and Noah Slater. Continuous Changes. http://guide.couchdb.org/draft/notifications.html.
10. Bas Lijnse and Rinus Plasmeijer. iTasks 2: iTasks for End-users. In Marco Morazán and Sven-Bodo Scholz, editors, *Revised Selected Papers of the International Symposium on the Implementation and Application of Functional Languages, IFL '09, South Orange, NJ, USA*, volume 6041 of *LNCS*, pages 36–54. Springer-Verlag, 2010.
11. Steffen Michels and Rinus Plasmeijer. Uniform data sources in a functional language. Submitted for presentation at Symposium on Trends in Functional Programming, TFP '12, 2012.
12. Microsoft. Using Query Notifications. http://msdn.microsoft.com/en-us/library/ms175110.aspx.
13. Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. Paper accepted for publication in the proceedings of the International Conference on Principles and Practice of Declarative Programming, PPDP '12, 2012.

Building JavaScript Applications with Haskell

Atze Dijkstra, Jurriën Stutterheim, Alessandro Vermeulen, and Doaitse Swierstra

Department of Information and Computing Sciences
22 Universiteit Utrecht
23 P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{atze, j.stutterheim, a.vermeulen, doaitse}@uu.nl

Abstract. We introduce the Utrecht Haskell Compiler (UHC) JavaScript backend; a compiler backend which allows one to cross-compile Haskell to JavaScript, so it can be run in the browser. To interface with JavaScript and overcome part of the impedance mismatch between the two languages, we introduce the Foreign Expression Language; a small subset of JavaScript for use in Foreign Function Interface (FFI) imports. Finally we discuss the implementation of a JavaScript application, completely in Haskell, with which we show that it is now possible to develop JavaScript applications completely in Haskell.

1 Introduction

When developing interactive web applications, JavaScript is often the language of choice, due to every major browser supporting it natively. In contrast to other client-side programming languages, no plugins are needed to execute JavaScript. Unfortunately, JavaScript is currently the *only* client-side programming language that is supported on all major browsers. People wishing to use other programming languages or paradigms have to rely on using existing plugins such as Flash or Java Applets, writing custom browser plugins, or hacking the browsers themselves. None of these options are ideal, since they either require a lot of work, or force the use of strict, imperative languages. Instead of choosing between the aforementioned options, we use the Utrecht Haskell Compiler (UHC) [9, 10] to compile Haskell code to JavaScript, effectively turning JavaScript into a high-level byte-code of some sorts.

In this paper, we introduce the UHC JavaScript backend, a compiler backend that allows one to compile Haskell to JavaScript, while keeping Haskell's lazy semantics. To overcome the impedance-mismatch between Haskell and JavaScript, we have extended UHC's Foreign Function Interface (FFI) with a small JavaScript-like expression language we call the Foreign Expression Language (FEL). With these enhancements to the FFI, we claim that it is now possible to write complete JavaScript applications using only Haskell. We back this claim up by porting a web-based Prolog proof assistant from JavaScript to Haskell. While this paper focusses on Haskell, the ideas should be relatively easy to implement in

similar languages. Additionally we provide a library containing bindings to the JavaScript standard functionality and bindings to several other commonly used JavaScript libraries.

With this paper, we make the following contributions:

- We introduce the UHC JavaScript backend; a compiler backend that allows one to compile any Haskell code supported by the UHC to JavaScript and execute it in the browser, maintaining Haskell's lazy semantics.
- We introduce the Foreign Expression Language (FEL), which allows for a more natural way of interfacing with object-oriented languages via the FFI.
- We show that it is now possible to write complete JavaScript applications using only Haskell.
- We provide a basic library with bindings to common JavaScript APIs.

The rest of this paper is structured as follows: section 2 introduces the UHC JavaScript runtime system (RTS) and FFI with our addition, after which section 4 shows the implementation of a complete JavaScript application in Haskell, after which sections 5 and 6 discuss future and related work respectively. Finally, section 7 concludes.

2 Compiling Haskell to JavasScript

2.1 Runtime System

There exists an obvious mismatch between Haskell and Object-Oriented (OO) languages, which has been addressed in various ways over time (Section 6):

- Mapping the runtime machinery required for Haskell to an imperative language has to deal with the lazy evaluation strategy imposed by Haskell (rest of this section).
- Use of OO language mechanisms as available in JavaScript, in particular prototype based objects; we only mention this topic in passing.
- Use of available JavaScript libraries; we deal with this in the next section by exploiting the freedom offered by Haskell's Foreign Function Interface (FFI)

The design of any backend for a lazy functional languages needs to deal with functions, their (lazy) application to arguments, and evaluating such applications to Weak Head Normal Form (WHNF). The design should also cater for under- and over saturated function applications as well as tail recursion.

In UHC's JavaScript backend, functions and their applications are both represented straightforwardly by objects:

$$
\begin{aligned}
&\mathit{Fun.prototype} = \{ \\
&\quad \mathit{applyN} : \textbf{function}\ (\mathit{args}) \ldots \\
&\quad \mathit{needsNrArgs} : \textbf{function}\ () \ldots
\end{aligned}
$$

```
}
function Fun (fun) {...}
```

We omit implementation details and only expose the programmatic interface as used by the runtime system. The actual implementation can be found in the UHC git repository[1]. A *Fun* object wraps a JavaScript function so that it can be used as a Haskell function. The *applyN* field is only used when function applications are being evaluated (forced); only then it is necessary to know the *needsNrArgs* number of arguments which must be passed. For the time being it stays unevaluated as a *Fun* object wrapped inside an *App* or *AppLT* closure object.

Similarly, partially applied (and thus undersaturated) functions need to store already passed arguments and how many arguments are still missing. An *AppLT* (*LT* stand for *less than*) object encodes this and again we provide its programmatic interface first:

```
AppLT.prototype = {
  applyN : function (args) ...
  needsNrArgs : function () ...
}
function AppLT (fun, args) {...}
```

An *AppLT* only wraps other *AppLT* objects or *Fun* objects.

Finally, for all remaining saturation cases an *App* object is used, knowledge about the degree of saturation is delegated to the encapsulated function object, which may be another *App*, *AppLT*, or *Fun*.

```
App.prototype = {
  applyN : function (args) ...
}
function App (fun, args) {...}
```

With this interface we now can embed Haskell functions; for example the function $\lambda x \rightarrow id(id\ x)$ is, assuming an elementary JavaScript function *id* is available, by:

```
new Fun (function (x) {
  return new App (id, [new App (id, [x])]);
})
```

Evaluation is forced by a separate function *eval* which assumes the presence of an *eOrV* (evaluator Or Value) field in all Haskell runtime values, which tells us whether the JavaScript object represents a Haskell non-WHNF value which needs further evaluation or not; in the former case it will be a JavaScript function of arity 0, which can be called. A Haskell function or application object does not evaluate itself since the entailed tail recursion will cause the stack of the

underlying JavaScript engine to flow over. The separate external function *eval* doing the evaluation allows non WHNF values to be returned, thus implementing a trampoline mechanism:

```
function eval (x) {
  while (x ∧ x.eOrV) {
    if (typeof x.eOrV == 'function') {
      x = x.eOrV ();
    } else {
      x = x.eOrV;
  } }
  return x;
}
```

Even normal JavaScript values can be thrown at *eval*, provided they do not (accidentally) contain an *eOrV* field. The actual *eval* function is somewhat more involved as it provides some protection against null values and also updates the *eOrV* field for all intermediate non WHNF objects computed in the evaluation loop.

As usual the evaluation is driven by the need to pattern-match on a value, e.g. as the result of a case expression or by a built-in JavaScript primitive which is strict in the corresponding argument such as in the wrapper of the primitive multiplication function, which contains the actual multiplication (∗):

```
new Fun (function (a, b) {
  return eval (a) ∗ eval (b);
})
```

Depending on the number of arguments provided, either an undersatured closure is built, or the function is directly invoked using JavaScripts *apply*. In case too many arguments are provided, a JavaScript closure is constructed, which subsequently is evaluated in the evaluation loop of *eval*. The implementation of *AppLT* is similar to that of *Fun*. *App*'s implementation of *applyN* simply delegates to *applyN* of the function it applies to. Also omitted are the encodings of nullary applications, used for unevaluated constants (CAF, Constant Applicative Form) and indirection nodes required for mutual recursive definitions. Data types and tuples are straightforwardly mapped onto JavaScript objects with fields for the constructor tag and its fields. If available, record field names of the corresponding Haskell data type are used.

2.2 The UHC-JavaScript library

We provide a library[28], simply called the UHC-JavaScript library, to streamline the development of JavaScript applications with UHC. It contains bindings to standard ECMAScript[12], the formal standard behind JavaScript, as well as

```
exp  ::= '{}'                -- Haskell constructor to JS object
       | (arg | i) post*     -- JS expression
post ::= '.' i               -- object field
       | '[' exp ']'         -- array indexing
       | '(' args ')'        -- function call
args ::= ε | arg (, arg)*    -- possible arguments
arg  ::= '%' ('*' | int)     -- all arguments, or a specific one
       | '"' str '"'         -- literal text
i    ::= a valid JavaScript identifier
int  ::= any integer
str  ::= any string
```

Fig. 1. Import entity notation for the JS calling convention

bindings to the jQuery library[25]. The library aims to provide a bare-metal interface that is consistent with the JavaScript functions. Eventually, this library should form the basis on which more (functional) abstractions are built. We shall make use of this library in the rest of this paper.

3 JavaScript Foreign Function Interface

We have extended the FFI with the Foreign Expression Language (FEL), a small language that greatly simplifies interfacing with the JavaScript world from Haskell. The FEL allows one to number and reorder the function arguments, explicitly use them as arguments to JavaScript functions, or use them as objects. Functions in these objects can be called in the FEL by using the dot, just like in JavaScript. Other features include hardcoding of literals, accessing array indices, and a built-in mechanism for converting data-types to JavaScript objects. The new grammar for importing functions is shown in figure 1. Common FFI features, such as the *dynamic* and *wrapper*[17] imports, work as expected, allowing one to deal with higher-order JavaScript functions.

3.1 Creating, manipulating and querying objects

Being a purely functional programming language, Haskell has no notion of objects. JavaScript, however, does. Objects come in two flavours: anonymous and named objects. The former is denoted in JavaScript as { }, while the latter is created by defining a constructor function of which the name starts with an uppercase letter, like so: **function** *MyObj* () { }. Objects can then be instantiated with the **new** keyword: **new** *MyObj* (). Each function also has a prototype object. This prototype allows for defining values and functions within the object scope. New object instances will automatically have the same values and functions as the prototype.

UHC now offers support for creating, manipulating and querying objects, using several new primitive functions in the runtime-system (RTS). Instead of showing the rather uninteresting function definitions in JavaScript, the code below shows the Haskell type signatures which need to be used when importing these primitives with the FFI:

```
primMkCtor :: JSString→IO ()
primMkObj :: JSString→JSPtr c
primMkAnonObj :: IO (JSPtr c)
primGetAttr :: JSString→JSPtr c→IO a
primSetAttr :: JSString→a→JSPtr c→IO (JSPtr c)
primModAttr :: JSString→(a→b)→JSPtr c→IO (JSPtr c)
primGetProtoAttr :: JSString→JSString→a
primSetProtoAttr :: JSString→a→JSString→IO ()
primModProtoAttr :: JSString→(a→b)→JSString→IO ()
```

JSString is a type synonym for *PackedString*, which is used as the type for JavaScript strings. The *primMkCtor* function creates a new constructor function if it does not yet exist in the *window* scope. This function is usually only called from within the other functions listed above. The *primMkAnonObj* function creates an anonymous object { }, while the *primMkObj* accepts a string with the class name of the new object. If the class does not exist yet, it is created using an empty constructor.

The other functions manipulate objects and prototypes, using a mechanism inspired by lenses[20, 15, 18]; an abstraction over accessors and mutators. The first argument is always the name of the object attribute of interest in the shape of a string. In case of the *set*-functions, the second argument is the value that needs to be set. Since JavaScript is a loosely typed language, this can be any type, even when interfacing with it from the Haskell world. The *mod*-functions take as second parameter a function which modifies the attribute specified in the first argument. Modifying an attribute can cause it to be of a different type, hence the $a \rightarrow b$ type for the function. Finally, the last argument is either a reference to an object, or the name of a class in the form of a string, in case of prototypes.

These functions can be used by importing them as primitives:

```
foreign import prim "primGetAttr"
  _getAttr :: JSString→JSPtr p→IO a
```

Objects are represented in the UHC-JavaScript libraryl by a *JSPtr a* type, which has no constructors, so it can't be instantiated directly. The only way an object can be obtained is by getting it via the FFI. A *JSPtr a* requires one type argument, which specifies the type of the JavaScript object. This should again be a type without constructor. Suppose we want a pointer to a *Book* object, we could define it as follows:

```
data JSPtr a
data BookPtr
type Book = JSPtr BookPtr
```

We can now define functions on the *Book* type, giving us a type-safe way to deal with JavaScript objects. This is similar approach as is often taken in GHC's C FFI to deal with pointer types.

We offer the *Language.UHC.JS.Primitives* module in the UHC-JavaScript library, which defines primitive imports and abstracts away from *JSString*. Using these functions we can now create, manipulate and query an object:

```
main = do
  o ← mkObj "Book"
  setAttr "pages" 123 o
  modAttr "pages" (+1) o
  p ← getAttr "pages" o
  print p   -- Prints 124
```

While defining objects as shown in the previous example works fine, the process is rather verbose and tedious, especially when dealing with several object attributes. It would therefore be ideal if we could use Haskell datatypes to achieve the same results. In some ways, datatypes and JavaScript objects have a lot in common, especially when the datatype has record selectors. Suppose we have a simple *Book* type in Haskell:

```
data Book
  = Book
  { author :: JSString
  , title  :: JSString
  , pages  :: Int
  }
```

A concrete *Book* value would look as follows:

```
myBook
  = Book
  { author = toJS "me"
  , title  = toJS "story"
  , pages  = 123
  }
```

The representation of *myBook* closely resembles an object with the same data in JavaScript:

```
myBook
 =
```

```
{  author  : "me"
,  title   : "story"
,  pages   : 123
}
```

In fact, a JavaScript object very similar to the one shown above is already being generated by the UHC. However, since it is generated as an application of a constructor to some values, the generated datatype values are not directly usable in other JavaScript libraries. We require a mechanism to convert the Haskell representation of the datatype into a JavaScript representation. This idea is similar to that of the FFI's wrapper import feature. Using a similar mechanism to the wrapper, we can make Haskell datatypes available as JavaScript objects. This mechanism is exposed via de FEL, simply as { }:

foreign import *jscript* `"{}"`
  $mkObj :: a \rightarrow IO\ (JSPtr\ b)$

It takes a datatype a and converts it to a plain JavaScript object, resulting in a pointer to the new object. If the datatype contains record selectors, they will be used as the object's indices. When no record selectors are available, an integer is used instead.

Creating the object is achieved by recursively evaluating and cloning the data inside the datatype to a new, empty object, disposing of RTS-specific information in the process. Using the object wrapper, we can simplify our example from before:

```
main = do
  let b' = myBook {pages = pages myBook + 1}
  b ← mkObj b'
  p ← getAttr "pages" b
  print p   -- Prints 124
```

Note that even though this example is only one line shorter, we also have the two strings available in our JavaScript object, which would haven taken two more lines in the original example. More importantly, Haskell's type system is in a much better position to catch programmer mistakes, since record selectors are used in the modification of the *pages* value instead of strings.

3.2 Pure objects

Objects in JavaScript are mutable by nature. By modifying an object, you modify it for everything that has a pointer to that particular object. This forces any update operation to be defined in IO. In order to escape the IO monad, update operations need to become non-destructive, which is achieved by creating a copy of an object before modifying it. The RTS exports a primitive to do just this:

$primClone :: JSPtr\ a \rightarrow JSPtr\ a$

By cloning an object first, all pointers to the original object remain untouched when modifying the clone. This enables pure variants of the *primSetAttr* and *primModAttr* functions:

$primPureSetAttr :: JSString \rightarrow a \rightarrow JSPtr\ c \rightarrow JSPtr\ c$
$primPureModAttr :: JSString \rightarrow (a \rightarrow b) \rightarrow JSPtr\ c \rightarrow JSPtr\ c$

Since a potentially large tree of objects will be cloned by these pure functions, they should be used with care. The cloning method used is a modification of the cloning method used by jQuery[25].

4 The JCU Application

To explore the limitations, and to demonstrate the features of the UHC JavaScript back-end in a real-life scenario, we ported the 'JCU Prolog Proof Assistant'[30], a web application developed to aid in teaching[27] Prolog at the Junior College Utrecht. It is a tool developed for students to learn about important concepts in computer science, such as proofs, trees, unification, and backtracking, by means of proving Prolog queries manually. Students enter a Prolog query, after which they are tasked with constructing a proof by dragging and dropping Prolog rules and facts, and by applying substitutions manually throughout the proof tree.

The application was originally programmed in `coffeescript` [7], a layer of syntactic sugar for JavaScript, and used the `Brunch` [21] framework. In the original implementation, all Prolog logic was implemented server-side in Haskell, using the NanoProlog[29] library.

We rewrote the application in Haskell using the UHC and the UHC-JavaScript library. We also use jQuery for interacting with the DOM and the jQuery AjaxQueue[24] plugin for sequential non-blocking communication with the server. The resulting application has the same functionality as the original implementation and appears to be at least as stable, although this has only been manually tested. As is expected of applications that interact heavily with a graphical user interface, a large part of the application's code lives in the *IO* monad.

With the ability to compile Haskell to JavaScript comes the possibility of running any Haskell library that compiles on the UHC in the browser, without modification. We use this feature in the JCU web application to run the NanoProlog library in the browser, allowing us to perform proof checking and unification client-side, eliminating the need for many AJAX requests.

4.1 Implementation Issues

Most of the problems we encountered in porting the JCU application to Haskell were due to the lack of advanced language features in UHC, such as functional

dependencies and type families, amongst others. Practically, this implies that only part of the libraries available on Hackage today can currently be compiled to JavaScript using the UHC JavaScript back-end.

Another issue arises from JavaScript's scoping rules. In JavaScript, the keyword **this** is dynamically scoped while all other variables are lexically scoped. Since we emulate lazy evaluation by native JavaScript functions encapsulated by objects, the **this** keyword can in some cases point to the runtime system, rather than the expected scope, exposing the runtime system to the programmer. Simply importing **this** as a function using the FFI is not an option then.

A common use-case of when this might happen is when an imported JavaScript library expects the programmer to make use of the **this** keyword in a callback function. The jQuery library, for example, expects event callbacks to get the active DOM-node using the **this** keyword. One way to still get a reference to the expected object when using **this** is to create a wrapper function that captures the expected scope and passes it to the wrapped function as explicit argument. We have implemented this solution in the *wrappedThis* function, which is part of our RTS.

Figure 2 shows how the *wrappedThis* function can be used to obtain the value of an HTML input field. *valString* is a function that gets the value of a jQuery object as a *String*, while *alert* shows an alert box containing the provided message. We query the DOM using jQuery, retrieving all `input` elements, such as text fields, in the DOM. We define a function *alertHndlr* that takes the string value of a jQuery object and then shows it in an alert box. Note the explicit **this** parameter. We then wrap it so it becomes a JavaScript function, after which we partially apply it to an explicit **this** parameter using *wrappedThis*. Finally, we bind the event handler to all input fields retrieved by our jQuery selector.

A last example of implementation difficulties is found in the lack of threading support in our current implementation. In addition to the web-based proof exerciser, we offer a web-based user interface to NanoProlog's interpreter. In some cases, the interpreter can get stuck in an infinite recursion when trying to unify a rule. For example, trying to proof the query $silly\ (X)$, where *silly* is defined as $silly\ (X) \vdash silly\ (X)$. will never terminate. Originally, we spawned a new thread on the server, which we would terminate after a given amount of time. Our current approach, however, does not yet offer threading, risking blocking the client-side process causing a tab or the whole browser to hang. JavaScript's WebWorkers might provide a solution to this problem, although we have yet to investigate this option. Another solution would be to change the implementation to limit it's recursion depth.

4.2 Performance

In general, the performance of the web application is on par with the original implementation in JavaScript, but only when using a state of the art JavaScript engine, as is found in Google Chrome or Safari. The biggest bottleneck seems to

```
data JQueryPtr
type JQuery = JSPtr JQueryPtr

foreign import js "%1.bind(%*)"
  bind :: JQuery→JSString→JEventHandler→IO ()

type ThisEventHandler   = JQuery→JQuery→JEventResult
type JEventHandler      = JSFunPtr (JQuery→JEventResult)
type JThisEventHandler = JSFunPtr ThisEventHandler

valString :: JQuery→IO String
mkJThisEventHandler :: ThisEventHandler→IO JThisEventHandler

foreign import js "wrappedThis(%1)"
  wrappedThis :: JThisEventHandler→IO JEventHandler

bindInput = do
  let alertHndlr :: ThisEventHandler
      alertHndlr this _ = valString this >>= alert
  inputField ← jQuery "input"
  eh ← mkJThisEventHandler alertHndlr >>= wrappedThis
  bind inputField (toJS "blur") eh
```

Fig. 2. Code for adding an event handler to an input field

be memory management. Building up lazy Haskell expressions leads to a large amount of JavaScript objects. The quick creation and then successive destruction of these large expressions places a strain on the memory manager and garbage collector. Other popular browsers, such as Firefox, Opera, and Internet Explorer, perform significantly worse than the aforementioned browsers, although this has only been tested informally.

5 Future Work

While we have shown that it already is possible to implement an entire JavaScript application in Haskell, there is still a lot of room for improvement. As mentioned before, the UHC itself lacks support for the more advanced Haskell features. Implementing these in the UHC would go a long way to making the UHC JavaScript back-end a true alternative to JavaScript.

Our current UHC-JavaScript library relies on the programmer to use imported functions correctly. The object-wrapper import, for example, will currently try to wrap anything, possibly failing at runtime. Extra constraints could be added, although the RTS cannot currently deal with them. Eventually, one could image a higher-level library being built on top of the low-level imports to provide improved type-safety. Such libraries may be based on generic programming to eliminate repetition, function reactive programming[13, 31, 6] to interact with the DOM, or they may be an entire user-interface toolkit, such as wxHaskell[19].

Working with WebWorkers as an alternative to Haskell threads is currently not investigated yet. Our JCU application would become significantly more usable with a threading alternative.

Communication with the server is currently encoded manually, possibly using existing libraries, like jQuery. One could imagine an approach inspired by Cloud Haskell's[14] typed channels, where communication proceeds over type-safe communication channels, abstracting away from the actual AJAX call.

Currently the only way of converting a datatype to a JavaScript object is to do so at runtime. This, however, is a process with time complexity linear in the number of datatype records. Future work could focus on generating (parts of) JavaScript objects at compile-time, so that only dynamic values will need to be copied to the object at runtime.

Targeting Haskell to a different platform means that some assumptions following from using a single platform only are no longer valid. First, a different platform means a different runtime environment. Almost all of the UNIX functionality is available for the usual Haskell UNIX runtime, but is naturally not available inside a web browser and, vice verse, specific JavaScript libraries like jQuery are not available on a UNIX platform. Some library modules of a package (partially) cannot be build on some platforms, while others (partially) can. To cater for this, UHC rather ad-hoc marks modules to be unavailable for a backend by a pragma `{-# EXCLUDE_IF_TARGET js #-}`. Of course *cpp* can still be used to select functionality inside a module. However, in general, awareness of platform permeates all aspects of a language system, from the compiler itself to the library build system like *Cabal*.

In particular, *Cabal* needs a specification mechanism for such variation in target and platform to allow for selective compilation of a collection of variants. Currently this means that UHC compilation for the JavaScript backend cannot be done through *Cabal*.

A second aspect has more to do with the evolution of Haskell as an ecosystem. Many libraries go far beyond the Haskell standard by making use of GHC extensions. Currently, such libraries evolve to use type families, a feature not yet available in UHC. For (non GHC) Haskell compiler writers to keep with this pace of evolution poses a considerable challenge; yet in our opinion there is value in the availability of compiler alternatives as well as variation in what those compilers are good at.

Currently, we generate JavaScript from the compiler's core language. It might be possible to generate faster code which uses native JavaScript language features when generating JavaScript at a later stage in the compiler pipeline, where the intermediate code is more imperative in nature.

6 Related work

The idea of running Haskell in a browser is not new. To our knowledge first attempts to do so using JavaScript were done in the context of the York Haskell Compiler (YHC) [3]. The Document Object Model (DOM) inside a browser was accessed via wrapper code generated from HTML standard definitions [2]. However, YHC is no longer maintained and direct interfacing to the DOM nowadays is replaced by libraries built on top of the multiple DOM variations.

The idea of running functional programs in a browser even goes further back to the availability of Java applets. The workflow framework *iTasks*, built on top of the Clean system [5], uses a minimal platform independent functional language, SAPL, which is interpreted in the browser by code written in Java. The latest interpreter incarnations are written in JavaScript [16, 8, 23]. Although currently a Haskell front-end exists for Clean, the use of it in a browser seems to be tied up to the iTasks system. The intermediate language SAPL also does not provide the facilities as provided by our Haskell FFI.

Of the GHC a version exists which generates JavaScript [22], based on the GHC API, supporting the use of primitives but not the FFI. Further down we elaborate on some consequences of multiple platforms and backends relevant for this GHC backend variant as well.

Both "Functional javascript" [26] and "Haskell in Javascript" [4] do not use a separate Haskell compiler. Instead, JavaScript is used directly in a functional style, respectively a small compiler for a subset of Haskell has been written in JavaScript.

A more recent attempt at cross-compiling Haskell to JavaScript it the Fay language[11], which aims to support a subset of Haskell. It shows promise, and even draws some inspiration from the work we present here, namely the FEL.

7 Conclusion

We have shown that the UHC is capable of supporting the development of complete client-side web applications. This opens the door to Haskell-only web development. In the process we added the FEL to UHC and provided a library that exposes the JavaScript world to Haskell.

Better abstractions are still required to reduce the amount of code that lives in the *IO* monad directly, and to give programming with the UHC JavaScript backend a more functional feel. While in most cases performance is acceptable, it needs to be improved if computationally heavy functions are to be run on the client. In order for most of the frequently used Hackage libraries to be run on the client, UHC and Cabal will need some more work as well.

References

1. Uhc git repository. `https://github.com/UU-ComputerScience/uhc/`.

2. Haskell in web browser.
`http://www.haskell.org/haskellwiki/Haskell_in_web_browser`, 2007.
3. Yhc/Javascript. `http://www.haskell.org/haskellwiki/Yhc/Javascript`, 2007.
4. A haskell interpreter in javascript.
`https://github.com/johang88/haskellinjavascript`, 2010.
5. Clean. `http://wiki.clean.cs.ru.nl/Clean`, 2011.
6. Heinrich Apfelmus. Reactive banana.
`http://www.haskell.org/haskellwiki/Reactive-banana`.
7. Jeremy Ashkenas. Coffeescript. `http://coffeescript.org/`.
8. Eddy Bruël and Jan Martin Jansen. Implementing a non-strict purely Functional Language in JavaScript. In *Implementation of Functional Languages*, 2010.
9. Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The Architecture of the Utrecht Haskell Compiler. In *Haskell Symposium*, 2009.
10. Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. UHC Utrecht Haskell Compiler. `http://www.cs.uu.nl/wiki/UHC`, 2009.
11. Chris Done. Fay programming language. `http://fay-lang.org/`.
12. ECMA International, Geneva, Switzerland. ECMAScript Language Specification. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, 2011.
13. Conal M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.
14. Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the Cloud. 2011.
15. Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric lenses. *SIGPLAN Not.*, 46(1):371–384, January 2011.
16. Jan Martin Jansen. *Functional Web Applications, Implementation and Use of Client-Side Interpreters*. PhD thesis, Radboud University Nijmegen, 2010.
17. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002.
18. Koji Kagawa. Compositional references for stateful functional programming. *SIGPLAN Not.*, 32(8):217–226, August 1997.
19. Daan Leijen. wxhaskell.
20. Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, London, UK, UK, 1991. Springer-Verlag.
21. Paul Miller, Nik Graf, Thomas Schranz, and Andreas Gerstmayr. Brunch.io. `http://brunch.io/`.
22. Victor Nazarov. ghcjs: Haskell to Javascript compiler (via GHC).
`https://github.com/sviperll/ghcjs`, 2011.
23. Rinus Plasmeijer, Jan Martin Jansen, and Pieter Koopman. Declarative Ajax and Client Side Evaluation of Workflows using iTasks. In *Principles and Practice of Declarative Programming*, 2008.
24. Oleg Podolsky. jquery-ajaxq. `http://code.google.com/p/jquery-ajaxq/`.
25. John Resig. jQuery. `http://jquery.com`.
26. Oliver Steele. Functional Javascript.
`http://osteele.com/sources/javascript/functional/`, 2007.
27. Jurriën Stutterheim, Wouter Swierstra, and Doaitse Swierstra. Forty hours of declarative programming – teaching prolog at the junior college utrecht. 2012.

28. Jurriën Stutterheim, Alessandro Vermeulen, and Atze Dijkstra. Uhc-javascript libraries. `https://github.com/UU-ComputerScience/uhc-js`.
29. Doaitse Swierstra and Jurriën Stutterheim. Nanoprolog package. `http://hackage.haskell.org/package/NanoProlog`.
30. Wouter Swierstra, S. Doaitse Swierstra, and Jurriën Stutterheim. Logisch en Functioneel Programmeren voor Wiskunde D. Technical Report UU-CS-2011-033, Universiteit Utrecht, 2011.
31. Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 242–252, New York, NY, USA, 2000. ACM.

Push-Pull Signal-Function Functional Reactive Programming

Edward Amsden

Rochester Institute of Technology
eca7215@cs.rit.edu

Abstract. Functional Reactive Programming is a promising class of systems for writing interactive and time-dependent programs. Signal-function FRP is a subclass of these systems which provides advantages of modularity and correctness, but has proven difficult to efficiently implement.
The abstraction of signal vectors provides the necessary type apparatus to distinguish components of the input and output of signal functions which benefit from a push-based implementation from those which benefit from a pull-based implementation, and to combine both implementation strategies in a single system.
We describe a signal-function FRP system which provides push-based evaluation for events, pull-based evaluation for signals, and a simple monadic evaluation interface which permits the system to be easily integrated with one or more IO systems.

1 Introduction

Functional Reactive Programming (FRP) is a class of systems for describing reactive programs. Reactive programs are programs which, rather than taking a single input and producing a single output, must accept multiple inputs and alter temporal behavior, including the production of multiple outputs, based on these inputs.

An FRP system will provide a means of manipulating *behaviors* and *events*. Behaviors are often referred to as *signals* in FRP literature, but the definition is the same. A behavior or signal is, semantically, a function from time to a value. An event is a discrete, possibly infinite, and time-ordered sequence of occurrences, which are times paired with values.

FRP systems can generally be categorized as "classic FRP," which corresponds to the originally described FRP system in that behaviors and events are manipulated directly and are first-class values in the FRP system, or "signal-function FRP," in which behaviors (generally termed signals in this approach) and events are not first-class values, but signal functions are first class values. Signal functions are time-dependent and reactive transformers of signals, events, or combinations of signals and events.

FRP combines behaviors and events through the use of *switching*, in which a behavior (in classic FRP) or a signal function (in signal-function FRP) is replaced by a new behavior or signal function carried by an event occurrence.

Classic FRP was first described as an system for interactive animations [1]. Recent work on classic FRP has focused on efficient implementation. Reactive is a system for push-pull FRP [2]. Push-based evaluation evaluates a system only when input is available, and is thus suitable for discrete inputs such as events. Pull-based evaluation evaluates the system as quickly as possible, polling for input, and is preferable for behaviors and signals. The initial implementations of FRP made use of pull-based evaluation for both behaviors and events. Reactive, as well as more recent systems such as "reactive-banana" [3], make use of push-based evaluation for events and pull-based evaluation for behaviors.

All implementations of signal-function FRP to date [4–7] have used pull-based evaluation. This is due to the ease of implementation of pull-based evaluation, and the types used for signal functions which do not permit distinguishing signals and events, or constructing only part of the input (for instance, one event occurrence.)

A recent extension of signal-function FRP called N-Ary FRP [7] describes a method of typing signal functions which, as we will show, enables the push-based evaluation of events in a signal-function FRP system. The notion of signal vectors allows the representation of signal function inputs and outputs as combinations of signals and events, rather than a single signal which may contain multiple values, including option values for events. Signal vectors are uninhabited types, which can be used to type partial or full representations of the signal function inputs and outputs.

We present TimeFlies,[1] a push-pull signal-function FRP system. We hope to demonstrate the feasibility of such an approach to FRP, and provide a basis for further research into efficient implementation of signal-function FRP. We also describe a powerful evaluation interface for TimeFlies, which permits us to use TimeFlies to describe applications which make use of multiple and differing IO libraries.

Section 2 describes design choices for the system, and provides an overview of the interface. Section 3 describes how the system is implemented, and how the separation of evaluation between events and signals is achieved. Section 4 is a discussion of the usefulness of our implementation. Section 5 describes the current and future work on this system. Section 6 gives an overview of related efforts. Section 7 concludes.

2 System Design

Our goal is to produce a composable and efficient FRP system. Signal function FRP has an advantage in terms of composability, because it permits the construction of self-contained objects through both input and output composition, rather than purely through output composition as in classic FRP. Signal-function

[1] The sentence "Time flies like an arrow." is a favorite quotation of one of the author's philosophy instructors, used to demonstrate the ambiguity of language. The origin of the quotation is unknown.

```
data SVEmpty
data SVSignal a
data SVEvent a
data SVAppend svLeft svRight
```

Fig. 1. Signal vectors.

FRP also avoids problematic properties common to classic FRP systems such as a large class of time and space leaks [8].

Efficient implementations of signal-function FRP have been approached through runtime optimization [6], but all implementations have been pull-based for both signals and events. A truly efficient implementation will likely combine run-time optimization with push-based evaluation for events.

The concept of N-Ary FRP was to encode additional safety properties into the types of a signal-function FRP system [7]. This system introduced the concept of *signal vectors* as input and output types for signal functions. Signal vectors are combinations of signals and events. In N-Ary FRP, signal functions are represented at the type level, and type-level functions (type families in Haskell) are used to specify the representations of signal functions.

In our system, we construct representations of signal vectors using Generalized Algebraic Datatypes [9, 10]. The use of GADTs enables us to use signal vectors to instantiate type parameters in the types of signal vector representations, as well as in the types of signal functions. GADTs are available as an extension in the Glasgow Haskell Compiler [11].

This approach permits us to construct partial inputs and outputs for signal functions. For instance, we can construct a representation for a single event in a signal vector, and another representation of updated values for a subset of the signals in the signal vector, and yet another carrying values for all signals in a signal vector. The signal vector types are shown in Figure 1.

With the ability to construct partial representations of signal vectors, we can represent signal functions with a datatype carrying multiple functions, one for each type of input. This allows us to separate the pull-based processing of signals from the push-based reaction to events.

The exposed interface is a set of combinators for constructing signal functions, as well as combinators for describing the evaluation of a signal function. The interface is shown in Fig. 2.

Signal functions are produced by combining primitive signal functions using the `>>>` (sequential composition), `first`, and `second` routing combinators. Other routing signal functions are provided, but they are intended to be combined with, rather than to modify, other signal functions.

2.1 Examples of Signal Function Primitives

The most basic signal is the `identity` signal function, which, as its name suggests, simply passes its input to its output.

```
-- Signal Functions
type :~> svIn svOut

-- Infix type alias for SVAppend
type :^: svLeft svRight = SVAppend svLeft svRight

-- Basic signal functions
identity :: sv :~> sv
constant :: a -> SVEmpty :~> SVSignal a
never    :: SVEmpty :~> SVEvent a
asap     :: a -> SVEmpty :~> SVEvent a
after    :: Double -> a -> SVEmpty :~> SVEvent a

-- Lifting pure functions
pureSignalTransformer :: (a -> b) -> SVSignal a :~> SVSignal a
pureEventTransformer  :: (a -> b) -> SVEvent a :~> SVEvent a

-- Composition and routing
(>>>)           :: (svIn :~> svMiddle) -> (svMiddle :~> svOut)
                   -> (svIn :~> svOut)
first           :: (svIn :~> svOut) -> (svIn :^: sv) :~> (svOut :^: sv)
second          :: (svIn :~> svOut) -> (sv :^: svIn) :~> (sv :^: svOut)
swap            :: (svLeft :^: svRight) :~> (svRight :^: svLeft)
copy            :: sv :~> (sv :^: sv)
ignore          :: sv :~> SVEmpty
cancelLeft      :: (SVEmpty :^: sv) :~> sv
cancelRight     :: (sv :^: SVEmpty) :~> sv
uncancelLeft    :: sv :~> (SVEmpty :^: sv)
uncancelRight   :: sv :~> (sv :^: SVEmpty)
associate       :: ((sv1 :^: sv2) :^: sv3) :~> (sv1 :^: (sv2 :^: sv3))
unassociate     :: (sv1 :^: (sv2 :^: sv3)) :~> ((sv1 :^: sv2) :^: sv3)

-- Reactivity
switch :: (svIn :~> (svOut :^: SVEvent (svIn :~> svOut)))
          -> svIn :~> svOut

-- Feedback
loop :: ((svIn :^: svLoop) :~> (svOut :^: svLoop)) -> svIn :~> svOut

-- Time dependence
time  :: SVEmpty :~> Double
delay :: Double -> (SVEvent a :^: SVEvent Double) :~> SVEvent a
class TimeIntegrate

-- Joining
union          :: (SVEvent a :^: SVEvent a) :~> SVEvent a
combineSignals :: (a -> b -> c)
                  -> (SVSignal a :^: SVSignal b) :~> SVSignal c
capture        :: (SVSignal a :^: SVEvent b) :~> SVEvent a

-- Events
filter         :: (a -> Maybe b) -> SVEvent a :~> SVEvent b
```

Fig. 2. Signal function interface.

The `constant` signal function has an empty input, and produces a constant signal as its output.

The `asap` signal function produces an event at the first time interval after it begins, and never again.

The `filter` signal function applies a predicate to the values of event occurrences and passes them along only if the predicate is true of the value. The predicate outputs `Maybe` values to allow the user to avoid partial functions on types such as `Maybe` or `Either`.

The `associate` signal function is a routing function, which takes as input a combination of three signal vectors to which the signal vector append constructor is combined left-associatively, and produce the same signal vector as output, but with the signal vector append constructor combined right-associatively.

The `switch` function is the essential signal function for reactivity. It acts as the supplied signal function until that signal function produces an event on its right output. This event's value is a new signal function, which replaces the switch signal function.

The `loop` function permits a signal function to receive its own output as an input.

Implementations for these examples will be discussed in Sec. 3.3.

2.2 Evaluation Interface

The evaluation interface provides a monad [12, 13] for specifying input to signal functions and handling their output. This interface also allows input actions to be separated from output actions, and from each other, so that a signal function may receive input from multiple IO systems and have its output handled by multiple output systems.

The evaluation interfaceis a monad transformer [14]. This allows a signal function evaluation to be constructed from primitive evaluation actions (event pushing, input signal updating, and sampling) as well as actions from the underlying monad. These actions can then be run in the underlying monad to actuate the signal function. Because the actuation takes a signal function's state as input and produces a new state as output, in addition to the monadic side effects of handling the signal function's output, different evaluation actions may be taken on the same signal function from distinct locations within an application's code. This allows the evaluation interface to be easily integrated with event-loop style systems as well as traditional imperative IO systems.

The evaluation interface consists of the monad transformer type, functions for constructing event inputs, initial input signal samples, and input signal updates, functions for constructing the vector of output handlers for a signal function, and evaluation actions. The full interface is shown in Fig. 3.

```
-- Input helpers
class SVRoutable -- Instances: SVSignalUpdate,
                 -- SVEventOccurrence
svLeft :: (SVRoutable r) =>
          r svLeft -> r (SVAppend svLeft svRight)
svRight :: (SVRoutable r) =>
           r svRight -> r (SVAppend svLeft svRight)

-- Signal inputs
data SVSignalUpdate
data SVSample
sample :: a -> SVSample (SVSignal a)
sampleEvt :: SVSample (SVEvent a)
sampleNothing :: SVSample SVEmpty
combineSamples :: SVSample svLeft -> svSample svRight
                  -> svSample (svLeft :^: svRight)
svSig :: a -> SVSignalUpdate (SVSignal a)

-- Event inputs
data SVEventInput
svOcc :: a -> SVEventInput (SVEvent a)
-- Actuation
update :: (Monad m) => SVSignalUpdate svIn
           -> SFEvalT svIn svOut m ()
push   :: (Monad m) => SVEventInput svIn
          -> SFEvalT svIn svOut m ()
sample :: SVEvalT svIn svOut m ()
-- Running
data SVHandler
emptyHandler :: SVHandler m SVEmpty
eventHandler :: (a -> m ())
                -> SVHandler m (SVEvent a)
signalHandler :: (a -> m ())
                 -> SVHandler m (SVSignal a)
combineHandlers :: SVHandler m svLeft
                   -> SVHandler m svRight
                   -> SVHandler m (svLeft :^: svRight)
initSFEvalT :: SVHandler m svOut -> SVSample svIn
               -> Double ->(svIn :~> svOut)
               -> SFEvalState m svIn svOut
runSFEvalT :: SFEvalT svIn svOut m a
              -> SFEvalState m svIn svOut
              -> m a
```

Fig. 3. Evaluation interface.

```
data SVSample sv where
  SVSample    :: a -> SVSample (SVSignal a)
  SVSampleEvt :: SVSample (SVEvent a)
  SVNothing   :: SVSample SVEmpty
  SVBoth      :: SVSample svLeft -> SVSample svRight
                 -> SVSample (svLeft :^: svRight)

sample         :: a -> SVSample a
sampleEvt      :: SVSample (SVEvent a)
sampleNothing  :: SVSample SVEmpty
combineSamples :: SVSample svLeft -> SVSample svRight
                  -> SVSample (svLeft :^: svRight)
splitSample    :: SVSample (svLeft :^: svRight)
                  -> (SVSample svLeft, svSample svRight)
sampleValue    :: SVSample (SVSignal a) -> a
```

Fig. 4. Signal sample representation.

3 Implementation

We now turn our attention to the implementation of the signal function system. We will discuss representations of inputs, outputs, and signal functions, as well as the implementations of specific signal function combinators.

3.1 Input and Output Representations

As discussed in Section 2, our implementation requires representations of the inputs and outputs of signal functions. Since signal functions are typed using signal vectors, the representation types must be parameterized over signal vectors. But signal vectors are uninhabited types, and in ordinary Haskell we cannot declare a data constructor which uses a component of a type parameter. Further, we cannot restrict which types may instantiate a type parameter for a particular data constructor.

GADTs lift these restrictions, permitting type constraints to be applied to individual data constructors. In the declaration of a GADT, a type signature is provided for each data constructor, including the types of the constructor's parameters. When a GADT is pattern-matched, the types of the parameters are inferred using the constraints given in this signature. This is called type refinement.

Using GADTs, we can construct representations of signal vectors to use in our implementation. The first such representation carries a value for each signal component of a signal vector, and is shown in Figure 4.

A value of this type represents the entire vector, though it has nullary constructors at event leaves and empty leaves. For event occurrences, we want to represent only a single point in the vector, as shown in Figure 5. This representation uses two constructors, each of which leaves one or the other of the type variables in the signal vector `SVAppend` constructor unconstrained.

```
data SVOccurrence sv where
  SVOccurrence :: a -> SVOccurrence (SVEvent a)
  SVOccLeft    :: SVOccurrence svLeft
                  -> SVOccurrence (svLeft :^: svRight)
  SVOccRight   :: SVOccurrence svRight
                  -> SVOccurrence (svLeft :^: svRight)

occurrence        :: a -> SVOccurrence (SVEvent a)
occLeft           :: SVOccurrence svLeft
                     -> SVOccurrence (svLeft :^: svRight)
occRight          :: SVOccurrence svRight
                     -> SVOccurrence (svLeft :^: svRight)
chooseOccurrences :: SVOccurrence (svLeft :^: svRight)
                     -> Either (SVOccurrence svLeft) (SVOccurrence svRight)
fromOccurrence    :: SVOccurrence (SVEvent a) -> a
```

Fig. 5. Event occurrence representation.

```
data SVDelta sv where
  SVDelta        :: a -> SVDelta (SVSignal a)
  SVDeltaNothing :: SVDelta sv
  SVDeltaBoth    :: SVDelta svLeft -> SVDelta svRight
                    -> SVDelta (svLeft :^: svRight)
delta         :: a -> SVDelta (SVSignal a)
deltaNothing  :: SVDelta sv
combineDeltas :: SVDelta svLeft -> SVDelta svRight
                 -> SVDelta (svLeft :^: svRight)
updateSample  :: SVDelta sv -> SVSample sv -> SVSample sv
```

Fig. 6. Signal delta representation.

The final representation "signal deltas," carries replacement values for some points in a signal vector. Here, we combine the unconstrained type variable from the occurrence representation with the "Both" constructor from the sample representation. The unconstrained type variable will be placed in a separate constructor, which allows the construction of empty signal deltas, as is shown in Figure 6.

3.2 Signal Function Representations

With representations for signal function inputs and outputs in place, we turn to the representation of signal functions. A signal function must be able to respond to time increments and new signal samples, as well as event occurrences. Since we wish to be able to "push" event occurrences independently of sampling, events are handled using the last-updated time and signal sample.

When a signal function is asked to respond to either of these types of inputs, it must produce an output consisting of zero or more event occurrences, a new

```
data Initialized
data NonInitialized

type :~> svIn svOut = SF NonInitialized svIn svOut

data SF init svIn svOut where
  SF :: (SVSample svIn -> (SVSample svOut, SF Initialized svIn svOut))
        -> SF NonInitialized svIn svOut
  SFInit :: (Double -> SVDelta svIn
              -> (SVDelta svOut, [SVOccurrence svOut],
                  SF Initialized svIn svOut))
             -> (SVOccurrence svIn -> ([SVOccurrence svOut],
                                       SF Initialized svIn svOut))
             -> SF Initialized svIn svOut
```

Fig. 7. Signal function representation

signal function with the same type (to enable reactivity and state), and, if it is responding to a time and sample update, a delta which represents updates to the sample of its output signals.

Finally, there is a special case for a signal function at time 0. It must be provided with an initial input sample before it can respond to incremental time and sample updates or to event occurrences.

Signal functions are represented as a datatype with two constructors. The first constructor wraps a function from an input sample to an output sample and an initialized signal function. The second wraps two functions. The first accepts a time delta and a signal delta, and produces a signal delta, a list of event occurrences, and a new signal function as the output. The second accepts an event occurrence and produces a list of event occurrences and a new signal function as the output. The signal function representation is shown in Figure 7.

This representation allows a signal function to respond to events and time updates separately, and does not enforce the representation of events during time updates, as previous signal function systems do.

3.3 Signal Function Implementations

Now that we have established representations for signal functions and their inputs and outputs, we show several examples of how signal function combinators are implemented.

The `identity` signal function's initialization function takes a sample, and returns that sample as its output sample. The initialized function returned is the `identityInit` function, which is not exported by the module. The time and sample function ignores the provided time delta, produces the input signal delta as the output signal delta, and returns `identityInit` as the new signal function. The event function takes an event occurrence and returns a singleton

```
identity :: sv :~> sv
identity = SF (\sample -> (sample, identityInit))

identityInit :: SF Initialized sv sv
identityInit = SFInit (\_ delta -> (delta, [], identityInit))
                      (\occ -> ([occ], identityInit))
```

Fig. 8. `identity` signal function implementation.

list containing that occurrence, along with the `identityInit` function as the replacement signal function. The implementation is shown in Figure 8.

The `constant` signal function is initialized with a value, which it then produces as its output forever. Thus, the initialized version of `constant` never produces any output other than itself as a replacement signal function, and an empty signal delta. The implementation of `constant` is shown in Fig. 9.

```
constant :: a -> SVEmpty :~> SVSignal a
constant x = SF (\_ -> (constantInit, sample x))

constantInit :: SF Initialized SVEmpty (SVSignal a)
constantInit = SFInit (\_ _ -> (deltaNothing, [], constantInit))
                      (\_ -> ([], constantInit))
```

Fig. 9. `constant` signal function implementation.

The `asap` signal function implementation requires a parameter for the initialized version of the signal function, to specify the value of the event occurrence. It replaces itself with the initialized version of the `never` signal function after the first time step. The implementation is shown in Fig. 10.

```
asap :: a -> SVEmpty :~> SVEvent a
asap x = SF (\_ -> (sampleEvt, asapInit x))

asapInit :: a -> SF Initialized SVEmpty (SVEvent a)
asapInit x = SFInit (\_ _ -> (deltaNothing, [occurrence x], neverInit))
                    (\_ -> (never))
```

Fig. 10. `asap` signal function implementation.

The `filter` function accepts events, applies a `Maybe` predicate to their values, and produces the value produced by the predicate as an event occurrence, or no occurrence if the value is `Nothing`. It is implemented by having the event

```
filter :: (a -> Maybe b) -> SVEvent a :~> SVEvent b
filter p = SF (\_ -> (sampleEvt, filterInit p))

filterInit :: (a -> Maybe b) -> SF Initialized (SVEvent a) (SVEvent b)
filterInit p = SFInit (\_ _ -> (deltaNothing, [], filterInit p))
                      (\evtOcc -> (maybe [] ((:[]) . occurrence) $
                                   fromOccurrence evtOcc,
                                   filterInit p))
```

Fig. 11. `filter` signal function implementation.

occurrence response function apply the `maybe` function from the Haskell prelude to the value returned by the predicate, as shown in Fig. 11

The `associate` signal function is a routing signal function. It transforms a signal vector which is left-associated at the top level to one that is right-associated at the top level. Its implementation is representative of all of the routing functions, and is shown in Fig. 12.

The `switch` function is the basic combinator used to introduce reactivity. It is given a signal function whose output is the append of two signal vectors. The left side of the signal vector is passed on as the output of the reactive signal function, and the right side is an event carrying signal functions. When an event occurrence is present on the right-side event output, the signal function carried by this occurrence replaces the signal function constructed by switch. Due to space concerns, the implementation is elided, but a brief description will suffice. The signal function stores the input sample provided during initialization, and updates it with deltas. When an event occurrence carrying a signal function is produced by either the event or time function of the wrapped signal function, the stored signal sample is used to initialize the new signal function. If this occurs while handling an event input, the sample output by the new signal function's initialization is stored and is combined with its first output delta to produce the output delta at the next time step.

The `loop` function allows a signal function to see a component of its own output as input. This is primarily useful when a signal function has components which mutually depend on each others outputs, such as in physics simulations or games. Care must be taken that the feedback output is not immediately dependent on the feedback input, or sampling the signal function will not terminate. Loop is implemented by generating a recursive list (which will be infinite if the feedback is not decoupled) of event inputs, and by splitting the output signal delta and using the right side output delta as the right side input delta within a recursive let-binding. (The let-binding in Haskell always admits recursive bindings.) If the system is not completely decoupled, this will result in non-termination during an evaluation step. Decoupling can be achieved using `delay` primitive.

```
associate :: SVAppend (SVAppend sv1 sv2) sv3
             :~> SVAppend sv1 (SVAppend sv2 sv3)
associate =
  SF (\sigSample -> let (sigSampleLeft, sigSampleRight) =
                          splitSample sigSample
                        (sigSampleLeftLeft, sigSampleLeftRight) =
                          splitSample sigSampleLeft
                    in (combineSamples sigSampleLeftLeft
                        (combineSamples sigSampleLeftRight sigSampleRight),
                        associateInit))

associateInit :: SF Initialized (SVAppend (SVAppend sv1 sv2) sv3)
                                (SVAppend sv1 (SVAppend sv2 sv3))
associateInit = SFInit (\_ sigDelta -> let (sigDeltaLeft, sigDeltaRight) =
                                             splitDelta sigDelta
                                           (sigDeltaLeftLeft,
                                            sigDeltaLeftRight) =
                                             splitDelta sigDeltaLeft
                                       in (combineDeltas sigDeltaLeftLeft
                                           (combineDeltas sigDeltaLeftRight
                                                          sigDeltaRight),
                                           [], associateInit))
                       (\evtOcc -> (case chooseOccurrence evtOcc of
                                      Left leftOcc ->
                                        case chooseOccurrence leftOcc of
                                          Left leftLeftOcc ->
                                            [occLeft leftLeftOcc]
                                          Right leftRightOcc ->
                                            [occRight $ occLeft leftRightOcc]
                                      Right rightOcc ->
                                        [occRight $ occRight $ rightOcc],
                                    associateInit))
```

Fig. 12. `associate` signal function implementation.

3.4 Evaluation Interface Implementation

The evaluation interface must maintain state including the current signal function, the current time (to produce time deltas), updates to the input sample which have yet to be sampled, and the output handlers.

The output handlers are stored in a structure similar to that for a signal sample. The difference is that both signal and event leaves contain values, and these values are functions from the leaf type to another type. The handler datatype is shown in Fig. 13.

We now need a datatype to hold the various components of the evaluation state. This type is shown in Fig 14.

The evaluation interface itself is a monad transformer, which we implement as a Haskell newtype wrapping the `StateT` monad transformer. We do not ex-

```
data SVHandler out sv where
  SVHandlerEmpty  :: SVHandler out SVEmpty
  SVHandlerSignal :: (a -> out) -> SVHandler out (SVSignal a)
  SVHandlerEvent  :: (a -> out) -> SVHandler out (SVEvent a)
  SVHandlerBoth   :: SVHandler out svLeft -> SVHandler out svRight
                     -> SVHandler out (SVAppend svLeft svRight)
```

Fig. 13. Handler datatype.

```
data SFEvalState m svIn svOut
  = SFEvalState {
                  esSF :: SF Initialized svIn svOut,
                  esOutputHandlers :: SVHandler (m ()) svOut,
                  esLastTime :: Double,
                  esDelta :: SVDelta svIn
                }
```

Fig. 14. Evaluation state datatype.

port the raw `put` and `get` actions of the state monad, but instead implement the push, update, and sampling operations for signal function evaluation using `put` and `get`. The evaluation monad transformer is shown in Fig. 15. Instances of typeclasses including `Monad` and `MonadTrans` are derived using the "GeneralizedNewtypeDeriving" extension to the Glasgow Haskell Compiler.

4 Discussion

The system presented here, TimeFlies, demonstrates how using signal vectors to type inputs and outputs enables push-based evaluation of events in a signal-function system. We take advantage of this representation in several ways.

First, by separating components of inputs and outputs in the types, we are free to create distinct, and often partial, representations of the input or output of a signal function. This enables us to represent only the event occurrence being pushed at that time.

Second, this separation also permits us to separate the process of gathering the input to a signal function, and the process of handling its output, into different points in a program. Using the evaluation interface described, an event occurrence may be pushed onto one input of a signal function from one point in a program (e.g. a mouse click handler), an input signal may be updated in another (e.g. a mouse movement handler), and finally the system may be sampled in a third place (e.g. an animation or audio timed callback).

Finally, this approach enables further work on the implementation of the signal function system to be separated from changes in the interface. By enabling differing representations of the inputs and outputs of signal functions, we are free

```
newtype SFEvalT svIn svOut m a = StateT (SFEvalState m svIn svOut) m a
```

Fig. 15. Signal function evaluation monad transformer.

to change these representations without the need to further constrain the input and output types.

5 Ongoing and Further Work

TimeFlies, the system described here, has been implemented, but not extensively tested. The immediate goal is to create a real-time application which will permit a performance and implementation comparison of TimeFlies with Yampa, the current state-of-the-art pull-based signal-function system.

In the future, we hope to apply run-time optimizations, using the technique used for Yampa, to create a push-pull self-optimizing signal-function system. Further, we hope to use this system as a basis for exploring signal-function FRP as a basis for general-purpose application frameworks.

6 Related Work

Signal Function FRP was introduced as a model for Graphical User Interfaces [4]. The system was originally termed "AFRP" (Arrowized FRP). Yampa is a rewrite of AFRP where signal functions apply a number of ad-hoc optimizatons to themselves as they evolve. Yampa demonstrated a modest performance improvement over AFRP [6].

Reactive is a classic FRP system which implements push-based evaluation for events by transforming behaviors to "reactive normal form," where a behavior is a non-reactive behavior running inside a switch, whose event stream carries behaviors in reactive normal form. The system is evaluated by forking a Haskell thread to repeatedly sample the non-reactive behavior, and then blocking on the evaluation of the first occurrence in the event stream. When this occurrence is yielded, the evaluation thread for the behavior is killed and a new thread forked to evaluate the new behavior [2].

7 Conclusion

We have presented TimeFlies, a system for push-pull signal-function Functional Reactive Programming, and have shown how the use of a signal vectors as input and output types for signal functions, together with GADT-based representations of the inputs and outputs, permits the implementation of a push-pull system.

We have also described a general and flexible monadic evaluation interface for TimeFlies, which permits us to interface the TimeFlies system with different styles of IO systems, including multiple IO systems in the same application.

This opens up the exciting possibility that a signal-function FRP could become an efficient and general framework for writing interactive applications.

References

1. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of the second ACM SIGPLAN international conference on Functional programming. ICFP '97, New York, NY, USA, ACM (1997) 263–273
2. Elliott, C.: Push-pull functional reactive programming. In: Haskell Symposium. (2009)
3. Apfelmus, H.: reactive-banana library. http://hackage.haskell.org/package/reactive-banana
4. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: Proceedings of the 2001 Haskell Workshop. (2001) 41–69
5. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. Haskell '02, New York, NY, USA, ACM (2002) 51–64
6. Nilsson, H.: Dynamic optimization for functional reactive programming using generalized algebraic data types. In: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming. ICFP '05, New York, NY, USA, ACM (2005) 54–65
7. Schulthorpe, N.: Towards Safe and Efficient Functional Reactive Programming. PhD thesis, University of Nottingham, UK (2011)
8. Liu, H., Hudak, P.: Plugging a space leak with an arrow. Electronic Notes in Theoretical Computer Science **193**(0) (2007) 29–45
9. Cheney, J., Hinze, R.: First-class phantom types. Technical report, Cornell University (2003)
10. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. SIGPLAN Not. **38**(1) (January 2003) 224–235
11. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming. ICFP '06, New York, NY, USA, ACM (2006) 50–61
12. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '93, New York, NY, USA, ACM (1993) 71–84
13. Peyton Jones, S.: Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Hoare, T., Broy, M., Steinbrüggen, R., eds.: Engineering Theories of Software Construction. Volume 180 of NATO Science Series: Computer & Systems Sciences. IOS Press, Amsterdam, The Netherlands (2001) 47–96
14. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '95, New York, NY, USA, ACM (1995) 333–343

Parameterized Parsers

Kathryn E. Gray

University of Cambridge

Abstract. Modular parser implementations aid in the development and maintenance of parsers for experimental language design as well as families of programming languages. However, existing modular parsers do not support parameters that span multi-non-terminal subsets of a grammar as well as supporting small readable specifications, both of which are beneficial in supporting modular design principles. Combinator parsers provide a common style of modular specification, with readable grammars and local parameters, but they lack grammar-wide abstractions and modular recursive dependencies.
We combine combinator parsers with a component system that provides parameterization and recursive dependencies to develop modular parser specifications. We demonstrate the organization and benefits of this technique with a modular parser specification for a family of Java-like languages. To develop this specification, we implement a unit-based combinator parser library for Racket.

1 Why parameterize a parser?

Experimenting on and maintaining most programming languages requires modifying or extending a parser. In some circumstances, such as within various teaching environments (see Ruckert and Halpern (1993); Findler et al. (2002); Gray and Flatt (2003); Hsia et al. (2005)), a set of related programming languages with varying restrictions (a *family* of languages) may be supported simultaneously, requiring multiple parser specifications. These different specifications often share many grammar productions despite their differences.
Typical parser generators, such as Yacc-like generators, do not support specifying parsers for multiple different yet similar languages. These generators strictly connect a usage of a grammar term to its definition point; thus requiring either numerous different productions (with similar format) or requiring completely separated repetitious specifications. A more modular approach to this task reduces errors and frustration.
Some compiler-writing tools and parser generators, such as ASF+SDF (Klint 1993) and antlr (Parr 1997), provide support for modular parser specifications. Existing modular systems provide support for system-wide grammar parameters, as well as limited support for language-wide parameters. However, their mechanisms for providing this support obfuscate the grammar specification and reduce readability. Combinator parsers (Hutton 1992) provide another means for developing modular parsers, by

functionally abstracting a parser specification on the definition of subgrammars with natural grammar specifications. However, these specifications do not cleanly support modular specifications either where parameters might vary over sets of terms.
In implementing variations of a programming language, a designer may choose to extend subgrammars, to exclude subgrammars, or to modify the definition of one subgrammar. To illustrate, in our family of Java languages implemented for ProfessorJ Gray and Flatt (2003), one Java variant excludes `for` loops from the statement subgrammar while another includes only the `for` loop production that requires a termination condition. The rest of the grammar specification refers to the statement subgrammar several times, including within the method subgrammar, constructor subgrammar, and within different productions of the statement subgrammar itself. Modifying the statement grammar indirectly impacts all of these references and their specifications.
Implementing different statement grammars, while reusing independent specifications, following a combinator approach requires a significant function parameter overhead. The method grammar specification must reside in a function with a statement parameter. Additionally, the constructor grammar and each production of the statement grammar must follow the same pattern. Further, to allow modifications within other subgrammars, each of these definitions should be parameterized over any reference to a subgrammar.
When instantiating these grammars, each statement parameter for each of the functions is satisfied by the same value, referred to multiple times. This repetition complicates the development process for the parser and suggests the use of a more modular approach. However, utilizing the module systems supported by most programming languages does not provide sufficient recursive parametric modularity to create clear modular parser specifications. The unit system from Racket (Owens 2007; Culpepper et al. 2005; Flatt and Felleisen 1998) provides flexible module support that matches the needs of parameterized parsers.
Combining a combinator parser and a unit-based component system allows the creation of grammar specifications that parameterize over the different subgrammars while maintaining a readable grammar that resembles a standard context-free grammar. This provides another illustration of the benefits of component-based modularity, even combined with traditionally flexible functional implementations. We demonstrate the benefits of this approach by outlining a parser for Java parameterized to support different language configurations. The parser utilizes a combinator parser library we developed to support unit interactions and clear specifications, written in Racket.
Section 2 presents an overview of our parser interface, including the unit connections. Section 3 presents representative examples of the Java parser implementation. Section 4 presents details of the implementation.

2 Combinator parsing and units

Combinator parsers allow a parser author to specify a grammar using BNF notation while simultaneously employing functional abstrac-

tions. Although at least one other combinator implementation exists for Scheme (Garnock-Jones 2005), we require an implementation that allows unit-based parameterizations.
Our combinator parser library uses units to achieve clean modular specifications. This combination led to choices in the interface that add flexibility to library and parser implementors.

2.1 Specifying a parser

As with most combinator parser libraries, we supply forms for specifying sequential and optional grammars that generate appropriate parser functions. The specifications may be abstracted with standard function definitions etc. Specifications may also nest anonymously. Figure 1 includes a specification for the standard calculator parser, using our interface to define two dependent grammars with nested `sequence` and `choose` declarations.

```
(define mul-exp
  (choose [(sequence
              (num (choose (*op /op)) num)
                   (lambda (l op r) (op l r)))
            num]
          "multiplicative expression"))
(define expr
  (choose [(sequence
              (mul-exp (choose (+op -op)) mul-exp)
              (lambda (l op r) (op l r))
           (sequence (o-prn (eta expr) c-prn)
                     (lambda (dl exp ld) exp))
           mul-exp]
          "expression"))
```

Fig. 1. Calculator grammar

Both the `choose` and `sequence` forms expect a parenthesized listing of parser specifications, followed by production specific information, as seen in figure 2. The `choose` form allows an optional string argument. When present, this argument provides a name for the production, used in providing debugging information as well as identifying errors. The `sequence` form first requires a function before allowing an optional naming argument as well. The arity of the function must match the number of subparser specifications; the function will be applied to the same number of arguments when building the resulting parse tree. When a production specification does not have a supplied name, the form automatically generates a generic name. However, within nested specifications without name annotations, an outer-most name applies to all contained forms.

```
(choose (parser ...) ?string)

(sequence (parser ...n) arityn-func ?string)

(eta parser)

(repeat parser)
```

Fig. 2. Provided combinator forms

The `eta` application within the calculator example supports the immediate definition of recursive grammars. Recursive definitions that rely on unit parameterization do not require an `eta` expansion.
The signature of the generated parser, and the signature required for each function identified in the specification lists, follows

`(listof` α`)` $\rightarrow$ `(struct:res` β `(listof` α`) ...)`

The β value arises as the result of calling the provided build function for a sequence, or returns a value representing a parse error. A `choose` built parser returns a sub-structure of the general `res` structure, containing the same basic information as in a sequence result, with the addition of a list containing the possible parses of the input. The returned α-list represents the remaining input stream not yet consumed by the parser.
In addition to these syntactic forms, our interface also exposes the underlying function implementations `seq` and `choice` —so that parser authors may opt for greater flexibility as needed— as well as a `repeat` combinator generator. These three basic forms simplify the process of specifying grammars as well as further parser generators. Figure 3 demonstrates both the use of `repeat` as well as the definition of a new parser generator.

```
(define (comma-sep pat n)
  (sequence
     (pat (repeat (sequence (comma pat) second)))
     cons
     (string-append "comma separated list of " n)))
```

Fig. 3. Portion of an expression grammar

The `repeat` function expects a parser argument and generates a function that accepts any number of repetitions of the provided parser before terminating. The functional forms of `sequence` and `choose` accept a list of parsers as their first argument, along with the same parameters as the macro forms.
Although the repeat combinator can be built by calls to sequence and choice, we provide it directly due to its prevalent use. Parser authors

may extend the parser generators themselves, following similar patterns as the *comma-sep* specification.

Terminals and scannerless parsing In our example calculator parser, we refer to number and operator parsers that are not presented. These parsers accept terminals. Many combinator parsers support scannerless parsing, where individual characters are the terminals and combinations of sequence or choice parsers form groups of characters into words. Our system supports either scannerless parsing or parsing a stream of tokens generated by a lexer. Individual terminal specification controls the granularity of a parser to permit flexible implementations; however, we provide additional specification support for parsing a list of lexed tokens.

We provide a `terminal` function that expects three values used in creating the initial parser:

`(terminal (`$\alpha \rightarrow$ *bool*`) (`$\alpha \rightarrow \beta$`) string)`

The returned parser consumes one input item from the list when the first argument returns `#t` and the result is a *res* struct embedding the β returned by the second argument. The choice of predicate permits the parser author to manage the granularity of parsing, and even combine scannerless and tokenized parsing. However, we recognize that specifying terminals by defining each terminal function and writing individual predicates can be tedious for reasonable languages.

We elected to provide a shorthand for specifying terminals that correspond to tokens generated by the Racket parsing library's lexer, and have not provided any additional support for generating scannerless terminals. Since we expect that parsers generated by our system will interface with the DrRacket development environment, we believe that authors will write a lexer for their language to utilize the provided syntax-coloring support. A modular lexer specification that produces either the coloring information or parsing tokens reduces development effort.

The `define-simple-terminals` and `define-terminals` forms, seen in figure 4, bind lists of given names to specific invocations of the `terminal` function with the second and third arguments either specified by the author or resolved by context. Token-based predicates are always generated and provided to the `terminal` function, with the particular predicate based on the name of the terminal. When not provided by the programmer, the default building operation is the identity function, and the default name matches the name of the terminal.

```
(define-simple-terminals operators
  ((+op (lambda (x) +) "+") o-prn c-prn ...))
(define-terminals values
  ((string make-string-lit)
   (num string->number "number")))
```

Fig. 4. Terminal specification

A simple terminal definition denotes a set of terminals/tokens that match language keywords and therefore contain tokens that do not hold values. The other form specifies terminals whose content changes, such as identifiers and numbers. The programmer must supply a building function for these forms, which receives the value contained within the token. Equivalent token definitions are also generated by the *define-terminal* forms, as expansions to *define-tokens* or *define-empty-tokens* macros from the parser-tools collection. So the forms in figure 4 bind the names *+op*, *o-prn*, *c-prn*, *num*, *string*, *token-+op*, *token-o-prn*, *token-c-prn*, *token-num*, *token-string* within their context.

2.2 A Unitized interface

We provide the constructs described above to the programmer through a unit interface. Before describing the format of this interface, we provide a brief summary of unit constructs and technology.

A background on units Units within Racket (Flatt and Felleisen 1998) support recursive component programming using three basic language constructs – `unit`, `compound-unit`, and `signature`. A `unit` collects and selectively exports definitions jointly parameterized over a set of imported values. A `compound-unit` selectively connects imports and exports from different units and forms a new unit out of them. A `signature` specifies either the names of the imports or exports (depending on position) for a unit or a compound-unit. A unit that requires no imported values can be invoked, executing expressions within the unit and potentially opens the exported names into the exterior namespace.
A signature form can include macro specifications (Culpepper et al. 2005), which places the defined macros into the body of an importing unit where the specifications may rely on values from an exporting unit. Signatures support nominal inheritance, with contravariant subtyping on unit imports. Additionally, the signature form can be extended to support new specification forms (Owens 2007) so that programmers may easily import and export definitions stemming from macros that expand into definitions. As an additional benefit, a *compound-unit/infer* form automatically resolves all linking dependencies between units when their signatures provide sufficient information to do so unambiguously.

Units, signatures, and parsers Parser authors import a signature, *combinator-parser^*, that provides all of the functions and macros described previously. When their parser is complete, the author then links their parser with a unit providing full definitions for all of these forms.
Uses of the `define-terminals` forms produce a significant number of definitions. In many parser organizations, each terminal and token definition must be specified in the signature for export and import. The combinator implementation additionally provides a shorthand for specifying terminal definitions within signatures, for an example signature see figure 5. The *terminals* list must contain the names of all terminals

that should be externally visible, regardless of the inclusion of the group name. This allows programmers to limit the scope of terminals while still grouping them meaningfully.

```
(define-signature terminals^
  ((terminals operators (+op o-prn c-prn))
   string->number
   (terminals values (num string))))
```

Fig. 5. Signature form for specifying terminals

Augmentations to the core combinators can be contained within separate units and provided to programmers via signature inheritance. For example, the *comma-sep* implementation could require that the *comma* form be provided as a unit import, while the function is exported. If the *comma-sep* author chooses to extend the general functionality of the combinators, then the signature for the unit extends the *combinator-parser^* signature, adding the new operation. Other parser authors may then import either the augmented signature or the original. The final compound unit linkages can accept a unit containing the *comma-sep* implementation in either case.
Presenting the combinator-parser interface via units and signatures also permits greater flexibility in the implementation of the library as well as simpler per-parser parameterization. Future versions of the library can provide multiple implementations that parser authors can then select between without modifying their specifications. Additionally, the combinator library can rely on imports from the parser author to provide per-parser customizations without using system-wide parameters or additional arguments for each specification. This information includes facilities for source tracking, error message contexts, etc.

3 Configurable parsers made easy

Experimenting with programming language syntax may require extending or modifying portions of the language definition, while maintaining both the original language definition and intermediate stages to aid in comparisons. Also pedagogic projects, such as ProfessorJ (Gray and Flatt 2003) and DrJava (Hsia et al. 2005), support multiple related language implementations simultaneously, indefinitely.
Our experience with implementing and maintaining parsers for ProfessorJ demonstrated the benefits of a modular parser that shares related definitions across multiple languages while supporting a clean grammar specification, instead of a hand written parser. Different related languages may require similar structures with different component pieces, for example one language may support type names that include dots while another may support an expression form not present in the first. These differences require varying support for modular specifications.

We describe the benefits of our approach to modular parsing through the example of the parser specification of our Java parsers. Previously, we supported four Yacc-style parser specifications and a single hand-written parser to identify and report error messages. Due to the lack of abstraction in this system, maintenance and modifications caused replication of mistakes and only partial corrections, difficulties alleviated by implementing a modular parser with significant levels of reuse.

3.1 Simple specifications

Conceptually related grammar productions reside within the same unit definition. Related specifications typically rely on the same set of imports and appear in the same final language production. While the latter grouping is not necessary for ease of specification, the conceptual grouping aids in extension and in locating related definitions. Figure 6 presents two partial units containing related definitions with language-wide parameterized dependencies.

```
(define-unit methods@
  (import ...)
  (export ...)

  (define method-sig
    (sequence (mtype identifier method-parms)
              build-signature
              "method signature"))
)

(define-unit expressions@
  (import ...)
  (export ...)

  (define unary-assignment
    (choose
      ((sequence (++ expression) build-unary)
       (sequence (-- expression) build-unary))
      "unary modification"))

  (define if-end
    (sequence (? expression : expression)
              build-if
              "conditional expression")))
```

Fig. 6. Expression and Method grammars

The different units for the Java language represent the different language constructs including expressions, statements, methods, etc. We expect

that most language definitions would group different constructs similarly. In this example, the *methods@* unit imports a definition of types and identifiers to parameterize these definitions across the language. This parameterization allows different languages to restrict or expand the set of types within the language without explicitly modifying the definition of method signatures, which uses the subgrammar but does not change form based on the specific allowed types.
The *expressions@* definition unit contains a recursive reliance on the *expression* definition, to be outlined in section 3.3. Language representations may contain a subset of all the expression forms, or they may contain new forms not in the original language. Specific expression grammar forms rely on the definition of all expressions, as they contain expressions themselves. Indeed, most expression forms rely on the definition a single expression. Providing these recursive definitions through a unit parameter supports the reliance without restricting the language designer to a single definition for expressions.
Either of these units can be combined with additional units to extend the allowed set of grammar specifications without impacting language definitions that do not rely on the extensions. Similarly, a program specification may elect to exclude definitions appropriately.

3.2 Configurable grammars

Not all grammar specification abstractions require or allow language-wide parameterization. For these circumstances, traditional functional abstraction continues to provide the means of defining appropriate grammar specifications, as outlined in figure 7.
These definitions may support abstractions that do not come from language wide definitions and may complicate a unit interface if added as a parameter. The *return-s* definition outlines such a parameterization, where the full definition of a language form relies on a per-language choice but not a per-language definition.
Other definitions may include conditional implementations of portions of the language and so cannot rely on language-wide definitions. Both the *for-loop* and *interface-def* rely on definitions of expressions, methods, or fields that may vary from the definitions found in other portions of the language. The support for both unit and functional parameters lets the parser author choose the level of abstraction required without forcing the over parameterization of individual definitions.

3.3 Whole language parameterization

The definition of language-wide parameters occurs within the unit that defines the overall language. Each language specification resides in individual units and implements a uniform signature that the component definitions require. Figure 8 demonstrates signatures for the Java grammars, including the signature implemented by each full language specification.
The *expression@* unit in figure 9 imports the *expression* definition from the *language-forms^* signature, while the language specification

```
(define-unit statements@
  ...
  (define (return-s expr-req?)
    (let ([with-exp (sequence ...)]
          [without (sequence ...)])
      (if expr-req?
          (choose (with-exp without) ...)
          without)))

  (define (for-loop expr ...) ...)

)

(define-unit interface@
  ...

  (define (interface-def body)
     ...
     (sequence (interface identifier ... body)
               ...))
)
```

Fig. 7. Functionally parameterized definitions

```
(define-signature literals^
  ((terminals EmptyLits (true_lit false_lit))))
(define-signature language-forms^
  (expression statement field method ....))
(define-signtaure expressions^
  (unary-assignment if-end ...))
```

Fig. 8. Signatures

beginner-language@ unit imports the definition of different expression forms to use in the definition of an expression for the current language. This demonstrates the use of per-language parameterization in defining abstracted yet dependent definitions. The final resolution of this definition occurs with unit-linking.

```
(define-unit expressions@
  (import combinator-parser^
          language-forms^ literals^ ...)
  (export expressions^)

  ...)
(define-unit beginner-language@
  (import combinator-parser^
          experssions^ statements^ ...)
  (export language-forms^)
  (define expr-start
     (choose (all-literals unary-assignment ...)
             "expression"))
  (define expr-end
     (choose (if-end call-end ...)
             "expression end"))
  (define expression
     (sequence (expr-start (repeat expr-end))
               build-expr "expression")))
```

Fig. 9. Using units to parameterize definitions

The three-tier definition of the expression grammar presented supports traditionally left-recursive grammar definitions without creating infinite evaluations. Our parser implementation provides no automatic support for detecting left-recursion or lifting left-recursive definitions, relying on the programmer to detect and avoid these situations manually.

3.4 Forming a grammar

The final step in building a particular parser combines the different unit implementations into one. This step provides the final ability to differentiate language definitions with concrete implementations. Figure 10 presents the customization of a language implementation to specify the form of identifiers supported.
The final compound unit definition restricts the programming language to exclude Java's qualified names (i.e. no indirect imports). The linkage passes the specification of *name* as a flat identifier to all positions within the language, additionally the compound passes the definition of expressions found in *beginner-language@* into the expression definitions in the

```
(define-unit flat-names@
  (import ...)
  (export java-variables^)
  (define identifier ID)
  (define name ID))
(define-unit non-flat-names@
  (import ...)
  (export java-variables^)
  (define identifier ID)
  (define name ...) ;support id.id
)

(define-compound-unit/infer beginner-p@
  (import ...)
  (export ...)
  (link .... expressions@ beginner-language@
             flat-names@))
```

Fig. 10. Different unit implementations of identifiers

expressions@ unit and vice versa. Other language definitions may allow qualified names by including the *not-flat-names@* unit.
If all the unit imports are now satisfied, the programmer may invoke the compound unit and begin parsing the defined language. This specification style supports modular parsers with varying levels of parameterization without unnecessarily obfuscating the grammar specifications. Although the whole grammar cannot be viewed until during a debugging phase, the individual pieces are clearly identifiable and comprehensible, as well as the locations and definitions on which they depend.

4 Implementation

Embedding the implementation within units required isolation of the different aspects of the parser library to properly present the interface to programmers. This task paves the way for a more flexible, extendable implementation in the future.

4.1 Component structure

Our combinator parser resides within three units. One contains the definitions of the necessary functions in parsing the program, a second contains a bare-bones error message generator, and a third provides a function, *parser*, to build a front-end parser by extracting results or calling within the second unit to build an error message for failed parses.
This division allows development and expansion of different portions of the implementation independently. For example, a future implementation

of the parser generators may forbid ambiguous or left-recursive specifications (Grimm 2006), or incorporate a component that provides analysis of the resulting parse or partial parse to inform programmers of potential mistakes. Neither of these changes need impact the interactions with other unitized implementations.
The provided compound-unit, `combinator-parser@`, expects authors to supply a unit containing parameters setting variables for error reports, source-annotations, and similar information. These parameters allow authors to tweak the settings for a specific language specification or to use the same settings multiple times.

4.2 Implementing *seq* and *choice*

The two base functions implement the standard operations, using a packrat-style (Ford 2002) strategy to alleviate performance issues.

seq A returned `seq` function progressively calls each of the specified subparsers on the next available input. A successful subparser consumes at least one element from the source list, and the next subparser receives the remaining list. A failed `seq` parse either contains a failed subparser or the input list becomes empty before the subparser list. Regardless of the outcome, the function packages result information into a structure containing an abstract summary of the parse.
A subparser built using the `choice` operator may result in multiple correct parses of the input. When returning this result to a `sequence` built parser, the parser cannot eliminate any of the provided parses as the potential best fit for the current input. Therefore, the remaining subparsers are mapped over the list of returned parses. This list propagates through the parsing as long as it contains multiple correct parses.

choice A returned `choice` functions maps the provided list of subparsers over the initial input, wrapping the resulting list of possible parses in a structure. A specialized filter removes incorrect parses from this list. A parse fails only when all of the subparsers fail. The results of any subparser returning a list of possible parses itself are merged into the result of parsing the top form, passing all of the potential parses directly to the next level.

Using the parsers The `parser` function, referred to in the previous section, accepts a parser specification function and returns a function that provides either an error message or the result of parsing an input list. This function selects the first correctly parsed item when presented with multiple parses and returns an error when any input remains after an otherwise successful parse. Parser authors may find this function useful but can use the individual parser functions directly if they desire.

4.3 Modular parsers in other languages

While our implementation and design is only within the Racket language, the techniques and style of parser library could be applied to other languages. The features we rely on are higher-order functions, recursively

dependent components, and signatures with inheritance-like flexibility. While we benefit from macros to generate special forms for choice and eta, macros are not strictly necessary for the flexibility. Further, the library forms are tapeable in a reasonable static (polymorhpic) type system. Thus this style of parser implementation could be implemented in ML variants supporting recursive modules, namely that of Dreyer and Rossberg (Dreyer and Rossberg 2008) or of Russo (Russo 2001).
The transition to a Dreyer-Rossberg-style module system would require few changes to the system addressed above. The recursive module definitions can support definitions of grammar terms, such as expressions, that cross module boundaries in a similar fashion to the unit definitions. The flexibility of implementation provided by the signature system will support a compatible level of interchange as the nominally subtyped signatures in the unit model, so there would be a base signature provided by the library and users could extend and refine this signature to suit their needs. The embedding into Russo-style structures requires eta-expansions within the combining structure for recursive definitions like expressions, but the flexibility of signature specifications permits a similar level of abstraction in the composition of modules.
The types as given for the parser combinators above will fit into a standard ML type system implementation and are not themselves recursive or generative, so do not cause a problem for integration in a typed setting. If the AST-type specifications are defined along with the grammar, then the recursive type definitions may cause difficulty in either setting depending on their form. However, this is an orthogonal problem of modular language implementation in a typed setting.

5 Related work

Our technique for developing a parameterized parser builds on both the experience of implementing multiple parsers for variants of Java as well as the efforts of previous parser implementors at creating modular parser specifications. These prior efforts demonstrated both techniques for parser-wide modularization and clear specifications, although not together. Further, our efforts benefited from the advances in unit and macro technology developed by work on Racket Culpepper et al. (2005).

5.1 Combinator parsers

Combinator parsers occur frequently in Haskell, where lazy evaluation simplifies the implementation as well as recursive specifications (Hutton 1992). Parsec (Leijen and Meijer 2001) and Mimico (Camarao and Figueiredo 2001) are two prominent implementations with robust performance and monadic parser specifications.
Garnock-Jones (2005) developed a combinator parser library for Scheme that utilizes macros to encode the specifications and delay evaluation. Although the macros provide a more specialized syntax, the parser specifications closely resemble the Haskell-style monadic parsers.

In these combinator parsing systems, individual specifications do resemble a BNF grammar but providing language-wide parameterizations suffers, as each production must be individually parameterized over multiple non-terminals. Haskell's module system does not provide the necessary separation between interface and implementation to selectively modify the implementation of imported values. As Garnock-Jone's parser is implemented in Scheme, it could be extended to utilize units as we describe, at the price of interoperability with other Scheme implementations.
Both systems use a similar syntax for binding parsed values to variables; `id <- identifier` binds the result of parsing an identifier to `id` within the body of the specification. We opted for a system that uses functions and does not provide binding to avoid potentially obscuring the grammar specification with information related to the actions.
Parsing combinators for object-oriented languages, including Rats! (Grimm 2006) for Java and pyparsing (McGuire 2006) for Python, use classes to represent different productions. This combinator style does not resolve the parameterization problem for system wide-parameters as the initialization position has merely shifted into object creation.

5.2 Other configurable parsers

Parser generators that support modular specifications led us to consider separating portions of the specification that refer to a single position for parameters. The Asf+Sdf (Klint 1993) and Eli (Gray et al. 1992) support defining pieces of a language, including syntactic entities and associated actions, and then combining them into one language processor with different specifications referring to different productions. However the level of modularity is not as flexible as in combinator parsers.
Similar systems, IPG (Heering et al. 1989) and antlr (Parr 1997), supports the incremental generation of parsers by extending existing parsers. Further antlr provides local parameterization through object parameters. Neither provides system wide parameters. These systems demonstrated techniques in modularizing a parser specification, but did not fully satisfy our requirements and do not provide clear grammar specifications.

6 Conclusions

By using units to provide abstraction boundaries within our parser implementation, we simplify the process of writing a parser for four variants of one programming language with shared grammar specifications but customized productions. The resulting parser still contains recognizable language productions, benefiting future maintenance and reducing the need for correcting errors. Modifying a language in this system requires tweaking appropriate booleans and modifying the contents of the top-level language form specifications, reducing the effort of both creating new variant-languages and refactoring the contents of old variants.
Our parser library can be downloaded with the DrRacket environment, within the combinator-parser collection. The Java parser presented in section 3 can be found at www.professorJ.org.

Bibliography

Carlos Camarao and Luclia Figueiredo. A monadic combinator compiler compiler. In *Proc. of Brazilian Symposium on Programming Languages*, 2001.

Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In *Proc. ACM GPCE*, 2005.

Derek Dreyer and Andreas Rossberg. Mixin' up the ML module system. In *Proc. ACM ICFP*, 2008.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *JFP*, March 2002.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for hot languages. In *Proc. ACM PLDI*, 1998.

Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proc. ACM ICFP*, 2002.

Tony Garnock-Jones. *Portable Packrat Parser Library for Scheme*, 2005.

Kathryn E. Gray and Matthew Flatt. ProfessorJ: a gradual introduction to Java through language levels. In *OOPSLA Educator's Symposium*, October 2003.

Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, Vol. 35(2), 1992.

Robert Grimm. Better extensibility through modular syntax. In *Proc. ACM PLDI*, 2006.

J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. In *Proc. ACM PLDI*, 1989.

James I Hsia, Elspeth Simpson, Daniel Smith, and Robert Cartwright. Taming Java for the Classroom. In *Proc. SIGCSE*, 2005.

Graham Hutton. Higher-order functions for parsing. *JFP*, 2(3), July 1992.

P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, Vol. 2(2), 1993.

Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Utrecht University, 2001.

Paul McGuire. *Pyparsing introduction*, 2006.

Scott Owens. *Compile-time Information in Software Components*. PhD thesis, University of Utah, 2007.

Terence Parr. ANTLR parser generator and translator generator, 1997. http://www.antlr.org/.

Martin Ruckert and Richard Halpern. Educational C. In *Proc. SIGCSE*, 1993.

Claudio V. Russo. Recursive structures for standard ML. In *Proc. ACM ICFP*, 2001.

Skew Generic Test Data Generation

Pieter Koopman and Rinus Plasmeijer

Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, the Netherlands
{pieter, rinus}@cs.ru.nl

Draft

Abstract. Property based test tools perform tests based on properties stated in (first order) logic. The test tool generates test values for the universal quantified variables, executes the associated tests, and evaluates the test results. Property based testing enhances unit testing by moving to a higher abstraction level; instead of specifying individual instances of properties, the test engineer specifies the properties themselves. This moves the burden of generating relevant test cases to the test tool. In this paper we compare the advantages and disadvantages of various test data generation approaches. We show that a modified generic generation algorithm combines the best aspects of those approaches.

1 Introduction

With the introduction of property based test tools an important abstraction step is made. Instead of specifying individual instances of properties as test cases, the test engineer specifies the underlying properties themselves. This has three advantages. First, the tested properties are explicit instead of implicit. Second, it is much easier to maintain such a set of properties than to maintain a large set of individual unit tests. Third, it is very easy to execute more tests, by just changing a parameter the tool can execute more tests. The seamy side of these advantages is that one has to relay on the test system for selecting relevant test cases. In this paper we discuss various approaches to for this test data generation. We introduce a generic algorithm that combines the best properties of those approaches.

Examples of these tools in lazy functional languages are QuickCheck [6], SmallCheck [13], and EasyCheck [5] for Haskell, and G∀st [10] for Clean. This property based approach to testing is quite successful. Hence, this idea is implemented in many other languages, see `http://en.wikipedia.org/wiki/QuickCheck` for an overview. Today this incomplete list contains 22 languages and even more implementations. The capabilities of the host languages often impose restrictions on the ported test system. However, the main approach remains unchanged: the user specifies a universal quantified logical property and the test system tries to falsify this property by automatically generating test cases. Although this approach to testing clearly set a trend, the idea of automatic generation of test cases and test scripts itself is much older. See e.g. [3, 4, 7].

A benefit of the logical property based testing approach is that the burden of selecting test cases is transferred to the testing tool. The tool developers however, need to determine how these test cases are generated. Usually the test suite, the set of test values, is selected based on the *type* of the universally quantified variables in the logical property. When a property is falsified the test system has found an *issue*. Such an issue might indicate an error in the tested software. Also incorrect properties or invalid test values might cause the test system to report issues. Whether or not a property that does not hold is indeed falsified depends critical on the test suite generated. Hence, effective test suite generation is critical for effective testing.

We focus on generation by a *generic* algorithm, which has the advantage that it works for any data type. Instead of defining generation of each and every new type by a tailor made instance of a class, the generation for the new type can be *derived* by the compiler. When there are restrictions on the data type (like sorted threes or restricted values of parameters), it is still possible to specify the generation manually.

In this paper we review the advantages and disadvantages of systematic test data generation. The main drawback appears to be that large test values are generated too late in some circumstances. We show how the generic algorithm can be improved to cure those problems.

2 Test Data Generation Revisited

In this section we review the selection of the test suite used in the tests and how such test suites are typically generated. The test suite to be used is either indicated explicitly by the user, or deduced from the type of the variables in the property. Whether the test values are selected directly by the user or with help of the type system is not very relevant for this paper. We focus on how to generate effective test values.

Two approaches are used for the generation of test suites. Presumably the simplest approach is to generate the test data in some pseudo random order. The other option is to generate test values in some systematic order.

2.1 Selection of the Test Suite

The test suites to check a property can be indicated explicitly, or be deduced from the type of the variables in the property. In a strongly typed language, like Haskell or Clean, the type system can be used to select the appropriate test suite for a universally quantified variable. A typical example is the property propAbs for G∀st:

propAbs x = abs x $\geq$ 0

The type system of Clean deduces that x has type Int[1]. Hence G∀st will generate test values of type Int for this property. Testing this property is just evaluating the function propAbs for a large number of integer test values and checking whether it yields True for all tests.

For test systems embedded in a language without a strong type system, like QuviQ or Triq in Erlang, the user has to indicate the test suite in one way or another. A typical example is (see [2]):

```
prop_reverse () →
  ?FORALL({Xs,Ys},
          {list(int()),list(int())},
          lists:reverse(Xs++Ys) = lists:reverse(Ys) ++ lists:reverse(Xs)).
```

Here one indicates that the property

$$\forall xs, \forall ys\,.reverse(xs\ ++\ ys) = reverse(ys)\ ++\ reverse(xs)$$

should be tested with test suites of type $List\,Int$ for xs and ys.

Even in strongly typed language the user has to indicate the test suite every now and then: when the type system cannot solve the overloading in the property the has to indicate the exact type in order to allow the test system to choose a test suite. Consider for instance the same property of the reverse function stated in G∀st:

```
propReverse xs ys = reverse (xs++ys) = reverse ys ++ reverse xs
```

The type system will derive the polymorphic type [a] [a] → Bool for this property. This does not tell the type system what test suites to generate. When we specify a more specific type like [Int] [Int] → Bool it is clear for the test system that it has to generate list of integers as test cases. A more elegant solution is to keep the property itself polymorphic and make a type restricted version to determine using the type to be used in the tests:

```
propReverseInt :: ([Int] [Int] → Bool)
propReverseInt = propReverse
```

For the test suite generation it is largely irrelevant how the test suite is selected. Given that a test suite of a well determined type is selected, we focus on the strategies to generated the values within a test suite.

Even when the default test suite is determined by the type of the variables, it is often possible to deviate from the default generation algorithm. For instance, we can use list of the form [0..i] for increaasing i for the property propReverse with a property like:

```
propReverse2 :: Property
propReverse2 = (λxs.propReverse xs xs) For [[0..i] \\ i ← [1..]]
```

[1] In contract to Haskell, 0 in not an overloaded value in Clean, it has type Int. Hence it is clear what type of test values to generate. In Haskell 0 is an overloaded value with type (Num a) ⇒ a.

2.2 Pseudo Random Test Data Generation

Pseudo random generation is a simple approach to test data generation. QuickCheck and most of its ports rely on pseudo random generation of test values. When the number of test data generated is large enough, it is likely that a value falsifying an invalid property is encountered. Using the primitives provided by the test system the test engineer has to specify how the actual test data are generated for all user defined data types used in the properties. Pseudo random generation of test data appears to be rather effective; issues with many properties are found using this test data generation strategy.

The pseudo random approach to test suite has three drawbacks. First, it requires effort and experience to generate effective test data. Generating appropriate test data is often a nontrivial task. The basic problem is that the set of possible test cases is often very large. For recursive types, that are heavily used in functional programming, the set of possible test values is even infinite. The pseudo random generation of test case often needs guidance to find counterexamples in a reasonable amount of time. Second, due to the random generation test cases might be missed while others are unnecessary duplicated. Especially missing small test cases is annoying, one might expect that a Third, the counterexamples found are often not the minimal values that falsify the property. For analysis of such an *issue* it is convenient to have small test values falsifying the property. A technique called shrinking [9] is used to search systematically for smaller test data showing the same issue. Shrinking is somewhat similar to delta-debugging as introduced by Hildebrant and Zeller [8, 14].

The original Haskell QuickCheck and most of its ports rely on pseudo random generation of test values [6]. For primitive types that are part of the host language the generation is predefined in the test system. For user defined data types, the test engineer has to specify how the test data are generated using the primitives provided by the test system. Technically the user has to define an instance of the class `Arbitrary` in Haskell.

The rationale behind random test data generation is that it is generally not known where the counterexamples are in the input space. Pseudo random test data generation will encounter the counterexamples sooner or later if they exist.

This pseudo random approach to generate test suite has several drawbacks. The first drawback is that it requires effort and experience to generate effective test data. Generating appropriate test data is often a nontrivial task. The main problem is that the set of possible test cases is often very large. The pseudo random generation of test case often needs some guidance to find counterexamples in a reasonable amount of time.

The second drawback is that the counterexamples found are often not the minimal values that falsify the property. Larger counterexamples make it harder to analyze the source of the issue. In order to reduce this problem QuickCheck provides *shrinking*. This is an algorithm that produces a test set of smaller values based on the counterexample found [2, 9]. If the counterexample found is not the minimal counterexample, it is likely that shrinking finds a smaller counterexample for the same property.

A third drawback of pseudo random test data generation is that testing large test cases takes usually more time than small test cases.

Furthermore, random test generation can generate duplicated test cases. Due to the referential transparency of functional languages a duplicated test will always produce exactly the same result. Hence duplicated tests are a waste of effort and provide unjustified confidence in the system under test since the actual number of tests is lower than the number of tests executed. Worse, in pseudo random generation one does not detect that all possible tests are executed and hence that one has in fact proven the property by exhaustive testing.

2.3 Systematic Test Data Generation

Another approach for test suite generation is based on the observation that if a property is falsifiable there is almost always a small counterexample. This is explained by the fact that functions over recursive types are usually fail because of an incorrect or missing alternative. There are usually small function arguments that selects the erroneous alternative and hence falsify the correctness property. Based in this observation it is worthwhile to enumerate all small test data systematically. This prevents problems with missed small test cases and avoids unnecessary duplication of tests.

The test data generation of SmallCheck limits the nesting depth of test data in order to ensure that only small test values are generated [13], this also prevents the need for shrinking.

The generic generation strategy of G∀st generates test suites with values ordered from small to large [11]. In this generic generation algorithm the size of a test value is determined by the number of constructors and basic values it contains. The test data generation of SmallCheck limits the nesting depth of test data in order to ensure that small test values are generated. Although it is easy to construct functions that have no small counterexamples, the generation of test values from small to large appears to be very effective in practise. For, sufficient small, finite domains it is possible to detect that the test system has executed all possible tests. Hence, the property is *proven* by exhaustive testing. A drawback of the systematic generation approach is that it might be too busy with the systematic generation of small test cases. Large test cases might not be generated, and hence issues might be missed.

There are usually small function arguments that select the erroneous alternative and hence falsify the correctness property. Based in this observation the generic generation strategy of G∀st generates test suites with values ordered from small to large [11].

Systematic generation of test values instead of pseudo random generation also has the advantage that no duplicated test values are generated. For small finite domains it is possible to detect that the test system has done all possible tests and hence the property is *proven* by exhaustive testing rather than just passes the tests. Such a *proof* is really stronger than the usual *pass* yielded by the test system when no counterexamples are found.

Another advantage of the systematic generation of test cases from small to large is that it eliminates the need for a separate shrinking phase. A counterexample found is always a minimal counterexample. If there was a smaller counterexample it would have been generated and tested before.

The disadvantage of systematic generation is that it may take (too) long before a sufficiently large test case is generated to falsify the property. Although this problem can often be solved by increasing the number of tests and having a cup of coffee, we want to find counterexamples as quickly as possible using systematic test data generation. In this paper we show how we can improve the power of testing polymorphic functions by choosing an argument type of the right size. In addition the systematic generation algorithm for test cases can be improved without losing the advantages.

Claessen and Hughes have considered the possibility to generate test cases automatically. In [6] they explain why they do not use this: "*This is partly because we want QuickCheck to be a lightweight tool, easy to implement and easy to use in a standard programming environment; we don't want to oblige users to run their programs through a pre-processor between editing them and testing them. But another strong reason is that it seems to be very hard to construct a generator for a type, without knowing something about the desired distribution of test cases.*" In Clean we do not suffer from problems with a separate preprocessor for generic functions since the generic system is completely integrated in the language. The systematic generation algorithm from small to large values in G∀st works very well in most situations. It appears to be a good solution to the hard problem mentioned by Claessen and Hughes. If the generic generation does not behave as desired it is always possible to define the generation by hand instead of deriving the generic behaviour. Typically pure generic generation is not adequate if there are more constraints than expressed by the type system, like *search* trees where the type system only enforces the *tree* structure, or if one uses unnecessary large data types.

2.4 Mixing Systematic and Pseudo Random Test Data Generation

It seems attractive to combine pseudo random generation and systematic generation of test data in order to combine the advantages of both approaches. For effective testing we do not want to miss small test cases and to include sufficient large test cases as well. This is applied in EasyCheck [5] that uses a kind of systematic generation with pseudo random local jumps deeper into the recursion. Also G∀st uses a pseudo random shuffling of test values since the beginning [11]. There are some positive results reported for this combination of generation strategies.

The big problem is of course finding a balance between the systematic generation and the pseudo random generation. When the contribution of the pseudo random generation is small it remains likely to miss issues shown by large tests cases. On the other hand, when the contribution of the pseudo random generation is big we might use a larger number of large test data, but we also encounter the drawbacks of pseudo random generation.

In this paper we propose a systematic way to generate test data that has a user controllable preference for the recursive cases of the generated test data. In this way we try to achieve the best of both worlds. We have a systematic generation and large test values.

First we review the generic generation algorithm and show how it can be improved to favor the recursive cases in the generated data types.

3 The Generic Generation Algorithm Revisited

Despite these advances of systematic generic generation of test suites we have also encountered some points where it can be improved:

1. As mentioned above, it can take very long before a sufficient large test case is generated. This can be cured by using small data types in the test, or by using a tailored made generator instead of derive the generic algorithm. Nevertheless, it would be better if the generic algorithm generates large test cases earlier. The new version of the generic generation algorithm improves this significantly by giving higher preference to recursive branches in the data type. It is easy to tune the preference of branches if that would be necessary.
2. The pseudo randomness added to the generic algorithm does not really improve it. Although the generated values are an enumeration of the inhabitants of the type, their order had some pseudo random perturbation. Sometimes this distribution of the systematic order ensures that a counterexample is found somewhat earlier, but in an equal amount of situation it pushes the counterexample further to the future.
   Especially, the pseudo random number generation for integers appeared to perform disappointingly. For integers G∀st generates the well known border cases $0, 1, -1$, maxint and minint followed by pseudo random numbers. The border cases are very effective, but the pseudo random numbers do not really contribute to the counterexample finding capabilities of G∀st.
   The pseudo random changes in the order of elements used in G∀st uses a significant amount of resources. Each time the generation algorithm splits, the pseudo random generation is split also to prevent that we have to pass the random generator through all element generators. This appears to consume significant amounts of heap space if we generate large test suites.
3. The current system controls the amount of test done. Sometimes it is desirable to test a property for all test cases of some depth (e.g. all lists of length up to 3, or all trees up to depth 2), rather than the number of tests. In those situation an approach like SmallCheck that implements this would be desirable.
4. The current algorithm assumes an order of constructors in a recursive algebraic data type: the nonrecursive case (empty list or tree) is assumed to be the first constructor. It is better not to rely on this kind of assumptions and to determine the nonrecursive constructor with the smallest number of elements for the actual data type used.

To overcome the drawbacks and maintain the advantages of generic test case generation the generic algorithm is improved. First we remove the pseudo random distribution of the order of elements.

This results in the basic generic algorithm introduced in [11]. The core of this algorithm becomes:

```
generic gen a :: [a]

gen{|UNIT|} = [UNIT]
gen{|PAIR|} f g = [ PAIR a b \\ (a,b)←diag2 f g ]
gen{|EITHER|} f g = merge (map LEFT f) (map RIGHT g)
where
    merge :: [a] [a] → [a]
    merge [a:as] bs = [a:merge bs as]
    merge []     bs = bs
```

The prompt generation of large test cases is a bigger challenge. Recall that generic functions are defined on the 'sum-of-products' structure of types [1]. In Clean and G∀st this amounts to defining cases for EITHER (sum) and PAIR (product). There are two places in the generic algorithm where the order of elements can be controlled. The choice of elements in with the generic construct EITHER, and the combination of elements with the generic PAIR.

The case for EITHER determines the choice between the constructors of a data type. Although this seems to be the appropriate place to change the order of elements generated, it is not the best place. Consider a type like list with one nonrecursive constructor, Nil, and one recursive constructor, Cons a (List a). In its generic representation there is a single LEFT in the list generated by the case for EITHER, all other elements will start with a RIGHT. Moreover, we risk nontermination if the generation of the nonrecursive case is delayed too much: the generated Cons expression will need its argument, which in its turn needs its arguments etc.

As a consequence, the combination of values in the case for PAIR is the place to improve the algorithm. Here the order of combinations of heads and tails in the generated list is determined. The existing algorithm makes the combinations in a fair way by diagonalization of combinations. We change this to a skew diagonalization that gives an adjustable priority to the recursive cases. We have introduced a generation state, GenState, to keep track of the necessary information, like the deviation from straight diagonalization and the generic representation of the nonrecursive element in a data type, path. The state also counts the current depth and stops at a maxDepth, similar to SmallCheck. By default this depth is maxint such that the given maximum number of tests terminates the of testing properties.

```
:: GenState
  = { depth    :: !Int        // current object depth
    , maxDepth :: !Int        // max object depth
    , path     :: ![ConsPos]  // path to non recursive constructor
    , skewl    :: !Int        // skew Left factor, default 1
    , skewr    :: !Int        // skew Right factor, default 3
```

```
    }
```

The generic function ggen becomes:

```
generic ggen a :: GenState → [a]

ggen{|UNIT|}      s = [UNIT]
ggen{|PAIR|} f g s = diag s.skewl s.skewr (f s) (g s) PAIR
ggen{|EITHER|} f g s
    = case s.path of
        [ConsRight:_]   = merge (map RIGHT (g s')) (map LEFT  (f s'))
        _               = merge (map LEFT  (f s')) (map RIGHT (g s'))
where s' = { s & path = tl s.path }
      merge [x:xs] ys = [x: merge ys xs]
      merge []     ys = ys
ggen{|CONS|} f s = map CONS (f s)
ggen{|OBJECT of gtd|} f s
    = take (s.maxDepth - s.depth)
        (map OBJECT (f {s & depth = s.depth + 1, path = path}))
where path = hd ([getConsPath gcd \\ gcd ← sortBy argCount gtd.gtd_conses
                 | recCount gcd.gcd_type == 1 ] ++ [[]])
```

The definition of ggen for OBJECT inspects the generic representation of the type to select the nonrecursive constructors, recCount gcd.gcd_type == 1. If there are more of these constructor we select the one with the fewest argument count sortBy argCount.

The diagonalization algorithm takes two integer arguments that determine the number of elements taken to the left and right in each step. The local function skew has three lists as arguments: the elements to process on this diagonal, the elements processed, and the unprocessed elements.

```
diag :: !Int !Int [a] [b] (a b→ c) → [c]
diag skewl skewr as bs f = skew skewl [] [] [[f a b \\ a ← as] \\ b ← bs]
where
    skew :: Int [[a]] [[a]] [[a]] → [a]
    skew n [[a:as]:asl] bs cs = [a: if (n>1) (skew (n-1) [as:asl] bs cs)
                                             (skew skewl asl [as:bs] cs)]
    skew n [[]    :asl] bs cs = skew skewl asl bs cs
    skew n []           [] [] = []
    skew n []           bs cs = skew skewl (rev bs cs1) [] cs2
    where (cs1,cs2) = splitAt (max skewr 1) cs
```

Using this algorithm we can derive the generation of lists or user defined types. Using skew instead of diag2 (the symmetric diagonalization algorithm that comes with the Clean distribution) in the case for PAIR is very effective. For (skewl,skewr) equal to (1,1) the result of diag is equal to the result of diag2. The skew factor (1,3) is the default provided by the new version of G∀st.

4 Results

It is hard to measure the measure the effect of the skewness generic algorithm in a fair way. The effect depends hugely on the examples chosen and on the actual pseudo random values used for strategies with a random component. In general pseudo random generation is very effective for properties with very many large counter examples. Systematic generation is more effective in situations with a specific small counter example. We will demonstrate the effect of skew generic generation compared to the straight generic generation.

4.1 Length of Generated Lists

In order to show the effect of the skew data generation we measured the length distribution of generated list. As a reference we give the results for plain generation of test data of type `[Int]`. This is equal to a skew generation with factor `Skew 1 1`.

Number of lists generated	Length of generated lists					
	0	1	2	3	4	5
100	1	14	44	38	3	0
200	1	20	77	88	14	0
500	1	32	158	236	73	0
1000	1	45	271	481	202	0
2000	1	63	460	956	520	0
5000	1	100	927	2291	1616	65
10000	1	141	1563	4380	3625	290

Since there are very many integer values there are also very many lists of length one possible. In that sense test values of type `[Int]` are a realistic example, but also shows one of the worst possible behaviors with respect to the length of the generated lists. For list of Booleans, `[Bool]`, there are only two possible lists of length one, and four lists of length two. Hence the test system will generate much longer test data of that type.

Nevertheless, the maximum length of the generated lists is 3 for the first 2000 test cases and 4 for the first 1000 test cases. Given that the default number of test for a property is 1000, it is obvious that issues only revealed by somewhat longer lists are not found in this way.

The next table shows the influence of the skew factor for the default number of 1000 test cases.

	skew factor					
length	1 1	1 2	1 3	1 4	1 5	1 6
0	1	1	1	1	1	1
1	45	32	26	22	20	18
2	271	166	123	97	83	72
3	481	337	250	193	159	135
4	202	329	303	248	200	166
5	0	131	210	228	199	166
6	0	4	83	153	180	166
7	0	0	4	57	117	149
8	0	0	0	1	41	95
9	0	0	0	0	0	32

Obviously, the length of the generated lists increases when the skew factor increases. On the other hand there remains a desirable good coverage of short lists.

4.2 Example 1: Finite Queue Detection

As a simple example we consider a queue implementation with amortized $O(N)$ complexity. A desirable property of such a queue is:

- When we enqueue a list of elements and dequeue elements until the queue is empty we should obtain the same list of elements.

As property in G∀st this is:

```
pDequeueQueue :: [T] → Bool
pDequeueQueue l = allElements queue == l
where
    queue = foldl (flip enqueue) newQueue l
    allElements :: (Queue a) → [a]
    allElements q | isEmptyQ q
        = []
        = let (e,q2) = dequeue q in [e: allElements q2]
```

In our test we use a queue implementation with bounded length of 5. Although the implementation works correct as bounded queue, the property requires clearly an unbounded queue.

The number of test cases needed by G∀st to discover an issue is strongly dependent on the type chosen for T. The smaller the type, the quicker a counterexample is found. However, the general advice in model-based testing seems to be "chose a type with is large enough". In the table below we use T := Char.

skewness	1 1	1 2	1 3	1 4	1 5
test cases needed	218590	813	64	24	12

The number of test cases needed to find an issue for pDequeueQueue clearly drops very significantly when the skew factor increases. This has no negative

consequences for the size of counterexamples for other properties of these queues, nor for the number of tests needed to find them. So, introducing this skewness has a huge positive effect on testing properties of polymorphic functions.

Despite this success there are of course still counterexamples that will be missed by generic generation. If the bound of the bounded queue implementation is for instance 100 instead of 5, no issue will be found. Whether or not the queue can hold at least N elements can be checked with the single test:

```
Start = test (pDequeueQueue For [[1..N]])
```

The problem with this is that someone must realize that this is a desirable property for a queue. The advantage of the new generation algorithm is that the general generation algorithm covers this quality aspect for modest queue sizes with a general property on the correct behaviour of queues.

4.3 Example 2: A Parser and Evaluator for Expressions

Next we consider a parser for expressions that evaluates the parsed expression immediately, see [12] for details about the parser combinators.

```
expr :: Parser Char Int
expr
    = fact /?λ λe.((λo.(+) e) @> symbol '+' ⩓ expr λ!/
                   (λo.(−) e) @> symbol '-' ⩓ expr)
fact
    = term /?λ λe.((λo.(∗) e) @> symbol '*' ⩓ fact λ!/
                   (λo.(/) e) @> symbol '/' ⩓ fact)
term
    = num @> !+! pDigit λ!/
      symbol '-' />λ (¬) @> term λ!/
      symbol '(' />λ expr /<λ symbol ')'
```

In order to test this parser we generate test inputs by generating instances of the expression trees.

```
:: ExpTree = Op ExpTree Op ExpTree | Int Int
:: Op = Mul | Add | Sub

derive ggen ExpTree, Op
```

A simple property for model based testing tell that the value obtained by evaluating a tree should be equal to the value obtained by transforming the tree to an input for the parser and applying the parser:

```
propEx :: ExpTree → Bool
propEx tree = expr (toChars tree) == [(eval tree,[])]
```

The number of test cases needed to show the error in the parser again changes with the skewness of the generic data generation:

skewness	1 1	1 2	1 3	1 4	1 5
test cases needed	286	1278	162	212	262

In this example a skewness of 1 3 appears to be the optimal value. This value appears to work very well in other examples as well. For that reason we have chosen it as default value. We recommend to use other values only if there are very good reason to deviate from this default.

5 General Recursive Types

The algorithm introduced above is geared towards two argument constructors. It appears to work also for other data types, but not in an optimal way. For optimal generation the algorithm has to determine in which arguments the constructor is recursive and favor those positions in the generic generation. Determination of the recursive positions can be done based on the type information available in the generic representation.

6 Conclusion

In this paper we argue that systematic generation of test data for logical property based testing is preferable over pseudo random generation. The existing generic systematic generation had some drawbacks. The most significant problem was that in some situation huge amounts of tests cases were needed to falsify a property.

In this paper we show that this can be cured with a new generic systematic algorithm to enumerate the values in a, recursive, data type. The main difference is that we give a higher priority to the recursive cases in the systematic generation. Even a moderate skewness reduces the number of required test values with a huge factor. The advantages of systematic generation are not effected by skew instead of straight test data generation. In our examples this skew test data generation has no negative effects for other properties.

However, knowing the algorithm it should always be possible to design a special situation where any systematic generation order finds the counter example extremely late.

References

1. A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. In T. Arts and M. Mohnen, editors, *Selected Papers of the 13th International Workshop on the Implementation of Functional Languages, IFL '01, Stockholm, Sweden*, volume 2312 of *LNCS*, pages 168–186. Springer-Verlag, 2002.
2. T. Arts, L. M. Castro, and J. Hughes. Testing Erlang data types with Quviq Quickcheck. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, pages 1–8, New York, NY, USA, 2008. ACM.
3. M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. *SIGSOFT Softw. Eng. Notes*, 14:210–218, 1989.
4. D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22:229–245, September 1983.

5. J. Christiansen and S. Fischer. Easycheck: test data for free. In *Proceedings of the 9th international conference on Functional and logic programming*, FLOPS'08, pages 322–336. Springer-Verlag, 2008.
6. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th International Conference on Functional Programming, ICFP '00*, pages 268–279. Montreal, Canada, ACM Press, 2000.
7. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. Woodcock and P. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 1993. 10.1007/BFb0024651.
8. R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 135–145. ACM Press, 2000.
9. J. Hughes. Software testing with quickcheck. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *Central European Functional Programming School*, volume 6299 of *Lecture Notes in Computer Science*, pages 183–223. Springer, 2010.
10. P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: generic automated software testing. In R. Peña and T. Arts, editors, *Revised Selected Papers of the 14th International Workshop on the Implementation of Functional Languages, IFL '02*, volume 2670 of *LNCS*, pages 84–100. Springer-Verlag, 2003.
11. P. Koopman and R. Plasmeijer. Generic generation of elements of types. In *Proceedings of the 6th Symposium on Trends in Functional Programming, TFP '05*, pages 163–178, Tallin, Estonia, 23-24, Sept. 2005. Intellect Books. ISBN 978-1-84150-176-5.
12. P. W. M. Koopman and M. J. Plasmeijer. Efficient combinator parsers. In *Selected Papers from the 10th International Workshop on 10th International Workshop*, IFL '98, pages 120–136. Springer-Verlag, 1999.
13. C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy Smallcheck: automatic exhaustive testing for small values. *SIGPLAN Not.*, 44:37–48, 2008.
14. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.

Advances in Lazy SmallCheck

Efficient testing of higher-order properties with mixed quantification (Preliminary Version)

Jason S. Reich, Matthew Naylor and Colin Runciman

Department of Computer Science, University of York
{jason,mfn,colin}@cs.york.ac.uk

Abstract A property-based testing library enables users to perform lightweight verification of software. This paper presents improvements to the *Lazy SmallCheck* property-based testing library. Users can now test properties that quantify over *first-order functional values* and *nest universal and existential* quantifiers in properties. When a property fails, Lazy SmallCheck now accurately *expresses the partiality of the counter-example*. The necessary architectural changes to Lazy SmallCheck result in a *performance speed-up*. All of these improvements are demonstrated through several practical examples.

1 Introduction

Property-based testing is a lightweight approach to verification where expected or conjectured program properties are defined in the source programming language. For example, consider the following conjectured property that in Haskell all reductions on lists of Boolean values to a single Boolean value can be expressed as a *foldr*.

```
prop_ReduceFold :: ([Bool] → Bool) → Property
prop_ReduceFold r = exists $ λf z → forAll $ λxs → r xs ≡ foldr f z xs
```

When this property is tested using our advanced version of *Lazy SmallCheck*, a small counterexample is found for r.

```
>>> test prop_ReduceFold
...
Depth 6:
Var 0: { [] -> False
       ; _:[] -> False
       ; _:_:_ -> True }
```

Reading the output in the style of Haskell's case-expression syntax in explicit layout, this function tests for a multi-item list. Several new features of Lazy

SmallCheck are demonstrated by this example. First, note that two of the quantified variables, `r` and `f`, are *functional values*. Secondly, an *existential quantifier* is used in the property definition. Thirdly, the property involves *nesting of universal and existential quantifiers* inside the property. Finally, the counterexample found for `r` is *concise* and easy to understand.

Previous property-based testing libraries can struggle with such a property. QuickCheck (Claessen and Hughes, 2000) does not support existentials as it *'would rarely give useful information about an existential property: often there is a unique witness and it is most unlikely to be selected at random.'* (Runciman et al., 2008) QuickCheck also requires that functional values be wrapped in a *modifier* (Claessen, 2012) for shrinking and showing purposes.

The previous Lazy SmallCheck (Runciman et al., 2008) supports neither existentials, nested quantification nor functional values. SmallCheck (Runciman et al., 2008) supports all the necessary features of the property but it takes longwer to produce a more complicated looking counterexample. This is because SmallCheck enumerates only fully defined test values and shows them by systematically probing function responses to input.

Contributions This paper discusses the use and implementation of new features in Lazy SmallCheck. We present several contributions:

- A method of lazily generating and displaying *functional values*, enabling the testing of higher-order properties.
- An architecture and refutation algorithm that permits properties containing *nested quantifications* in a Lazy SmallCheck-style testing library.
- An evaluation of these additions with respect to *functionality and run-time performance*.

Roadmap The next sections give examples demonstrating the new features of the Lazy SmallCheck *(§2 and §3)*. The paper then focuses on the architectural changes that enable these new features *(§4)* and the formulation of functional values *(§5)*. Performance *(§6)* and features *(§7)* are evaluated and compared with other Haskell property-based testing libraries.

Note to reader: In a final version of this paper, further examples will be included to demonstrate the features of the library.

2 Functional values

Example: Left and right folds Let us look for a counterexample of another conjectured property. This property states that *foldl1* is extensionally equivalent to *foldr1*.

```
prop_foldlr1 :: (Peano → Peano → Peano) → [Peano] → Property
prop_foldlr1 f xs = (¬ ∘ null) xs ⟹ foldl1 f xs ≡ foldr1 f xs
```

As we shall be testing this with list elements from the user-defined data-type *Peano*, we shall need to create an *Argument* instance in order to produce functional values with *Peano* arguments. A *Template Haskell* function (Sheard and Peyton Jones, 2002) — *deriveArgument* — does this automatically.

```
data Peano = Zero | Succ Peano deriving (Eq, Ord, Show, Data, Typeable)
instance Serial Peano where series = cons0 Zero <|> cons1 Succ
deriveArgument "Peano
```

Lazy SmallCheck finds a counterexample at depth 6. The function *f* returns *Succ Zero* if its input is *Zero* and returns *Zero* in all other cases. The list *xs* is of length three where the last element is *Zero*.

```
>>> test prop_foldlr1
...
Depth 6:
Var 0: { _ -> { Zero -> Succ Zero
              ; Succ _ -> Zero } }
Var 1: _:_:Zero:[]
```

The underscore symbol has overloaded meaning. If it is on the left-hand side of a functional mapping, it indicates a *wildcard pattern*. Elsewhere, it represents *undefined input*.

Example: Generating predicates More complex patterns and partial functions can be represented, as shown in the next example related to *prop_PredicateStrings* from Claessen (2012).

```
prop_bitstring :: ([Bool] → Bool) → Property
prop_bitstring p  =  p [False, False, True,  False, False, True]
                  ∧  p [False, False, False, False, True,  True]
                  ⟹  p [False, False, False, False, False, True]
```

The counterexample is a function that returns *False* for all bitstrings that have *False* in their third and fifth positions, *True* for all functions that have *False* in their third but *True* in their fifth positions and *True* for all bitstrings that have *True* in their third position. Notice that the function is *undefined* for strings of length two or less and for strings that are of length 4 or less but have *False* in their third position.

```
>>> test prop_bitstring
...
Depth 14:
Var 0: { _:_:False:_:False:_ -> False
       ; _:_:False:_:True:_ -> True
       ; _:_:True:_ -> True }
```

Lazy SmallCheck has found the minimal definition of a function that falsifies the property. Wildcard patterns are used where the bit at that position simply does not matter. The function is undefined for the regions of the function space that do not affect the property.

3 Stronger properties

Example: Prefix of a list In the next example, taken from Runciman et al. (2008), we assert that a (flawed) definition of *isPrefix* satisfies a soundness specification of the function.

$$
\begin{aligned}
&\mathit{isPrefix} :: \mathit{Eq}\ a \Rightarrow [a] \to [a] \to \mathit{Bool}\\
&\mathit{isPrefix}\ [\,]\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = \mathit{True}\\
&\mathit{isPrefix}\ (x : xs)\ (y : ys) = x \equiv y \vee \mathit{isPrefix}\ xs\ ys\\
&\mathit{isPrefix}\ \_\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = \mathit{False}\\
&\mathit{prop\_isPrefixSound}\ xs\ ys = \mathit{isPrefix}\ (xs :: [\mathit{Peano}])\ ys \implies\\
&\quad (\mathit{existsDeeperBy}\ (*2)\ \$\ \lambda xs' \to xs +\!\!+ xs' \equiv ys)
\end{aligned}
$$

In Runciman et al. (2008), this property could only be checked using SmallCheck as Lazy SmallCheck did not support existential properties. Running it through Lazy SmallCheck gives another concise counterexample: the first argument of *isPrefix* is a multi-item list with first element *Zero*, and the second argument is [*Zero*]; *isPrefix* incorrectly returns *True*.

```
>>> depthCheck 3 prop_isPrefixSound
Var 0: Zero:_:_
Var 1: [Zero]
```

Given the recursive behaviour of *isPrefix*, a counterexample with both *xs* and *ys* non-empty shows that the error is likely in the second clause. Indeed, a disjunction has been used in place of a conjunction.

4 Under the hood

SmallCheck operates by exhaustively constructing all possible values of a particular type, bounded by the depth of construction (or some appropriate metric for non-algebraic types). *Lazy SmallCheck* extends this by first generating a partial

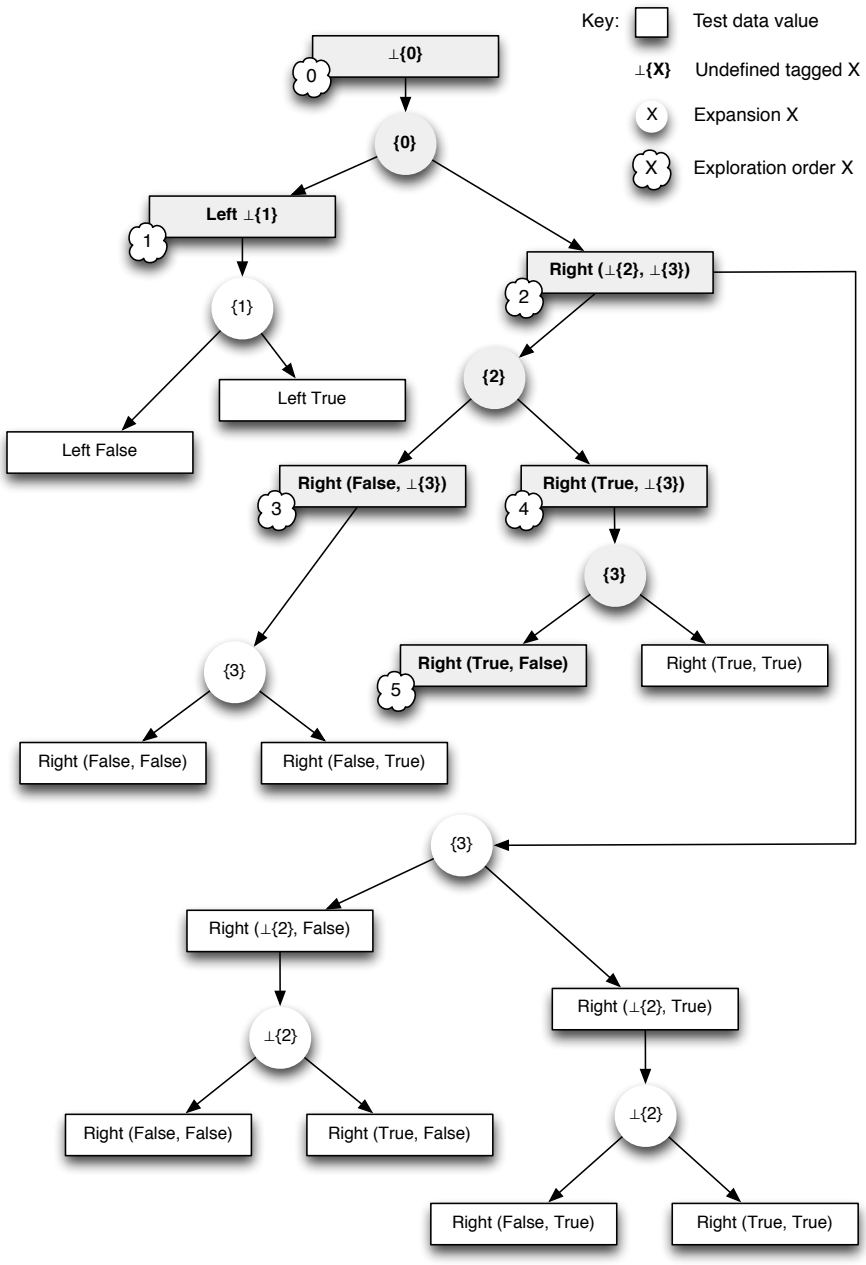


Figure 1. The Lazy SmallCheck counterexample search space for *prop_strange*.

value at each point in the structure and filling it only if a property function interrogates its value.

To illustrate, Figure 1 visualises the complete Lazy SmallCheck search space for test data of type *Either Bool* (*Bool*, *Bool*). Consider the following illustrative property, *prop_strange*.

$$\begin{aligned} &\mathit{prop\_strange} :: \mathit{Either\ Bool\ (Bool, Bool)} \to \mathit{Bool} \\ &\mathit{prop\_strange}\ (\mathit{Left}\ \ \_) \quad\ = \mathit{True} \\ &\mathit{prop\_strange}\ (\mathit{Right}\ (x, y)) = x \implies y \end{aligned}$$

Testing this property gives the following output, as one would expect.

```
>>> test prop_strange
...
Depth 1:
Var 0: Right (True,False)
```

Only the grey nodes are explored by the refutation algorithm. The thought bubbles in Figure 1 indicate the order in which the search space was explored to discover this 'counterexample'.

Notice that entire sections of the search space are never visited because either the property does not force the value (children of `Left _`) or because the order of evaluation explores those terms through a different route (second child of `Right _ _`). Only the term representing the counterexample is a total value.

What follows is a description of how the *new Lazy SmallCheck* achieves this process. Some simplification has been performed to ease reading: in particular, simpler but less efficient formulations are used in places.

4.1 Lazy SmallCheck terminology

In Lazy SmallCheck, a *Property* is a predicate written using *Bool* functions or in a domain-specific language that exposes Lazy SmallCheck features. If the free variables in a *Property* are of types belonging to the *Serial* type class, then they are instantiated by the refutation algorithm to find a counterexample.

Instances of the *Serial* type class define a constant *series* of type *Series a*, where *a* is the type of the test data being generated. *Series a* is a function that produces test data *Terms a*, for a given *Depth* bound.

The type *Term a* is a tuple consisting of a value (rectangle nodes in Figure 1) and list of expansions (children circle nodes in Figure 1). The refutation algorithm selects the appropriate expansions when a property forces an undefined region of a test-data value.

The previously mentioned *value* of a term is a composition of several types to provide the necessary partiality and contextual information. The following sections will describe those types and their combination.

4.2 Partial values

The original Lazy SmallCheck involved a property refutation function that existed in the *IO* monad. This was to make use of exception catching functions to detect partial values. The new Lazy SmallCheck takes some slightly different approaches and abstractions that enable the richer property language and the display of partial counterexamples.

LSC Exceptions The latest version of Lazy SmallCheck still uses exceptions to implement partial values, although it makes use of GHC's user-defined exceptions (Marlow, 2006) facility. These allow arbitrary types to be included in exceptions, rather than just strings. In any case, the library used by the original Lazy SmallCheck had long since been deprecated.

```
data LSC = Expand Location deriving (Show, Typeable)
instance Exception LSC
type Location = (Path, Nesting)
type Nesting = Int
type Path     = [Bool]
```

The expansion exceptions are tagged with a *Path* representing their location in the structure and the *Nesting* level of the quantifier.

Partial values monad To ensure that all Lazy SmallCheck exceptions are caught by the refutation algorithm, a monad of *Partial* values is defined.

```
newtype Partial e a = Partial {unsafePeek :: a}

instance Functor (Partial e) where
  fmap f = Partial ∘ f ∘ unsafePartial
instance Monad (Partial e) where
  return = Partial
  Partial xs >>= f = f xs

inject :: Exception e ⇒ e → Partial e
inject = Partial ∘ throw

runPartial :: (Exception e, NFData a) ⇒ Partial e a → Either e a
runPartial = unsafePerformIO ∘ try ∘ evaluate ∘ force ∘ unsafePeek
```

The intention is to ensure that we can perform any operation within the context of the *Partial* value monad but only *safely* retrieve the value with *runPartial*. The entire value is forced to make sure that an exceptions of type *e* are caught before they can escape outside the monad.

Showing partial values A *Show* instance is defined over *Partial* values which have *Data* and *Typeable* instances for the value type. The definition is omitted

here but it follows the *'Chasing Bottoms'* technique from Danielsson and Jansson (2004). This is what allows the display of *wildcard patterns* in the left-hand side of functional values and *partial values* in counterexamples.

4.3 Quantification contexts

The quantification context is simply a (co)monad container that carries information about quantifiers.

```
type QuantInfo = [String]
data QuantCtx a = QC {qcCtx :: QuantInfo, qcVal :: a}

instance Functor QuantCtx where
  fmap f (QC ctx val) = QC ctx (f val)
instance Applicative QuantCtx where
  pure = QC []
  QC ctx0 f <*> QC ctx1 x = QC (ctx0 ++ ctx1) (f x)
```

QuantInfo simply holds the pretty-printed representations of instantiated quantification variable values.

4.4 Test data terms

A *Term* is a pairing of possibly partial values with their possible expansions. The *tValue* component takes in a root *path* values and returns a possibly *Partial* value (containing exceptions of type *LSC*), wrapping in a *quantification context* (*QuantCtx*) holding pretty-printed representations of instantiated quantification variables. The *tExpand* component returns a list of *Term*s that are *expansions* at the path provided.

```
data Term a = Term {tValue  :: Location → QuantCtx (Partial LSC a)
                   ,tExpand :: Path     → [Term a]}

instance Functor Term where
  fmap f (Term v es) = Term ((fmap (fmap f)) ∘ v) (map (fmap f) ∘ es)
instance Applicative Term where
  pure x = Term (const $ pure $ pure x) (const [])
  f@(Term fv fes) <*> x@(Term xv xes) = Term
    (λloc      → (<*>) <$> fv loc <*> xv loc)
    (λ(p : ps) → if p then map (f <*>) (xes ps)
                      else map (<*> x) (fes ps))
```

4.5 Series and Serial generators

Generators of Lazy SmallCheck values are defined by the *Series* functor. Instances of *Functor*, *Applicative* and *Alternative* are provided such that the depth-bounding and partiality functionality is introduced and preserved.

```
type Depth = Int
newtype Series a = Series {runSeries :: Depth → [Term a]}
instance Applicative Series where
  return = Series ∘ const ∘ pure ∘ pure
  Series fs <*> Series xs = Series $ λd →
    [f x | d > 0, f ← fs d, let x = mergeTerm xs]
instance Functor Series where
  fmap f xs = pure f <*> xs
instance Alternative Series where
  empty = Series $ const []
  Series xs <|> Series ys = Series $ (++) <$> xs <*> ys
mergeTerm :: [Term a] → Term a
mergeTerm []  = error "LSC: Cannot merge empty terms."
mergeTerm [x] = x
mergeTerm xs  = Term (pure ∘ inject ∘ Expand) (const xs)
```

Using this interface, we can define series generators for types. For example, a series generator for *Peano* numerals could be represented as;

```
peanoSeries :: Series Peano
peanoSeries = pure Zero <|> (pure Succ <*> peanoSeries)
```

When instantiating quantification variables, the *QuantInfo* representation is stored. The instantiation is performed automatically for types satisfying the *Serial* type-class.

```
class Serial a where
  series :: Series a
  seriesWithCtx :: (Data a, Typeable a) ⇒ Series a
  seriesWithCtx = storeShow ∘ series
```

The *storeShow* value uses the *Partial* instance of *Show* to store a pretty-printed representation of each *Term*'s value in its quantification context.

Now that we have the *Serial* type-class, the $cons\chi$ family of combinators can be constructed as described by Runciman et al. (2008).

4.6 Properties

An internal *Property* domain-specific language supplies more information than simple Boolean properties. The *PAnd* constructor is parallel conjunction, as described in Runciman et al. (2008). The *ForAll* and *Exists* constructors represent *nested quantification* over some property, with some *depth modification*.

```
data Property = Lift Bool | Not Property | And Property Property
              | PAnd Property Property | Implies Property Property
              | ForAll (Depth → Depth) (Series Property)
              | Exists (Depth → Depth) (Series Property)
```

Smart constructors are provided for these, either automatically lifting *Bool*-typed expressions to *Property* or automatically instantiating free variables in properties with appropriate series from *Serial* instances.

4.7 Refuting properties

The algorithm for refuting properties (finding counterexamples) is defined using using auxiliary functions, making use of the *Partial* values library. At various points, exceptions are made explicit through the *runPartial* function.

Note to reader: In the final paper, a full description of the refutation algorithm will be supplied.

5 Implementing functional values

The key to generating functional values is the ability to represent the data-types of arguments as *tries*, also known as prefix trees. Lazy SmallCheck can then generate an appropriate trie and convert it into the requried function.

The previously described architectural changes enable the storage of a pretty-printed trie representation *before* the trie is converted into a Haskell function. This removes the need for a Claessen (2012) style *modifier*.

5.1 Custom data-types for functional value arguments

Users only need to define the *Argument* instance to allow custom data-types to be functional value arguments. The definition of *Argument* is related to the formulation of *HasTrie* instances by Elliott (2008). The instance defines a the *type of the trie structure*, an *application function* for looking up arguments in the trie structure and a *tabulation function* that converts the trie structure into a argument/value table ready for display.

A generic *Serial* instance for functional values uses the *Argument* instance as follows:

```
instance (Argument a, SerialF (Store a), Data a, Typeable a,
         , Serial b, Data b, Typeable b) ⇒ Serial (a → b) where
  series          = applyT <$> trieSeriesF series
  seriesWithCtx = applyT <$> storeShowTrie (trieSeriesF series)
```

We provide a generic *trie construction kit* for building these instances with $(\dot{+})$, $(\ast)$ and V representing *sums* of *product* types leading to output *values*. Using these types provides users with free *Serial* instances for the required tries.

Writing the *Argument* instances for most algebraic data types follows a regular pattern. We have therefore written a *Template Haskell* (Sheard and Peyton Jones, 2002) function to generate instance definitions on behalf of the user. In a previous example, the definition for *Peano* was automatically derived in this way. It produced a definition equivalent to;

```
instance Argument Peano where
  type Store Peano = V :+: Peano :*: V
  applyT' (z :+: s) Zero     = unV z
  applyT' (z :+: s) (Succ k) = unV $ (`applyT` k) $ unPro s
  tabulateT' (z :+: s) = return (pure Zero, unV z) `Branch`
                         (do (k, v) <- tabulateT $ unPro s
                             return (pure Succ <*> k, unV v))
```

5.2 Implementation highlights

While based on the Elliott (2008) trie implementation, our differs the following respects.

- Non-strict *'wildcard'* matching, where the argument is not interrogated.
- A standard *trie construction kit*, rather than producing a concrete trie type for each possible argument.
- The lack of a *functional-value-to-trie* function as it is unnecessary for first-order functional values.
- The inclusion of a *tabulation* function to expose a trie for display.

Note to reader: A fuller description of the implementation of tries for Lazy SmallCheck will be provided in the final paper.

6 Performance comparison

The performance of the new Lazy SmallCheck is compared to that of the original (Runciman et al., 2008), previously published. Experiments were performed using GHC 7.0.3 with `-O2` optimisation on a 2GHz dual-core PC with 4GB of RA. The original benchmark programs used by Runciman et al. (2008) have been run using the *Criterion* (O'Sullivan, 2011) benchmarking library to perform execution time measurements.

Table 1 shows the execution times. Run-time ratios less-than one show a performance improvement. A geometric mean of the ratios is calculated to indicate overall performance gains.

There is a performance increase in all but three benchmarks. Overall, there is a 147% speed-up compared with the previously published version of Lazy SmallCheck.

Note to reader: The final paper will have further analysis of why some benchmarks perform better than others.

7 Discussion and related work

A comparision of several Haskell property-based testing libraries can be found in Table 2. Test space exploration strategy is the main distinction between the QuickCheck library and SmallCheck family of libraries. QuickCheck assumes that test data detecting a failure is likely within some probability distribution. SmallCheck, on the other hand, appeals to the *Small Scope hypothesis* (Jackson, 2012) — programming errors are likely to appear for small test data.

Functional values The original QuickCheck paper (Claessen and Hughes, 2000) explains how functional test values can be generated through the *Arbitrary* instance of functions with a *Coarbitrary* instance of argument types. At this stage, QuickCheck could not display the failing example without bespoke use of the *whenFail* property combinator.

QuickCheck has since gained the ability not only to display functional counterexamples but also to reduce their complexity through *shrinking*. Claessen (2012) achieves this by transforming functions generated using the existing *Coarbitrary* technique into tries.

Claessen's formulation of tries differs from ours. Among its advantages is a simpler interface is presented that could be derived using a generics library, such as Scrap Your Boilerplate (Lämmel and Peyton Jones, 2003), rather than through Template Haskell meta-programming. However, Claessen (2012) requires that
functions are wrapped in a *'modifier'* at quantification binding. This *Fun* modifier retains information for showing and shrinking at the expense of a slightly more complex interface presented to users.

In Lazy SmallCheck, on the other hand, we directly generate a trie and then convert it into a Haskell function. A pretty-printed representation of the trie is stored at the time of generation and retrieved for counterexample output.

The SmallCheck representation of functional values uses a *coseries* approach, analogous to QuickCheck's *Coarbitrary*. However, functional values are displayed by systematically enumerating arguments.

Existential and nested quantification Runciman et al. (2008) discusses the lack of existential properties in QuickCheck. *"Testing a random sample of values as in QuickCheck would rarely give useful information about an existential property: often there is a unique witness and it is most unlikely to be selected at random."* (Runciman et al., 2008)

Unfortunately, the previous design of Lazy SmallCheck made it difficult to concieve of a refutation algorithm that could handle the nested quantification required to make existential properties useful. The use of the *Partial* values monad gives statically-typed guarantees that term expansions would be performed at the correct quantifier nesting.

Table 1. Comparative execution time performance.

Benchmark	Original	New	Ratio	Benchmark	Original	New	Ratio
Catch	2.37s	0.67s	0.28	Huffman2	0.24s	0.14s	0.59
Circuits1	4.85s	4.84s	1.00	ListSet1	0.05s	0.04s	0.80
Circuits2	0.02s	0.02s	1.04	Mate	0.13s	0.08s	0.60
Circuits3	4.45s	2.48s	0.56	RedBlack[a]	0.06s	0.04s	0.66
Countdown1	0.89s	0.49s	0.55	SumPuz	1.42s	1.37s	0.97
Countdown2	0.59s	0.60s	1.01	Turner	0.18s	0.11s	0.62
Huffman1	1.04s	0.69s	0.67		*Geometric mean*		*0.68*

[a] Counterexample found.

Table 2. Comparision of property-based testing library features.

Feature	QuickCheck	SmallCheck	Original LSC	New LSC
Test strategy	Random	Bounded exhaustive	Bounded exhaustive	Bounded exhaustive
Test space pruning	N/A	N/A	Lazy test-data generation	Lazy test-data generation
Minimal counterexamples	Shrinking	Natural	Natural	Natural
Functional values	Yes[a]	Yes	No	Yes
Existentials	No	Yes	No	Yes
Nested quantification	Yes	Yes	No	Yes
Displays partial counterexamples	N/A	N/A	No	Yes
Haskell 98/2010	Partial[b]	Compatible	Compatible	No[c]

[a] Functional value is wrapped in a modifier at its quantification binding if showing or shrinking is required.
[b] Originally Haskell 98 compatible but functional values modifier requires GADTs.
[c] Requires Haskell extentions such as type families, flexible instances, flexible contexts and Template Haskell.

As highlighted by Runciman et al. (2008), occasionally the depth of the existential quantification space needs to be larger than the surrounding universal quantification space. The *existsDeeperBy* combinator provides this functionality.

Benefits of laziness Runciman et al. (2008) discussed the benefits and fragility of exploiting the laziness of the host language to prune the test-data search space. When applied to functional values, we see further benefits. The partiality of underlying trie representation of functions corresponds directly with the partiality of the resulting function. Whereas Claessen (2012) needs to shrink total function to partial functions, the latest Lazy SmallCheck has partial functions as a natural result of its construction.

Runciman et al. (2008) notes that Lazy SmallCheck gets best results under universal quantification when the antecedent of an implication does not need to force the entire structure. Under existential quantification, it is hard to conceive of a similar design pattern.

8 Conclusions and further work

This paper has described the extension of Lazy SmallCheck with several new features; *(1)* quantification over functional values, *(2)* existential and nested quantification in properties and *(3)* the display of partial counterexamples. These features required some architectural changes to Lazy SmallCheck giving an improvement in execution time.

Further investigation is needed to investigate if the trie representation used by Claessen (2012) can be adapted for our purposes. This is an appealing prospect as trie implementations could be shared between the libraries and the Claessen style tries do not necessarily require Template Haskell to be automatically derived.

We have not discussed the handling of primitive types (e.g. *Int* and *Char*) as functional value arguments. Our approach has been quite similar to Claessen (2012) — reducing them to a pseudo-inductive definition. However, showing partially generated values of these types is difficult. Further work may resolve this omission.

Properties that quantify over functional values occur often in higher-order functional programming. Similarly, many properties may involve existential quantification and even nesting of quantification within property definitions. The examples in this paper have demonstrated the power of a tool that can find counterexamples for such properties.

As noted in the discussion section and in previous research (Runciman et al., 2008), the exploitation of laziness can effectively prune the test-data search space. This, combined with the performance improvements presented in this paper, makes finding counterexamples in a large search space attainable.

Acknowledgements We would like to acknowledge an e-mail suggestion from Max Bolingbroke that suggests using Elliott's (2008) *MemoTrie* library to implement functional values. The specifics of our implementation are quite different, however. The authors would also like to thank Michael Banks for his help with the paper.

This research was supported, in part, by the EPSRC through the Large-Scale Complex IT Systems project, EP/F001096/1.

Bibliography

Koen Claessen. Shrinking and showing functions. In *Proceedings of the fifth ACM SIGPLAN symposium on Haskell*, Haskell '12. ACM, 2012. Draft version.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279. ACM, 2000.

Nils Danielsson and Patrik Jansson. Chasing bottoms. In Dexter Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 85–109. Springer, 2004.

Conal Elliott. Elegant memoization with functional memo tries. Date accessed: 26th July 2012, URL: `http://conal.net/blog/posts/elegant-memoization-with-functional-memo-tries`, October 2008.

Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, Revised edition, 2012.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37. ACM, 2003.

Simon Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pages 96–106. ACM, 2006.

Bryan O'Sullivan. The criterion package, v0.5.1.1. URL: `http://hackage.haskell.org/package/criterion`, 2011.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 37–48. ACM, 2008.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16. ACM, 2002.

Functional Proxy Programming or Tinker Tailor Soldier Spy

David Wakeling

University of Gloucestershire
Cheltenham, Gloucestershire, United Kingdom
Tel: +44 1242 714267, Fax: +44 1242 714111
Email: dwakeling@glos.ac.uk

Abstract. Thanks to mission creep, network proxies, or simply proxies, are often rather large, somewhat disorganized pieces of software. Motivated by a concern for security, we have recently been working to bring the power of functional programming to bear on the problem of constructing proxies. In this paper, we construct a proxy for the Hypertext Transfer Protocol to illustrate one nice result of our work: a large, complicated proxy may be far more clearly expressed as a transparent one with many small, simple controls, each one an executable specification.

Keywords: functional programming, proxy programming.

1 Introduction

Nowhere are the dangers of mission creep more apparent than in the construction of *network proxies*, or simply *proxies*, whose duties may have grown haphazardly to include anonymization, authentication, filtering, screening, bridging, tunneling, caching, accounting, logging, monitoring, and more besides [1]. Look at a proxy nowadays and you will often see a rather large, somewhat disorganized piece of software[1]. This matters because proxies have an increasingly important part to play in ensuring our security [2]. Motivated by a concern for security, we have recently been working to bring the power of functional programming to bear on the problem of constructing proxies. Here, we illustrate one nice result of our work: *a large, complicated proxy may be far more clearly expressed as a transparent one with many small, simple controls, each one an executable specification*.

This paper is organized as follows. Section 2 introduces clients, servers and proxies. Section 3 introduces basic and composite controls. Section 4 considers an example proxy. Section 5 considers some example controls. Section 6 describes a

[1] The reader is invited to inspect, for example, the source code of `squid`, version 3.1.19, (76,401 non-blank lines), or even that of `tinyproxy`, version 1.8.3, (7,004 non-blank lines).

proxy test bench. Section 7 reviews some closely related work. Section 8 suggests some possible future work. Section 9 concludes.

Throughout, we use the Haskell programming language [3] — indeed, this paper is a literate Haskell script — and assume some familiarity with it. Those without such familiarity may sometimes need to consult one of the many excellent introductory textbooks [4–7].

2 Clients, Servers and Proxies

2.1 Clients and Servers

A *server* receives a request message of type a from a *client* over a network, and transmits a response message of type b back to it

```
type Server a b
  = a → IO b
```

It is convenient to have a class for message types, equipped with operations to transmit and receive message values on network handles

```
class Message m where
  transmit :: Handle → m → IO ()
  receive  :: Handle → IO m
```

and further convenient to have classes for request and response message types, equipped (for this work at least) with an operation to get request message destinations as (host, port) pairs

```
class Message a ⇒ Request a where
  destination :: a → (String, Int)


class Message b ⇒ Response b where
    -- nothing
```

2.2 Proxies

A *proxy* stands in for a server

```
type Proxy a b
  = Server a b
```

A *transparent proxy* relays a request, x, from a client, and a response, y, back to it

```
transparentProxy :: (Request a, Response b) ⇒ Proxy a b
transparentProxy x
```

```
= do let (hst, prt) = destination x
       hdl ← connectTo hst (PortNumber (fromIntegral prt))
       transmit hdl x
       y ← receive hdl
       hFlush hdl
       hClose hdl
       return y
```

3 Basic and Composite Controls

3.1 Basic Controls

A *basic control*, c, adapts a proxy, p, so as to improve its security[2]. See Fig. 1. In other words

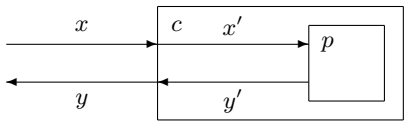


Fig. 1. A basic control.

```
type Control a b
  = Proxy a b → Proxy a b
```

3.2 Tinker Controls

One basic control adapts a proxy by replacing its requests; that is, it *tinkers* with it. A replacement request is obtained by applying a function to a request

```
tinker :: (Request a, Response b) ⇒ (a → a) → Control a b
tinker f pxy
  = λx → do y ← pxy (f x)
            return y
```

[2] A *control* is a means of managing risk, including policies, procedures, guidelines, practices or organizational structures, which can be of an administrative, technical, management or legal nature [2].

3.3 Tailor Controls

Another basic control adapts a proxy by replacing its responses; that is, it *tailors* it. A replacement response is obtained by applying a function to a request and response

$$
\begin{aligned}
&\mathit{tailor} :: (\mathit{Request}\ a, \mathit{Response}\ b) \Rightarrow (a \to b \to b) \to \mathit{Control}\ a\ b \\
&\mathit{tailor}\ f\ \mathit{pxy} \\
&\quad = \lambda x \to \mathbf{do}\ y \leftarrow \mathit{pxy}\ x \\
&\qquad\qquad\ \ \mathit{return}\ (f\ x\ y)
\end{aligned}
$$

3.4 Soldier Controls

Yet another basic control adapts a proxy by partially replacing it; that is, it *soldiers* for it. A replacement is made by applying a function to a request and a proxy

$$
\begin{aligned}
&\mathit{soldier} :: (\mathit{Request}\ a, \mathit{Response}\ b) \Rightarrow (a \to \mathit{Proxy}\ a\ b \to \mathit{IO}\ b) \to \mathit{Control}\ a\ b \\
&\mathit{soldier}\ f\ \mathit{pxy} \\
&\quad = \lambda x \to \mathbf{do}\ y \leftarrow f\ x\ \mathit{pxy} \\
&\qquad\qquad\ \ \mathit{return}\ y
\end{aligned}
$$

3.5 Spy Controls

A final basic control adapts a proxy by recording it; that is, it *spies* on it. A recording is made by applying a function to a request and response

$$
\begin{aligned}
&\mathit{spy} :: (\mathit{Request}\ a, \mathit{Response}\ b) \Rightarrow (a \to b \to \mathit{IO}\ ()) \to \mathit{Control}\ a\ b \\
&\mathit{spy}\ f\ \mathit{pxy} \\
&\quad = \lambda x \to \mathbf{do}\ y \leftarrow \mathit{pxy}\ x \\
&\qquad\qquad\ \ f\ x\ y \\
&\qquad\qquad\ \ \mathit{return}\ y
\end{aligned}
$$

3.6 Composite Controls

A *composite control*, $c_0 \circ c_1$, also adapts a proxy, p, so as to improve its security. See Fig. 2. In other words

$$(\circ) :: \mathit{Control}\ a\ b \to \mathit{Control}\ a\ b \to \mathit{Control}\ a\ b$$

4 An Example Proxy

As an example proxy, we consider one for the *Hypertext Transfer Protocol* (*HTTP*) [1].

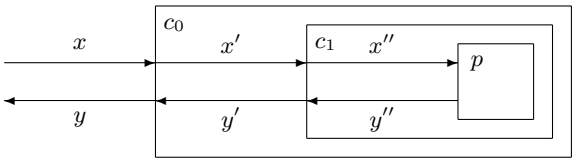


Fig. 2. A composite control.

4.1 HTTP Requests

An *HTTP request*, or simply a *request*, consists of a start line, some header lines, and a body. The start line consists of a method string, a Universal Resource Identifier (URI) string, and a version string. Each header line consist of a name string and a value string, separated by a ':'. The body consists of either a single content string or several chunk strings

```
data HttpRequest
  = HttpRequest
      (String, String, String) [(String, String)] (Either String [String])
```

An *HttpRequest* is a *Request*

```
instance Request HttpRequest where
  destination = destinationHttpRequest

instance Message HttpRequest where
  transmit = transmitHttpRequest
  receive  = receiveHttpRequest
```

although to save space, we omit the functions that make it so.

4.2 HTTP Responses

An *HTTP response*, or simply a *response*, also consists of a start line, some header lines, and a body. The start line consists of a version string, a status code string, and a reason string. The header lines and the body are as for requests

```
data HttpResponse
  = HttpResponse
      (String, String, String) [(String, String)] (Either String [String])
```

An *HttpResponse* is a *Response*

```
instance Response HttpResponse where

instance Message HttpResponse where
  transmit = transmitHttpResponse
  receive  = receiveHttpResponse
```

although again to save space, we omit the functions that make it so.

4.3 An HTTP Proxy

An *HTTP proxy*, or simply a *proxy*, is a specialized transparent proxy

```
httpProxy :: Proxy HttpRequest HttpResponse
httpProxy
  = transparentProxy
```

5 Example Controls

As example controls, we consider some for our example proxy.

5.1 An Anonymization Control

Some users prefer *anonymity*. Let us assume — with good reason [8] — that the extraordinarily precise version numbers found in request `user-agent` header values decrease user anonymity. Two typical `user-agent` header values are

```
chrome19 :: String
chrome19
  = "Mozilla/5.0 (X11; Linux i686) "
  ++ "AppleWebKit/535.21 (KHTML, like Gecko) "
  ++ "Chrome/19.0.1041.0 Safari/535.21"
```

and

```
safari533 :: String
safari533
  = "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_6; en-gb) "
  ++ "AppleWebKit/533.20.25 (KHTML, like Gecko) "
  ++ "Version/5.0.4 Safari/533.20.27"
```

A request is *anonymized* by mapping many `user-agent` headers to a few valid, appropriate, generic ones

```
anonymize :: HttpRequest → HttpRequest
anonymize (HttpRequest sl hdrs bdy)
  = case partition ((≡) "user-agent" ∘ fst) hdrs of
      ([(n, v)], nvs) → HttpRequest sl ((n, generic v) : nvs) bdy
      other           → HttpRequest sl hdrs bdy


generic :: String → String
generic v
    |  any (isPrefixOf "Chrome/19") ws = chrome19
    |  any (isPrefixOf "Safari/533") ws = safari533
```

```
    | otherwise                    = v
  where
    ws    = words v
```

A tinker control for anonymization is then initialized as follows

```
initHttpAnonymize :: IO (Control HttpRequest HttpResponse)
initHttpAnonymize
  = return (tinker anonymize)
```

In general, initializations may make reference to the environment, and especially to the file system, and so we give them an *IO* type. This control replaces all requests.

5.2 An Authentication Control

Some administrators permit only *authenticated* users. The credentials used for authentication might consist of colon-separated usernames and passwords, such as "`smiley:letmein`", that have been hashed and formatted in a standard way, such as "`Basic c21pbGV5OmxldG1laW4=`". Let us assume that a list of valid credentials like this may be loaded from the file system using a function

```
loadList :: FilePath → IO [String]
```

A request is then *authenticated* if any credentials in its headers are the same as those on such a list

```
authenticated :: [String] → HttpRequest → Bool
authenticated crds (HttpRequest (mth, uri, ver) hdrs bdy)
  = case lookup "proxy-authorization" hdrs of
      Just crd → crd ∈ crds
      Nothing → False
```

A tailor control for authentication is then initialized as follows

```
initHttpAuthenticate :: IO (Control HttpRequest HttpResponse)
initHttpAuthenticate
  = do crds ← loadList "credentials"
       return (tailor (λx y → if authenticated crds x then y
                              else demand))
```

This control replaces the response corresponding to a request that is not authenticated with an authentication demand

```
demand :: HttpResponse
demand
  = HttpResponse
      ("HTTP/1.1","407","Proxy Authentication Required")
      [("proxy-authenticate","Basic realm=\"The Circus\"")]
      (Left "")
```

5.3 A Filtering Control

One way to block undesirable content is to filter certain addresses according to a *blacklist* [2]. A blacklist might consist of a number of authorities, such as "`www.facebook.com`". Let us assume that a blacklist may be loaded from the file system using *loadList* as before. A request is *blacklisted* if its URI, such as "`http://www.facebook.com`", has an authority that is on a blacklist

```
blacklisted :: [String] → HttpRequest → Bool
blacklisted atys (HttpRequest (mth, uri, ver) hdrs bdy)
  = let (scm, aty, pth) = schemeAuthorityPath uri in
    aty ∈ atys


schemeAuthorityPath :: String → (String, String, String)
schemeAuthorityPath uri
  = case break (λx → x ≡ ':') uri of
      (scm, ':' : '/' : '/' : ts) →
        case break (λx → x ≡ '/' ∨ x ≡ '?') ts of
          (ath, "")  → (scm, ath, "/")
          (ath, pth) → (scm, ath, pth)
```

A tailor control for filtering is then initialized as follows

```
initHttpFilter :: IO (Control HttpRequest HttpResponse)
initHttpFilter
  = do atys ← loadList "blacklist"
       return (tailor (λx y → if blacklisted atys x then blocked
                              else y))
```

This control replaces the response corresponding to a request that is blacklisted with an innocuous one

```
blocked :: HttpResponse
blocked
  = let msg = "Blocked" in
    HttpResponse
      ("HTTP/1.1", "200", "OK")
      [("Content-length", show (length msg))
      ,("Content-type", "text/plain")
      ]
      (Left msg)
```

5.4 A Screening Control

Another way to block undesirable content is to screen it for certain *signatures*, and then perhaps to *quarantine* it [2]. The signatures might be represented by

a *Bloom filter* [9] — a compact set representation with an efficient probabilistic membership test [10]. Let us assume that membership of a Bloom filter may be tested using a function[3]

```
elemBloom :: String → Bloom → Bool
```

and that a Bloom filter of signatures may be loaded from the file system using a function

```
loadBloom :: FilePath → IO Bloom
```

A response is considered to be *infected* if a scan of its content, or of its concatenated chunks, finds any window that is a member of such a Bloom filter

```
infected :: Bloom → HttpResponse → Bool
infected blm (HttpResponse (ver, cde, rsn) hdrs bdy)
  = either (scan blm) (scan blm ∘ concat) bdy


scan :: Bloom → String → Bool
scan blm cs
  = any (λw → elemBloom w blm) (windows cs)
```

where a window is chosen to be six characters

```
windows :: [a] → [[a]]
windows []
  = []
windows cs
  = take 6 cs : windows (tail cs)
```

A tailor control for screening is then initialized as follows

```
initHttpScreen :: IO (Control HttpRequest HttpResponse)
initHttpScreen
  = do blm ← loadBloom "signatures"
       return (tailor (λx y → if infected blm y then quarantine y
                              else y))
```

This control replaces a response that is infected by a similar one that is marked with a cautionary header

```
quarantine :: HttpResponse → HttpResponse
quarantine (HttpResponse sl hdrs bdy)
  = HttpResponse sl (("quarantine", "yes") : hdrs) bdy
```

[3] For a full Bloom filter implementation, see [11].

5.5 A Caching Control

A *cache* is often used to avoid repeated communication with a server [1]. Let us assume that a cache is represented by an association list of request URIs and corresponding responses. If a request URI is successfully looked up in such a list, then the response should be taken from there; otherwise, the request should be transmitted to the server, and the request URI and the response received from the server should be added to the list

```
cache :: IORef [(String, HttpResponse)]
      → HttpRequest
      → Proxy HttpRequest HttpResponse
      → IO HttpResponse
cache ref x@(HttpRequest (mth, uri, ver) hdrs bdy) pxy
  = do lst ← readIORef ref
       case lookup uri lst of
         Just y  → return y
         Nothing → do y ← pxy x
                      writeIORef ref ((uri, y) : lst)
                      return y
```

A soldier control for caching is then initialized as follows[4]

```
initHttpCache :: IO (Control HttpRequest HttpResponse)
initHttpCache
  = do ref ← newIORef []
       return (soldier (cache ref))
```

This control replaces the proxy.

5.6 A Monitoring Control

Some administrators like to *monitor* system activity [2]. Let us assume that monitoring involves nothing more than displaying a time, a request URI, and a response status code and reason on the console

```
monitor :: HttpRequest → HttpResponse → IO ()
monitor (HttpRequest  (mth, uri, ver1) hdrs1 bdy1)
        (HttpResponse (ver2, cde, rsn)  hdrs2 bdy2)
  = do utc ← getCurrentTime
       putStrLn (show utc ++ "  " ++ uri ++ "  " ++ cde ++ "  " ++ rsn)
```

A spy control for monitoring is then initialized as follows

```
initHttpMonitor :: IO (Control HttpRequest HttpResponse)
initHttpMonitor
  = return (spy monitor)
```

This control records all requests.

[4] Although strictly speaking a cache is not a control, it is something we are often asked to demonstrate!

5.7 A Composite Control

Of course, we could create many more small controls like this, each one an *executable specification*; that is, an inefficient solution which comes from transcribing the problem statement that may later be transformed into an efficient one by more-or-less formal reasoning [12]. Let us pause, however, to show how these small controls may be put together as a large one. A composite control that increases user anonymity, permits only authenticated users, filters undesirable addresses, screens undesirable content, caches content, and monitors system activity is initialized as follows

```
initHttpControl :: IO (Control HttpRequest HttpResponse)
initHttpControl
  = do httpAnonymize   ← initHttpAnonymize
       httpAuthenticate ← initHttpAuthenticate
       httpFilter      ← initHttpFilter
       httpScreen      ← initHttpScreen
       httpCache       ← initHttpCache
       httpMonitor     ← initHttpMonitor
       return
         ( httpAnonymize
         ∘ httpAuthenticate
         ∘ httpFilter
         ∘ httpScreen
         ∘ httpCache
         ∘ httpMonitor
         )
```

6 A Proxy Test Bench

Before closing, we briefly describe a proxy test bench. Straightaway, we stress that this test bench is robust enough for proxy development, but not for proxy deployment. The test bench behaves as both a client and a server. The function that makes the test bench behave as a client takes a proxy and a network handle. It receives a request on the handle, acts as the proxy with the request, and transmits the resulting response on the handle

```
client :: (Request a, Response b) ⇒ Proxy a b → Handle → IO ()
client pxy hdl
  = do x ← receive hdl
       y ← pxy x
       transmit hdl y
       hFlush hdl
       hClose hdl
```

The function that makes the test bench behaves as a server takes a proxy and a port. It casts off a new thread to act as a client with the proxy and a network handle for each connection on the port

```
server :: (Request a, Response b) ⇒ Proxy a b → PortID → IO ()
server pxy prtid
  = do sck ← listenOn prtid
       forever
         (do (hdl, hst, prt) ← accept sck
             forkIO (client pxy hdl)
         )
```

A main program may then use the test bench to run a particular proxy on a particular port

```
main :: IO ()
main
  = withSocketsDo
      (do httpControl ← initHttpControl
          server (httpControl httpProxy) (PortNumber 8000)
      )
```

7 Related Work

A general taxonomy of "middleboxes" was presented by Carpenter and Brim, who defined them as intermediate devices performing functions other than those of a standard router [13]. After studying this taxonomy, Joseph and Stoica constructed a middlebox model consisting of *zones*, which describe packet entry and exit points, *input pre-conditions*, which describe what packets are to be processed, *processing rules*, which describe how packets are to be processed, *state databases*, which describe the general state involved when packets are processed, *interest* and *state fields*, which describe the specific state involved when packets are processed, and *auxiliary traffic*, which describes any additional packets generated when packets are processed [14]. In his famous Turing Award lecture, Backus observed that conventional programming languages are both *fat*, because of their close coupling of semantics to state transitions, and *weak*, because of their inability to effectively use powerful combining forms for building new programs from existing ones [15]. Similarly, it seems to us that Joseph and Stoica's middlebox model is both fat, because it closely couples protocols and their processing, and weak, because of its inability to effectively use powerful combining forms for building new middleboxes from existing ones.

So far, functional programmers seem to have worked more on router programming than on middlebox programming. As far as router programming is concerned, Voellmy and Hudak used a domain specific language embedded in Haskell [16], and Loo *et al.* used a declarative language based on Datalog [17].

As far as middlebox programming is concerned, Capretta *et al.*, considered a proof of correctness of a conflict detection algorithm for firewall access control lists [18].

8 Future Work

Firstly, we plan to work on the systematic transformation of controls. The idea here would be to demonstrate both how and why we arrive at good solutions — ideally, with the clarity of [4] — so as to encourage others to try functional proxy *development*. Secondly, we plan to work on increasing the performance of controls. The idea here would be to improve the way that we process content and perform input/output — perhaps incorporating the ideas of [19] and [20] — so as to encourage others to try functional proxy *deployment*. Finally, we plan to combine this work by considering controls as *algorithmic skeletons* [21]. Algorithmic skeletons provide *skeleton definitions*, which are the same for all computers, and usually take the form of a higher-order functions, and *skeleton program transformations*, which introduce these functions [22]. Algorithmic skeletons also provide *skeleton performance models*, which give the resource consumption of skeleton functions, and *skeleton implementations*, which are different for sequential and parallel computers [22]. These models and implementations should assist functional proxy development and deployment — already, a composite proxy may be considered as a pipeline skeleton.

9 Conclusion

Network proxies are often rather large, somewhat disorganized pieces of software. Motivated by a concern for security, we have been working to bring the power of functional programming to bear on the problem of constructing proxies, and in this paper we illustrated one nice result of our work: a large, complicated proxy may be far more clearly expressed as a transparent one with many small, simple controls. These controls are either basic ones — tinkers, tailors, soldiers and spies — or composite ones. In future, we plan to work on the systematic transformation of controls and on their performance, so that they may be considered as algorithmic skeletons.

10 End Note

The subtitle of this paper is borrowed, with apologies, from le Carré's novel *Tinker Tailor Soldier Spy* [23]. Set during the Cold War, this story is one of the hunt for a mole at the very top of British Secret Service. The head of the Service, Control, succeeds in narrowing-down the list of suspects to five men, and assigns them unambiguous code-names: the ambitious Percy Alleline (Tinker); the suavely confident Bill Haydon (Tailor); the stalwart Roy Bland (Soldier); the officious Toby Esterhase (Poorman); and the razor-sharp George Smiley

(Beggarman). However, the operation supposed to yield one of these code names is compromised, and Control is forced to resign, dying soon afterwards. It later falls to Smiley reconstruct the operation, and by examining the actions of each of the other suspects, to uncover the mole.

References

1. Peterson, L.L., Davie, B.S.: Computer Networks: A Systems Approach (Fifth Edition). Morgan Kaufmann (2011) ISBN 978-0123850591.
2. Stallings, W., Brown, L.: Computer Security: Principles and Practice. Pearson Education (2012) ISBN 978-0273764496.
3. Peyton Jones, S.L., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003) ISBN 978-0521826143.
4. Bird, R.S.: Introduction to Functional Programming (Second Edition). Prentice Hall (1998) ISBN 978-0134843469.
5. Hudak, P.: The Haskell School of Expression: Learning Functional Programming Through Multimedia. Cambridge University Press (2000) ISBN 978-0521644082.
6. Hutton, G.: Programming in Haskell. Cambridge University Press (2007) ISBN 978-0521692694.
7. Thompson, S.: Haskell: The Craft of Functional Programming (Third Edition). Addison Wesley (2011) ISBN 978-0201882957.
8. Eckersley, P.: How unique is your web browser? In: Proceedings of the Symposium on Privacy Enhancing Technologies, Springer Verlag (July 2010) 1–18 LNCS 6205.
9. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7) (July 1970) 422–426
10. Cha, S.K., Moraru, I., Jang, J., Truelove, J., Brumley, D., Andersen, D.G.: SplitScreen: Enabling efficient, distributed malware detection. In: Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, USENIX Association (April 2010) 377–390
11. O'Sullivan, B., Goerzen, J., Stewart, D.B.: Real World Haskell. O'Reilly (2008) ISBN 978-0596514983.
12. Turner, D.A.: Functional programs as executable specifications. In Hoare, C.A.R., Shepherdson, J.C., eds.: Mathematical Logic and Programming Languages. Prentice Hall (1985) 29–54 ISBN 0135614651.
13. Carpenter, B., Brim, S.: Middleboxes: Taxonomy and issues. Request for Comments RFC3234, Internet Engineering Task Force (February 2002)
14. Joseph, D., Stoica, I.: Modeling middleboxes. IEEE Network **22**(2) (September-October 2008) 20–25
15. Backus, J.W.: Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. Communications of the ACM **21**(8) (August 1978) 613–641
16. Voellmy, A., Hudak, P.: Nettle: Taking the sting out of programming routers. In: Proceedings of Practical Aspects of Declarative Languages, Springer (January 2011) 235–249 LNCS 6539.
17. Loo, B.T., Gill, H., Liu, C., Mao, Y., Marczak, W., Sherr, M., Wang, A., Zhou, W.: Recent advances in declarative networking. In: Proceedings of Practical Aspects of Declarative Languages, Springer (January 2012) 1–16 LNCS 7149.
18. Capretta, V., Stepien, B., Felty, A., Matwin, S.: Formal correctness of conflict detection for firewalls. In: Proceedings of the Workshop on Formal Methods in Security Engineering, ACM (November 2007) 22–30

19. Snoyman, M.: Warp: A Haskell web server. IEEE Internet Computing **15**(3) (May — June 2011) 81–85
20. Collins, G., Beardsley, D.: The Snap framework: A web toolkit for Haskell. IEEE Internet Computing **15**(1) (January — February 2011) 84–87
21. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989) ISBN 0262530864.
22. Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N., Wu, Q., While, R.L.: Parallel programming using skeleton functions. In: Proceedings the International Conference on Parallel Architectures and Languages Europe, Springer-Verlag (June 1993) 146–160 LNCS 694.
23. le Carré, J.: Tinker Tailor Soldier Spy. Sceptre (2011) ISBN 978-0340993767.

The Quintessential Neural Network Programming Language

Gene I. Sher

Department of EECS, University of Central Florida

gsher@knights.ucf.edu

Abstract. The architecture of most programming languages differs significantly from the architecture of the problem domain concerning the construction of neural network based computational intelligence. This paper makes a claim that Erlang has the architecture which maps perfectly to neural network based systems, and is thus the quintessential neural network programming language, and which this author believes to be an essential element in the future of computational intelligence research and implementation. Furthermore, this paper presents a case study of a topology and parameter evolving universal learning network called DXNN, developed purely in Erlang. Finally, the paper demonstrates the advantages Erlang provides to computational intelligence based systems built through it, and the new frontiers that it paves the way towards within the field.

Keywords: Erlang, Neuroevolution, Concurrency, Distributed Computing, Evolutionary Computation, Neural Networks, DXNN.

1 Introduction

There is nothing mystical about the human brain, it is but a neurocognitive computer, carved out in flesh through billions of years of evolution. When it comes to computational intelligence, we know of one method that, given advanced enough technology and computational power, will produce truly intelligent agents. That method is the simulation of such a biological neurocognitive system on a non biological substrate, it is the approach to computational intelligence through neural networks and evolutionary computation. The proof that this approach works is us, neural network based intelligent agents, whose neurocognitive computers have been evolved through billions of years of trial and error. Finally, as was shown by the blue brain project [1], we already have proof that the non biological neural networks behave just as their biological counterparts. Nature does not care whether the computations are performed in flesh or in silicone, as long as the mathematics behind it is the same.

Artificial neural networks are mathematical abstractions of the biological neurocognitive systems, and similarly to a biological neural network, the artificial neural network is a vast graph of interconnected signal processing nodes as is shown

adfa, p. 1, 2011.

in Fig-1. The following figure presents a diagram of a neural network, a graph of interconnected simple processing elements. Those in the input layer are fed vector encoded percepts by sensors, and those in the output layer, forward their signals to the actuators which interpret those vector signals and act upon the world. The sensors of a neural network can be cameras (real, or simulated within a virtual environment), pressure sensors, Geiger counters, or any other programs which interface with hardware, the OS, or some database for information, and then vector packages it and forwards it to the neurons. The actuators can be programs controlling servos of a robot (real or simulated), or programs which write data to files, or dump it to screens.

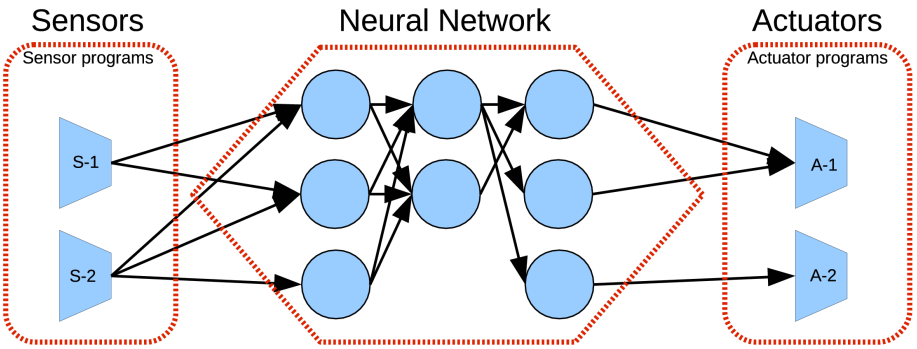


Fig. 1. A neural network with sensors and actuators

The neural network (NN) approach to computational intelligence (CI) has produced some very powerful and interesting systems. With just a sample [2-9] of the contributions of these systems ranging from controlling robotic systems, to controlling artificial organisms in ALife experiments, giving the organisms the ability to hunt and ambush each other [10], to controlling highly complex appendages [11,12], to performing time series analysis and autonomously trading financial instruments [13-16], to being used in vision systems, picking out patterns, and finally, to even predicting epitopes on an antingen amino acid sequence [17]. There are many ways to evolve topologies for the NNs and teach NNs to perform various tasks, or even have neural plasticity based on biological plasticity rules [18,19], letting the NNs learn and adapt during their lifetime all on their own. But the architecture of these systems, as can be seen from the above figure, is significantly different from the architecture of the standard programming languages most commonly used within the software industry and the Scientific community, languages like C/C++/Java/Perl/Python... Thus there exists a conceptual gap between the architecture of the tools used to build such systems, and the actual architecture of these systems. A conceptual gap that we must constantly walk around and trudge through, to convert our ideas into prototypes and functional systems. Neural networks, population based evolutionary algorithms, these are all concurrent, distributed, multi-agent based systems where the various self contained agents interact with each other through signals and messages, yet the languages we use to develop such systems are non-parallel, not inherently concurrent, and vastly different in behavior, and concept. It is this difference that requires us to constantly jump through unnecessary hoops to get

from ideas to implementations, and it is this difference that at times prevents us to even consider the actual ideas due to our being stuck thinking in the programming languages that is most prevalently taught in universities (C/C++/Java), being held back by linguistic determinism [22].

This paper makes a claim that there is indeed an existing functional programming language that is perfect for the development of neural network based computational intelligence research, and indeed all distributed and multi-agent based computational intelligence systems, and that this language is Erlang. To demonstrate the validity of this claim, the author first asks the question of what are the essential features a neural network programming language would need to have, had one the chance to build it from scratch without being constrained by resources or other obstacles. The paper then discusses the features that Erlang offers, and notes a 1:1 mapping between this concurrency oriented (CO) programming language, and the NN based CI problem domain. Once this perfect mapping is demonstrated, the author provides a case study of the currently existing, purely Erlang built, topology and parameter evolving universal learning network system called DXNN [20]. Discussing the architecture of this system, the features it offers, and the features that it could later offer, which would be very difficult to develop through another programming language, yet naturally provided by Erlang. Finally, the paper concludes with a discussion of the future of this language and machine learning, and the urging of this author for the scientific community to utilize the Erlang programming language due to the direction of hardware towards many-core architectures, and the accelerated results the language can provide within the problem domain. It must be noted that it is not the aim of this paper to discuss NN algorithms, or compare neuroevolutionary systems, but merely to discuss and demonstrate why Erlang is the quintessential neural network programming language, the benefits, features, and opportunities that Erlang offers to the future, research, and creation of distributed CI systems.

2 Creating a Perfect Neural Network Programming Language

If you had the chance to create a programming language that is perfect for developing NN systems, and other types of distributed CI systems, what features would it need?

The first thing we'd want from our neural network programming language, is for its architecture to mirror that of the NN's, so that there is as little of a conceptual gap as possible between the two, or preferably no conceptual gap at all. This would make it possible for all the ideas and innovations within the field of neural computation to be exactly and directly represented by this new programming language. This would allow the researcher to think about NNs rather than the procedural or object oriented systems instead, and then having to translate the ideas back into the field of NN based systems. This alone already requires that the programming language has structures which are similar to those within neural networks: connections, message passing ability, concurrent and independent signal processing elements... For this reason, this programming language would need at least the following two features:

1. Neural networks are composed of independent, concurrent, distributed processing

units called neurons. Thus the programming language architecture would need to support having such elements, having independently acting processes which can all function in parallel, and which can be easily distributed throughout the modern parallel hardware, or even the global network, and which are able to accept and process signals asynchronously.

2. The neurons in NNs communicate with each other through signals. Thus the programming language architecture needs to allow the processes to communicate with each other through messages or signals too, and for those signals to have the ability to be encoded in any form or syntax.

But just having the programming language's architecture mirror the architecture of neural networks is not enough. Our brains are robust, they posses graceful degradation, where a number of neurons can die, indeed hundreds die every day, yet our brains continue to function, degrading gracefully. This type of robustness is provided through the vastness and interconnectedness of the NN graph, and simply requires that neurons are independent, concurrent processing elements. But there is another type of robustness: in our wetware we usually don't encounter situations where we get a “bug”, and suddenly crash. In other words, our brains, our neural systems, our biological makeup is robust, fault tolerant, and self recovering. Thus if we want for the language architecture to support the creation of a true computational intelligence, it needs to allow for a similar level of robustness. It must allow not simply the graceful degradation due to the interconnectedness within the NN, but the language must be made with fault tolerance in mind, with the built systems having the ability to self recover, to run forever, so that the programming language does not get in the way of the actual NN system. The programming language must give us the ability to build distributed systems with at least 99.999% uptime. To accommodate this, it needs the following features:

1. Allow for an easy way to recover from errors.

2. If one of the elements of the computational intelligence system crashes or goes down, the CI system must have features that can recover and restart the crashed elements automatically.

3. There must exist multiple levels of security, such that the processes are able to watch each other's performance, monitoring for crashes and assisting in recovering the crashed elements.

Another feature we we want is as follows: though the NN itself takes care of the learning part, performs the incorporation of new ideas, the growth and experience gaining, there is one thing that biological organisms do not have the ability to do with regards to their intelligence. Biological organisms do not have the ability to modify their own neural structures, we do not have the ability to rewrite our own NNs at will, nor our genetical makeup, we do not have the ability to update the very manner in which we function, in which our biological NNs process information... but that is the limitation of biological systems only, and non biological systems need not have such limitations. Thus an ideal CI programming language architecture must also provide

the following features:

1. The programming language must allow for code hot-swapping. For the ability for the CI system to rewrite the code that defines its own structure and functionality, its own neural network, and have the ability to fix errors and then update itself, its own source code, without taking anything offline.

2. The programming language architecture must allow for the CI system to be able to run forever, crashes should be local in nature, and with code hot-swapping, should thus be fixable by the CI system itself.

Finally, taking into account that the neural network based CI systems should be able to interface and control robotic systems, be used in Unmanned Ariel Vehicles (UAVs), or in bipedal robots, the programming language should from the start make it easy to develop and allow for control of a lot of different types of hardware. It should allow: for an ability to easily develop different and numerous hardware drivers.

In summary, the programming language architecture that we are looking for must process information through the use of independent concurrent and distributed processes. It should allow for code hot-swapping. It should allow for fault tolerance, error fixing, self healing, and recovery. Finally, it should be made with ability to interface with a large number of hardware parts, allow for an easy way to develop hardware drivers so that not only the software part of the CI be allowed to grow and self modify and update, but it should also be able to incorporate and add new hardware parts over time, whatever those new parts may be. A list of features that a NN based CI system needs, as quoted from the list made by Bjarne Däcker [2], is as follows:

1. “The system must be able to handle very large numbers of concurrent activities.

2. Actions must be performed at a certain point in time or within a certain time.

3. Systems may be distributed over several computers.

4. The system is used to control hardware.

5. The software systems are very large.

6. The system exhibits complex functionality such as, feature interaction.

7. The systems should be in continuous operation for many years.

8. Software maintenance (reconfiguration, etc) should be performed without stoping the system.

9. There are stringent quality, and reliability requirements.

10. Fault tolerance“

Surprisingly enough, Däcker was not talking about a NN based general

computational intelligence programming language when he made the above list, he was talking about a programming language for the development of *telecom switching systems*. And it is for the construction of exactly such systems that Erlang was specifically created for.

3 From Telecommunications Networks To Neural Networks

Erlang is a functional concurrency oriented (CO) programming language. With its origins and inspiration from Prolog. It was developed at Ericsson, a project lead by Dr. Joe Armstrong. Erlang was created for the purpose of developing telecom switching systems. Telecom switching systems have a number of demanding requirements, such systems are required to be highly reliable, fault tolerant, they should be able to operate forever, and act reasonably in the presence of hardware and software errors.

These features are so close to those needed by NN based CI systems, that the resulting language's features are exactly those of a neural network programming language. The features that Erlang possesses, as quoted from Armstrong's thesis [3], is as follows:

"1. Encapsulation primitives — there must be a number of mechanisms for limiting the consequences of an error. It should be possible to isolate processes so that they cannot damage each other.
2. Concurrency — the language must support a lightweight mechanism to create parallel process, and to send messages between the processes. Context switching between process, and message passing, should be efficient. Concurrent processes must also time-share the CPU in some reasonable manner, so that CPU bound processes do not monopolize the CPU, and prevent progress of other processes which are 'ready to run.'
3. Fault detection primitives — which allow one process to observe another process, and to detect if the observed process has terminated for any reason.
4. Location transparency — If we know the PId of a process then we should be able to send a message to the process.
5. Dynamic code upgrade — It should be possible to dynamically change code in a running system. Note that since many processes will be running the same code, we need a mechanism to allow existing processes to run "old" code, and for "new" processes to run the modified code at the same time.
With a set of libraries to provide:
6. Stable storage — this is storage which survives a crash.
7. Device drivers — these must provide a mechanism for communication with the outside world.
8. Code upgrade — this allows us to upgrade code in a running system.
9. Infrastructure — for starting, and stopping the system, logging errors , etc."

Erlang provides all of these, and it is for this reason why it is such a perfect programming language for distributed CI system development. Whereas before one would need to first create the NN algorithms, topologies, and architectures separately,

and then try to figure out how to map the programming language like C++ to the task domain, or even worse, think in C++, and thus create a sub-par and compromised NN based system... The ideas, the algorithms and NN structures are mapped to Erlang perfectly, and vice versa. The ideas that would otherwise be very difficult to implement, or even consider when one thinks in one of the more commonly used languages, are easily and clearly mapped-to, and implemented-with, this functional programming language. One does not need to switch from thinking about NN systems, algorithms, and architectures when developing in Erlang, to thinking about the programming language and how to implement such systems through it, for they are one and the same. The conciseness of the language, the clarity of the code and the programming language's architecture... make even the most complex problems which would otherwise not be possible to solve, simply effortless.

3.1 The Conceptual Mapping of a NN to Erlang's Architecture

In Erlang, concurrency is achieved through processes. Processes are self contained, independent, concurrently running micro servers/clients, primarily interacting with each other through message passing. Already you can visualize that these processes are basically neurons, independent, distributed, concurrent... primarily communicating with each other by sending signals, action potentials, messages.

Once again taking a quote from Armstrong's thesis, where he notes the importance of there being a one to one mapping between the problem and the program: “It is extremely important that the mapping is exactly 1:1. The reason for this is that it minimizes the conceptual gap between the problem and the solution. If this mapping is not 1:1, the program will quickly degenerate, and become difficult to understand. This degeneration is often observed when non-CO languages are used to solve concurrent problems. Often the only way to get the program to work is to force several independent activities to be controlled by the same language thread or process. This leads to an inevitable loss of clarity, and makes the programs subject to complex and irreproducible interference errors.” We see that the mapping from Erlang's architecture to neural networks is 1:1, as shown in Fig-2.

From the figure it becomes obvious that indeed, there is a perfect correlation between the architecture of this programming language, and the NN problem domain. In the figure each neuron is directly mapped to a process, each connection between the neurons is a connection between processes. Every signal, simulated action potential that is sent from one neuron to another is a signal, a message in vector/list or tuple form, sent from one process to another. We could not have hoped for a better mapping. Erlang was also created with an eye towards scaling to millions of processes working in parallel, so even here we are in great luck, for the future in NN will require vast NN based systems, with millions or even billions of adaptive neurons on every computing node. Also, because robotics is such a close field to CI, the evolved NN based systems will need to be able to interface with numerous sensors and actuators, with the hardware in which the CI is embedded and which it inhabits; again Erlang is perfect for this, it was made for this, it was created to interface and interact with varied types of hardware, and it was created such that developing drivers is easy. Erlang was created to be not simply concurrent, but distributed, over the web or any other medium. This alone makes the frontier of large, distributed CI systems that

much more approachable and manageable.

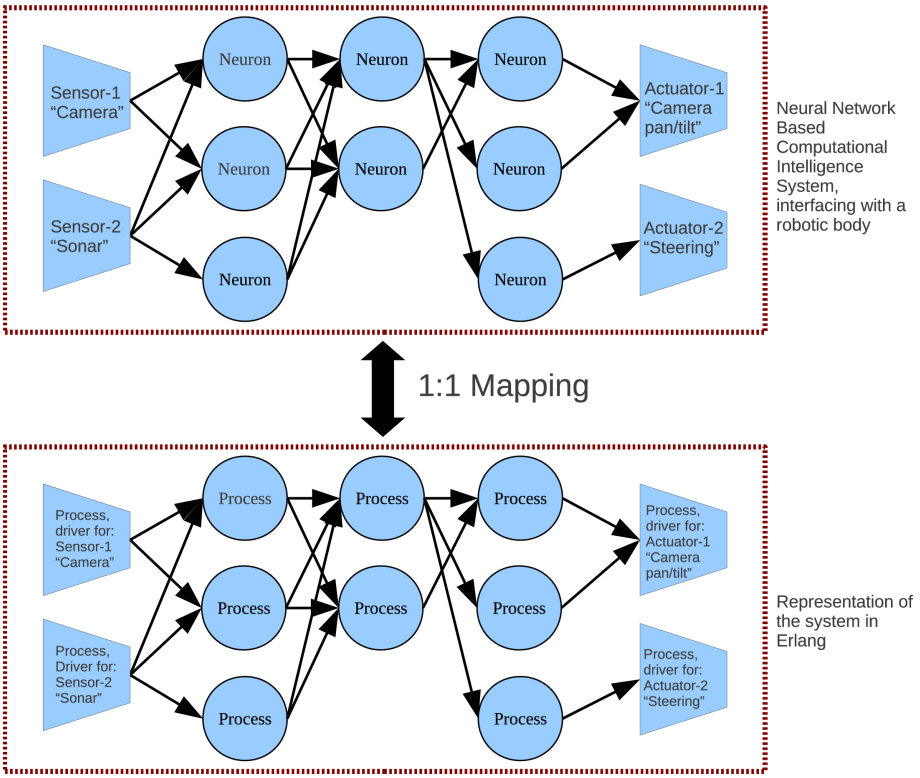


Fig. 2. A 1:1 mapping from Erlang to the NN based CI systems

But of course there are other important features, beyond that of scaling and the perfect mapping from the problem to the solution domain. There is also the issue of fault tolerance, the issue of robustness... It would be rather comical if it were possible for an advanced CI system to be brought down by a single bug. Here again Erlang saves us. This programming language was designed to develop systems that must run forever, that cannot be taken off-line, even when there is a bug, and even when it absolutely must be fixed. Through supervision trees Erlang allows for processes to monitor each other, and to restore each other. Thus if any element of the neural network crashes, it does not bring down the whole system. We can create an external self monitoring system, an *exoself* of the CI system, which can monitor the system's performance and restore it to the previously functional form. But not only can it restore the CI to a previously functional form in the case of emergency, it can also allow for the bug to be fixed, and the new updated source code to be ran without taking the system offline, all using the Erlang's code hot-swapping capabilities. It allows for the system to fix itself, to self heal, to recover, to upgrade, to rewrite itself, and to evolve. What other programming language can be said to offer such features so, naturally?

4 DXNN: A Case Study

Having now discussed the features that Erlang offers, and thus the advantages and the reasons behind this author's considering Erlang to be such a perfect fit for the field of neural network based computational intelligence, we now move to a discussion of the application of Erlang to the construction of an open source topology and parameter evolving universal learning network called DXNN [20,21]. DXNN is the *first purely Erlang built,* general, topology and parameter evolving universal learning network platform. In the following subsections we will overview this system, by first briefly discussing the standard memetic neuroevolutionary algorithm, then discuss DXNN's NN based agent architecture, and then finally the whole system, putting it into the perspective by framing it within the memetic algorithm based steps of our discussion.

4.1 Overview of a Memetic Algorithm

What makes the memetic algorithm different from genetic, is simply the separation of the local and global search parts of the algorithm into two separate phases. In the case of the DXNN, the algorithm is as follows:

1. **Initialization:** A population of minimalistic seed agents is created.
2. **Evaluation:** Agents are converted from their genotypes to their phenotypes, and applied to the problem to be evaluated.
    1. **Local Search/Tuning:** For each agent, after each evaluation, a perturbation to its synaptic weights is applied. The algorithm selecting the neurons and the weights to be perturbed or modified, is based on what local search algorithm is used. It can be as simple as a stochastic hill climbing algorithm, or as complex as a custom ant colony optimization algorithm. Tuning is performed until some local search termination condition is reached by the agent (some maximum number of local search evaluations, or processing time used for this weight *tuning* phase.)
3. **Selection:** After all the agents have been tuned (through local search), some selection algorithm (*rank, top_3...*) is used to choose the fit agents to act as parents for the purpose of creating offspring agents.
4. **Global Search:** From the selected parent agents the offspring are created, by for example taking the parent, creating a clone of it, mutating its topology and parameters, and designating the mutant clone as the offspring. After the new generation of agents is created, we go back to step-2.

The loop composed of steps 2-4, continues until some termination condition is reached, such as the maximum number of total evaluations performed within the population, or some maximum amount of computational power used by the system.

4.2 A Quick Overview of a DXNN's NN Based Agent

With DXNN being written purely in Erlang, every Neuron, Sensor, Actuator, the Synchronizing element called Cortex, and the monitoring process called Exoself, are independent concurrent processes which can only communicate with each other through message passing. An agent, an example architecture of which is shown in Fig-3, is composed of 3 structural levels. At the lowest level are the n*eurons*, *sensors*, and a*ctuators* that form the NN. At the second level is the *cortex* which synchronizes the neurons. Finally, at the top level is the e*xoself* process which monitors these agent composing elements, and can perform NN repairing routines, and communication routines between the NN based agent, and the infrastructure of the neuroevolutionary platform, the evolutionary algorithm, as will be discussed in the next subsection. In DXNN, a Neuron can utilize any type of signal integration function, not just the standard dot product, but also for example weighted input multiplication, or a function calculating the weighted result of the difference between input vector at time *T* and *T-1*... Also any type of activation function, such as a sigmoid, Gaussian, sin, cos... And any type of signal pre and post processors can be used. All of these functions can be mutated between the ones available to the agent, and the availability is presented to the evolutionary algorithm through a list of such function names.

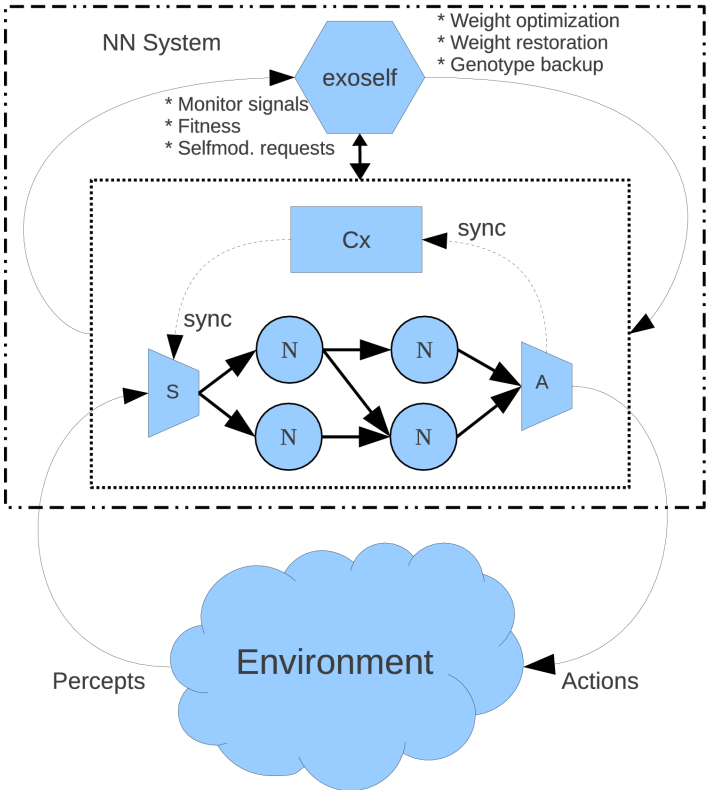


Fig. 3. The hierarchical structure of a directly encoded NN based agent

What input the NN gets and what its output is used for, is determined by the sensors and actuators respectively. The Sensors and Actuators are the processes which deal with the acquisition of percepts from the environment, and perform actions upon it, respectively. The environment can be a simulated world, a database, the OS, or the outside world, and is referred to within the system as a *scape* [20]. The sensors poll the environment for data, which they then package as vectors and pass them to the appropriate input layer Neurons. The Neurons in the output layer, those neurons which are connected to Actuators, forward their output signals to the actuators they connect to, which then post-process those output signals, and based on the resulting vector interact with the environment. The Cortex element is necessary to synchronize and pace the NN system. When the actuators of the NN have finished acting upon the environment, they send the *cortex* element the sync message, once the cortex element received such a message from all the actuators, it sends a sync message to the sensors, calling them to action. The cortex element allows us to synchronize the system, and provides an extra process monitoring element. It also gives us a process through which we can put the whole NN system on pause, for example when we need to mutate the synaptic weights of the NN's neurons, or revert those weights back to their previous state. This is done simply by contacting the cortex, which in response to such a message would simply gather all the sync messages from the actuators, but not call the sensors to action, thus effectively pausing the NN system.

Beyond all of this, there is another process, the process that is part of the NN based agent, but which is not part of the NN processing or synchronizing system, that process is called the *exoself*. The exoself is a process that is part of the NN based agent, but is outside the NN, monitoring the elements of the NN, and thus having access to the whole system. This means that the exoself can monitor all the processes in the NN, have access to all the neurons, have the ability to backup the whole system regularly to the mnesia database, repair elements that have crashed, perform algorithms that require global knowledge/view of the state of the NN system (exp. SOM), and act as the part of the agent that communicates with the neuroevolutionary system, and whose Id/PId can act as the unique designation of the whole agent.

But what must be noted here though is how every single one of these elements, is a concurrent process, communicating with the others through messages. It is natural to perceive these parts of the NN system as independent and concurrent, and Erlang can represent them as such, allowing for a natural 1:1 mapping, as is shown in Fig-3.

4.3 The DXNN Neuroevolutionary Platform

Having discussed the memetic algorithm used, and the architecture of a DXNN agent, we now move to a discussion of the DXNN's whole neuroevolutionary architecture. A diagram of the global overview of the system is shown in Fig-4. Let's put this system into perspective by going through the memetic algorithm, and see how all these parts fit into it, and the roles they play within it.

Because DXNN uses the mnesia database to store the genotypes of the agents, before anything can be run, the infrastructure of the system is started. To do this, the researcher executes *polis:start()*, which starts mnesia, and all the simulated

environments/scapes and other programs that need to run independently of the agents.

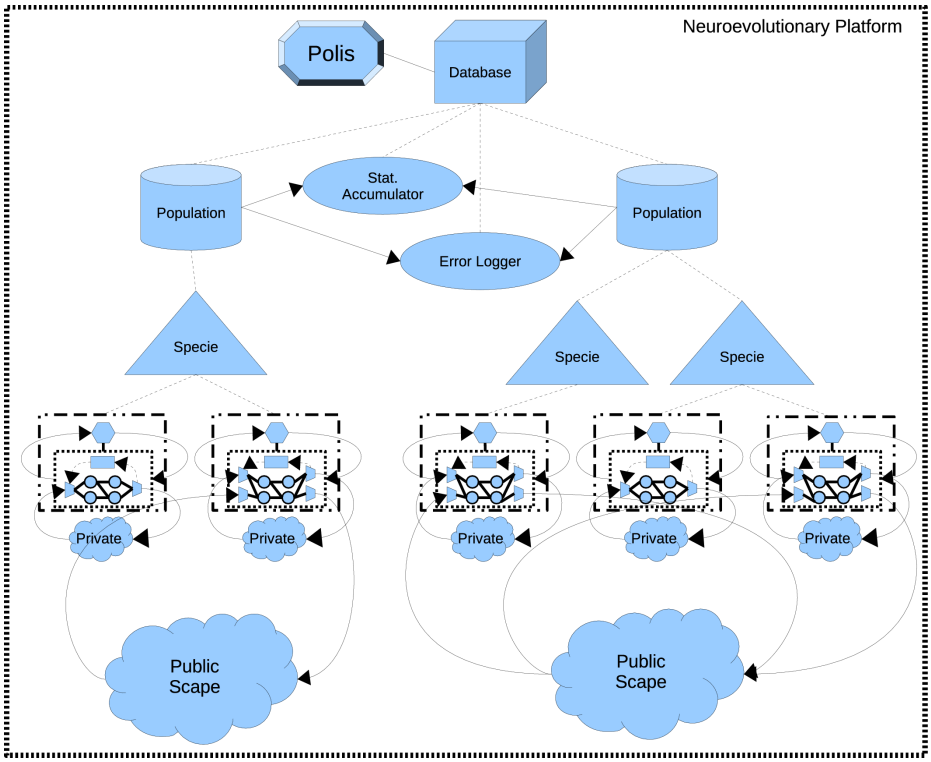


Fig. 4. The DXNN platform

Once the polis process is started, which keeps track of the mnesia, error logger, and acts as a router for persistent and agent independent simulations (public scapes), the researcher can run the experiment:

1. **Initialization:** The researcher first sets up the record by the name *constraint* which specifies all the parameters (constraints) for the particular evolutionary run. A sample of these parameters is as follows: Population size. Number of distinct species, and the set of sensors and actuators that each agent belonging to that specie starts with and has access to. A list of available activation functions as a list of tags, where each tag is the name of the actual activation function. With this, during evolution, the evolutionary algorithm can create new neurons which are created with a randomly selected activation function from this list. A list of sensors and actuators available for each particular specie. Though the seed agents start with some particular minimal set, or even a single sensor and actuator, over time as new offspring are created, a mutation operator like *add_sensor* and/or *add_actuator,* can be applied to the agent, which will add and connect a new sensor and/or actuator to the NN, allowing the agent to explore new morphological properties, and how different sensory inputs and actions

performed, can be utilized to achieve higher fitness. A list of global mutation operators from which random ones are selected during the topological mutation phase (global search). A list of plasticity rules which could be used by the neurons. A list of signal integration functions (such as the dot product, but also others)... The full list is available in source code [21].

2. With the *Constraints* record set, *population_monitor:new(Constraints)* is executed by the researcher. This function, based on the specified constraints, creates the seed population of NN based agents. The genotypes of the agents are Id linked records stored within the mnesia database. Each cortex, sensor, actuator, and exoself, is a record with all the needed data for that process. Once the genotypes of seed agents are created, the *population_monitor* process is spawned, which then runs through the mnesia database, and converts each exoself record into a process. The exoself process of each agent then spawns the remainder of the processes (neurons, sensors, actuators, and the cortex), links them together, and then triggers the cortex to action. At this point all the NN based agents are functional.
3. **Evaluation:** Each agent is evaluated by interacting with some *scape/s,* where the scape is a simulation, not necessarily of a 3d environment. A scape is used to represent a problem, with an ability to also gage fitness of the agent's performance. For example a XOR truth table, or a double pole balancing simulation, can be represented as scapes. The scape, based on its parameters, determines when the evaluation has finished (after agent reaches the end of the truth table, or when for example the agent dies within the simulated environment in a ALife scenario), and gives the agent it's fitness score.
    1. **Local Search/Tuning:** The exoself of each agent decides on whether to further perturb the synaptic weights of the NN using some local search algorithm, or whether to backup the best parameter set (synaptic weights and others) combination achieved, and its correlating fitness, to the database, and inform the population_monitor that it has finished with its local search evaluations and synaptic weight & parameter tuning.
4. **Selection:** Once the population_monitor process receives the *completion* signals from all the NN based agent exoselves, it executes the selection function: *population_monitor:Selection(Agent_Ids)*. Just as with everything else, there are numerous selection algorithms available, and can be used and experimented with. Using some selection algorithm, lets say a variation of a multi-objective, hall of fame based *rank* algorithm, the population_monitor selects a list of fit agents to be used as parents.
5. **Global Search:** Once the fit agents have been chosen, the population monitor, based on a function which calculates for each agent how many mutation operators to apply, applies random number of random mutation operators chosen from the list of such operators, with the probabilities specified within said list, values which themselves are mutatable (which gives the system the ability to evolve evolutionary strategies as well). The mutant clones, the offspring, compose the new generation of agents, at which point we go to step-2.

Finally, the population_monitor keeps track of how many total evaluations have been

performed, and other population statistics, and based on this data and the chosen termination condition, it then decides on whether the evolutionary run has completed.

5 Beyond the Horizon

Having now discussed the reason for Erlang's excellent fit into the field of computational intelligence, particularly the distributed neural network based types, and having discussed the use of Erlang in the construction of DXNN neuroevolutionary platform, we now discuss things that have not yet been implemented within the DXNN system, but can easily be done so in the future.

5.1 Self Modification and Self Evolutionary Experimentation

One of the very interesting features that Erlang based computational intelligence can posses, which would be difficult or even impossible to implement in another language, is a self modifying and rewriting intelligent agent. Though we humans do posses brains which can change over our lifetimes, it is not the case that we can double the size of our brains, or rewrite the very deep structures of our neurocognitive system as we find more efficient and better solutions of computation. This is not the case with non-biological systems, which can evolve and change at the speed of their creativity. As soon as a new idea for computation emerges, an intelligent system can rewrite itself to incorporate it. We have already noted that with Erlang and in DXNN, an evolving agent can incorporate new actuators and sensors. One set of such sensors and actuators can be a set of sensors that read the agent's own NN topology and architecture. The sensor would encode it in a vector form, and thus let the agent peak inside its own neural structure. Also actuators can be made available which let the agent traverse its neural topology, and have the ability to add new neurons, or make new connections and cut old ones, live. This set of sensors and actuators would allow the NN based agent to rewrite itself, once it has evolved for such functionality in some ALife environment, which is not a far-fetched idea when the agents are evolved within the ALife scenarios in which such self rewriting provides a survival advantage.

Or a sensor can be made to peak into another agent's neural structure, analyze the architectures of other agents, to get new ideas for neurocognitive processing. Self rewriting is easy in Erlang, which allows code hot-swapping. This means an actuator that opens the source code for the agent's own modules, can allow it to rewrite the code, recompile it, add new features, and thus truly give the agent the ability to rewrite itself, to truly evolve and change in any direction, and gain any functionality imaginable. If the agent rewrites something that causes a crash, the crash is local, and the self healing and monitoring and robustness that Erlang provides can come into play, the *exoself* process comes into play. At such an event the exoself can respawn the crashed elements, and restore the previous functionality/implementation, with a self sent signal such that the agent knows it has just recovered from a crash. One of the great things about Erlang is that it's made for the creation of such robust systems, systems that must stay on-line forever. The monitoring features and the architecture used within DXNN agents, makes it very easy to create such distributed, self healing neural networks, where self rewriting mistakes are manageable.

5.2 Decreasing Message Passing Overhead Ratio

Erlang is most efficient when processes do large computations and exchange small messages. So it can be noticed that if we have very simple neurons with each given its own process, the processing conducted by each such neuron, and the number of messages exchanged, produces an non-advantageous messages/computation ratio. As the NN grows and evolves, some parts of the NN will be untouched by the evolutionary process in surviving agents. Some NN structures will stabilize in such a manner that when they get modified during offspring creation, the offspring will not be functional, thus the only surviving offspring will be those in which mutation did not touch those structures. We can keep track which structures stabilized within a NN, and rewrite them as single process modules rather than compositions of neuron processes. This will allow the NN based agent to increase the ratio of processing done per process against the number of messages exchanged.

But even without this, one has to note that the more advanced NN based agents will have not simple static neurons, but plastic neurons. Neurons using plasticity and activation and signal integration functions more complex than the simple sigmoid with dot product respectively, will have the messages/computation ratio significantly more advantageous.

5.3 Concurrency and Distribution

The hardware is moving towards the many-core architecture. The number of Cores available on a single machine is increasing with every year. Only recently Intel announced the release of Xeon Phi, a co-processor composed of 50+ x86 cores. A programming language like Erlang, and thus the systems like DXNN written in it, are some of the few of the systems which could seamlessly be migrated to such a machine and begin to utilize it immediately.

Furthermore, Erlang allows us to distribute the NN based systems not simply over the Cores and CPUs, or over multiple machines, but over the entire global network. This opens up possibilities of a globally distributed neural network with significant computational power, robustness, fault tolerance, and ability for recovery.

6 Conclusion

Procedural and other standard programming languages do not have a 1:1 architectural mapping to the field of neural network computational intelligence. But there is one programming language that does, Erlang. Erlang is a concurrency oriented message passing paradigm functional programming language. It maps perfectly to the field of NN based CI, and has a slew of features perfect for building advanced neurocognitive systems that would be prohibitive or even impossible to implement in another language.

One of the first such neuroevolutionary systems built in Erlang is called DXNN [20-21]. DXNN is a neuroevolutionary platform capable of evolving universal learning networks, taking advantage of the process and message passing based computing offered by Erlang. But there is still an enormous amount of things to

explore and advance within the field, all possible and easily testable in Erlang. Erlang offers the flexibility within the field that we have not seen before. It allows for rapid prototyping, and it allows us to transfer our ideas directly to the implementation, without jumping through the hoops of a programming language not made for this field. With Erlang, linguistic determinism does not limit our ideas, and we can think naturally about distributed NN based CI systems. Erlang offers new opportunities within the field, it opens new frontiers previously not even considered. Erlang gives us the ability to look beyond the horizon of Computational Intelligence.

7 References

[1] The Blue Brain Project EPFL, http://bluebrain.epfl.ch.

[2] Floreano D, Mondada F (1998) Evolutionary neurocontrollers for autonomous mobile robots. Neural Networks 11:1461–1478

[3] Vintan LN, Iridon M (2002) Towards a high performance neural branch predictor. In: IJCNN99 international joint conference on neural networks proceedings (IEEE Service Center), p 868–873

[4] Floreano D, Urzelai J (2000) Evolutionary robots with on-line self-organization and behavioral fitness. Neural Networks 13:431–443

[5] Engel Y, Szabo P, Volkinshtein D (2006) Learning to control an octopus arm with gaussian process temporal difference methods. Adv Neural Inf Process Syst 18c:347–354

[6] Alon K (2004) Analyzing evolved fault-tolerant neurocontrollers. In: Proceedings of the ninth international conference on the simulation and synthesis of living systems. (ALIFE9)

[7] Hastings EJ, Guha RK, Stanley KO (2009) Automatic content generation in the galactic arms race video game. IEEE Trans Comput Intell AI in Games 1:1–19

[8] Edition S (2005) Artificial life models in software. In: Adamatzky A, Komosinski M, (eds) Springer-Verlag, New York

[9] Floreano D, Urzelai J (2000) Evolutionary robots with on-line self-organization and behavioral fitness. Neural Networks 13:431–443

[10] Sher GI (2010) DXNN platform: the shedding of biological inefficiencies. Neuron 1–36. Available at: http://arxiv.org/abs/1011.6022.

[11] Engel Y, Szabo P, Volkinshtein D (2006) Learning to control an octopus arm with gaussian process temporal difference methods. Adv Neural Inf Process Syst 18c:347–354

[12] Siebel NT, Sommer G (2007) Evolutionary reinforcement learning of artificial neural networks. Int J Hybrid Intell Syst 4:171–183

[13] Halliday R (2004) Equity trend prediction with neural networks. Res Lett Inf Math Sci 6:15–29

[14] Min Qi, Peter GZ (2008) Trend time-series modeling and forecasting with neural networks. IEEE Trans Neural Networks 19:5

[15] Jung H, Jia Y et al (2010) Stock market trend prediction using ARIMA-based neural networks. In: 2008 Proceedings of 17th international conference on computer communications and networks 4:1–5

[16] Li Y, Ma W (2010) Applications of artificial neural networks in financial economics: a survey. In: 2010 International symposium on computational intelligence and design, pp 211–214

[17] Saha S. Raghava G (2006) Prediction of Continuous B-Cell Epitopes in an Antigen Using Recurrent Neural Network

[18] Oja E (1982) A simplified neuron model as a principal component analyzer. J Math Biol 15:267–273

[19] Soltoggio A, Bullinaria JA, Mattiussi C, Durr P, Floreano D (2008) Evolutionary advantages of neuromodulated plasticity in dynamic, reward-based scenarios. Artif Life 2:569–576

[20] Sher G (2012) Handbook of Neuroevolution through Erlang, Springer-Verlag, New York

[21] DXNN Source Code: https://github.com/CorticalComputer/DXNN

[22] Whorf, B.L. (1956). "The Relation of Habitual Thought and Behavior to Language." In J.B. Carroll (ed.) *Language, Thought, and Reality: Selected Writings of Benjamin Lee Whorf* (pp.134–159). Cambridge, MA: MIT Press. ISBN 0-262-73006-5

Have Your Cake And Eat It, Too

Generic sorting and partitioning in linear time and fully abstractly-simultaneously

Fritz Henglein

henglein@diku.dk@uu.nl
Department of Computer Science (DIKU)
University of Copenhagen
2100 Copenhagen, Denmark

Abstract. Discrimination is a generalization of both sorting and partitioning: It partitions list elements according to a user-specifiable equivalence or ordering relation and, for ordering relations, lists the resulting blocks in ascending order. We show how discriminators (discrimination functions) can be defined generically by structural recursion on representations of ordering and equivalence relations. They improve the asymptotic performance of generic comparison-based sorting and partitioning and yet do not to expose more information about the input elements than their pairwise ordering, respectively equivalence relation. For a large class of order and equivalence representations, including all standard orders for regular recursive first-order types, the discriminators execute in worst-case linear time.

The generic discriminators can be coded compactly using list comprehensions, with order and equivalence representations specified using Generalized Algebraic Data Types (GADTs). We give examples of the uses of discriminators, including most-significant-digit lexicographic sorting, type isomorphism with an associative-commutative operator, and database joins.

Since practical efficiency requires a thread-local bucket table reused by all same-thread discriminator calls we argue that built-in primitive types, notably pointers (references), should come with efficient discriminators since they, in contrast to binary comparison operations or hashing functions, facilitate the construction of discriminators for abstract types that are *both* highly efficient *and* representation independent.

Fusion in the Utrecht Haskell Compiler: Extended Abstract

Thomas Harper
tom.harper@cs.ox.ac.uk

Department of Computer Science
University of Oxford

Fusion is a popular technique for speeding up programs in Haskell. It allows programmers to program in a modular, compositional style without paying the performance cost introduced by intermediate data structures. This is often accomplished by writing libraries with functions that fuse when composed together. This allows the transformation to take place without any intervention from the client programmer. An elementary example of this is *map* fusion. This allows a program consisting of two *maps*

$$map\ (\lambda x \to x + 1) \circ map\ (\lambda y \to y * 2)$$

to be written as one *map* instead

$$map\ (\lambda x \to (x * 2) + 1)$$

In the Glasgow Haskell Compiler [5], this is accomplished using two kinds of transformations. The first is an *algebraic transformation*, which is stated as an equation where instances matching the left hand side are rewritten into instances matching the right hand side. For *map* fusion, the rule might be phrased as

$$\forall\ xs.\ map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$$

These are implemented as *rewrite rules* [7], which take the form of compiler pragmas that specify these transformations as above. The second kind of transformation is simply the normal optimisations that are applied during compilation [6], which inline f and g to create a single function, and then move on to even lower-level optimisations. The onus on the library writer is to write f and g such that they can be combined, which means they must be nonrecursive. The benefit of this optimisation is that it takes the burden off the user of the library, who does not have to worry about the fusion mechanics, and the library writer need not worry about the underlying transformation machinery that accomplishes it, provided they write their programs appropriately. Libraries can be written in such a way that all the functions of a library fuse with each other (i.e. not just *map*s with other *map*s). This method of implementing fusion underlies common fusion techniques, including *foldr/build* [3], *destroy/unfoldr* [8], and stream fusion [1].

Getting programs to fuse using such a system, however, is not always straightforward. First, they must be written in a certain form. Even then, the transformations are rather fragile; it is difficult to make any guarantees about whether

fusion will actually occur in a given program, because fusion results from the ad-hoc application of standard transformations. For example, the algebraic transformations seek instances of the left hand side, but there is guarantee that they will be found before they are inlined away, meaning that the opportunity for fusion is lost. This means that the library writer must result to using more pragmas, in this case to tune the compiler to choose which functions to inline and when. Even in this case, fusion is not guaranteed to occur due to the interaction of various transformations that could preemptively remove opportunities.

Currently, the Utrecht Haskell Compiler (UHC) [2] implements no inlining or case transformations, which are the basis for fusion using this method. We see this clean slate as an opportunity to implement a fusion infrastructure based on the experiences and challenges encountered by library writers. We start with the idea of that this method of fusion, which we call *shortcut fusion*, is a general program transformation of which each of the aforementioned techniques is a specific instantiation [4]. This formalisation provides uniform method for implementing shortcut fusion.

With this in mind, we have begun implementing the necessary features for shortcut fusion in UHC, where the focus is on simplicity in the frontend and robustness in the backend. In the frontend, the programmer uses keywords that allow him to declare which functions should be targeted for fusion. These fusion-specific declarations give the compiler more information about the programmer's specific intention, as opposed to using general-purpose pragmas. In the backend, small, simple transformations are composed together to fuse functions. However, the information gathered from the programmer's declarations is used to choose when and how to apply the necessary transformations, which can be focussed on inlining the necessary functions and while preserving others. We also limit the scope of transformations to the targeted functions, which allows us to be more aggressive with inlining without concerns for the rest of the program.

We present the results of the project so far. We review which local transformations are necessary for fusion and show how they can be composed to achieve a fusion algorithm. We describe what guarantees can be made using this approach, focussing on the robustness of the fusion and under what conditions and it be guaranteed to succeed, but also issues of correctness and termination of the algorithm. The performance aspect will be addressed using benchmarks. We will also present plans for future work. In particular, features are sought that will provide more feedback to the programmer about the outcome of attempted fusion, which would be useful both to verify successful fusion and to help the programmer debug situations in which fusion is meant to occur and but does not.

References

1. Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion. *ACM SIGPLAN Notices*, 42(9):315, October 2007.
2. Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09*, pages 93–104, New York, New York, USA, 2009. ACM Press.
3. Andrew Gill, John Launchbury, and Simon Peyton Jones. *A short cut to deforestation*. ACM Press, New York, New York, USA, 1993.
4. Thomas Harper. A library writer's guide to shortcut fusion. In *Proceedings of the 4th ACM symposium on Haskell - Haskell '11*, page 47, New York, New York, USA, 2011. ACM Press.
5. Simon Peyton Jones and Simon Marlow. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications, Volume II*, chapter 5. 2012.
6. Simon Peyton Jones and André L M Santos. Compilation by transformation in the Glasgow Haskell Compiler. Technical report, University of Glasgow, 1994.
7. Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, pages 203–233. ACM SIGPLAN, 2001.
8. Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming - ICFP '02*, volume 37, pages 124–132, New York, New York, USA, 2002. ACM Press.

OCaml-Java: from OCaml sources to Java bytecodes

Xavier Clerc

ocamljava@x9c.fr
http://www.ocamljava.org/

Abstract. This article presents the code generation scheme of the OCaml-Java compiler. The goal of the OCaml-Java project is to allow execution of OCaml programs on a Java Virtual Machine. In order to achieve decent performances, it is necessary to build a compiler producing optimized bytecode that will relies on an efficient support library at runtime.
The OCaml-Java project thus provides (*i*) an efficient runtime written in pure Java, and (*ii*) an optimizing compiler based on the original OCaml compilers for the front-end and on the Barista library for the back-end.

Keywords: OCaml, Java, bytecode, compiler, code generation

1 Introduction

The OCaml-Java project is presented at large in [1]; in the present article, we will focus on the code generation process as implemented in the OCaml-Java compiler. In the remainder of this section, we will nevertheless summarize the goals and state of the OCaml-Java project. Then, section 2 will expose the architecture of the various OCaml compilers. Section 3 will present the runtime representation of values in the different compilers, and section 4 will give an overview of the Barista library that is used as the compiler back-end. Section 5 shows examples of actual bytecode generation. Finally, sections 6 will discuss future work.

Why the JVM is an interesting target

The official OCaml distribution features both bytecode (for a dedicated virtual machine), and native compilers (for common architectures and OSes). It may seem at first sight that nothing more is needed, the former meeting portability needs and the latter meeting performance needs. However, being able to run OCaml code on a Java Virtual Machine is appealing for mainly two reasons:

- access to a larger choice of libraries;
- access to multicore programming.

The availability of libraries, which is essentially correlated with the size of a language community, is still a known weakness of the OCaml ecosystem in spite

of a vibrant community. Having the ability to run on a Java Virtual Machine gives access to all the libraries of the Java ecosystem. Its huge community has developed frameworks and tools for almost any purpose, and those can now be used by OCaml developers.
Indeed, to be able to use such Java libraries, it is not sufficient to be able to produce Java bytecode. It is also necessary to give to the OCaml developer means to manipulate Java elements from an OCaml program. For this reason, the OCaml-Java compiler features an extension of the typer to allow the construction and manipulation of Java instances from a pure OCaml program. More details regarding the extensions to the typer can be found in [1].

Multicore programming can be done in OCaml without resorting to compilation to Java bytecodes. However, the current implementation of OCaml is based on a global runtime lock allowing only one OCaml thread to run at a time. For this reason, leveraging multiple cores is often done through libraries using indeed multiple processes (most notably, map/reduce implementations such as [2] or [3]).
Another option is to modify the OCaml runtime to get rid of the global runtime lock. Such a modification implies of course to develop a parallel garbage collector (see for example [4]) and needs a lot of manpower, as well as some modifications to core OCaml libraries that are not reentrant. At the opposite, by targeting a Java Virtual Machine, we get a parallel garbage collector for free, and in addition can take advantage of Java standard libraries such as the fork/join framework to develop multicore OCaml programs based upon shared-memory.

Java 1.7 features for functional programming

The latest major release of the Java platform has brought a lot of exciting new features. Among them, two are particularly interesting when implementing functional languages:

- the `invokedynamic` framework;
- the *G1* garbage collector.

The `invokedynamic` framework is a very powerful addition to the Java platform as it allows a language implementor to define new semantics for method dispatch. In the OCaml-Java project, we in fact only use the method handles (which are akin to function pointers in C) provided by the framework in order to easily and efficiently implement closures.

The *G1* garbage collector is actually pretty important for functional language implementors because it is known to better suit the allocation/collection pattern found in functional programs. Such programs are typically allocating a lot of small and short-lived values while classical Java programs tend to put less pressure on the allocator.

Past and present of OCaml-Java

The 1.x versions of the OCaml-Java project should be regarded as mere proofs of concept, whose goal was to reach compatibility with the original implementation. The compatibility is almost total: all language constructs are supported and most library exhibit the same behavior (some minor differences are due to the fact that the Java Virtual Machine does not implement every POSIX primitives).

The 2.0 version keeps the same compatibility level, and proposes great improvements in both memory usage and performances. The goal is to be able to execute typical OCaml code on a Java Virtual Machine while remaining at worst two times slower than native code. The current prototype fulfills this objective on the majority of tested benchmarks.

2 Compilers architecture

Original compilers

The original OCaml distribution ships with two compilers: one producing bytecode for a dedicated virtual machine, and the other one producing native code. The bytecode compiler is available on every architecture while the native one is only available on the following:

- tier 1 (*i. e.* officially maintained): `amd64`, `ia32`, `powerpc`, and `arm` under Linux, MacOS X or Windows;
- tier 2 (*i. e.* unofficially maintained): `sparc`, and tier 1 architectures under BSD or Solaris flavors.

Both compilers naturally share a large codebase: parsing and typing are identical, thus relying on the very same code. Figure 1 shows the successive passes of both compilers from an implementation source file (*i. e.* a `.ml` file) to an implementation compiled file (*i. e.* a `.cmo` file for the bytecode compiler, and a `.cmx` file for the native compiler). We do not detail the compilation of interface source file because they (*i*) do not produce code, and (*ii*) are identical in both compilers.

Figure 1 presents the various passes from a source file to a binary file, as well as the different data structures used during the process. We only skip the passes that are just intended to optionally pretty-print the intermediate data structures on standard output to ease debugging. As previously stated, both compilers share the passes related to parsing (`Pparse.file`) and typing (`Typemod.type_implementation`). They also share the very first passes related to code generation: `Translmod.transl_implementation` and `Simplif.simplify_lambda`. These passes produces so-called *lambda code*, which is the most abstract representation of code to be executed.

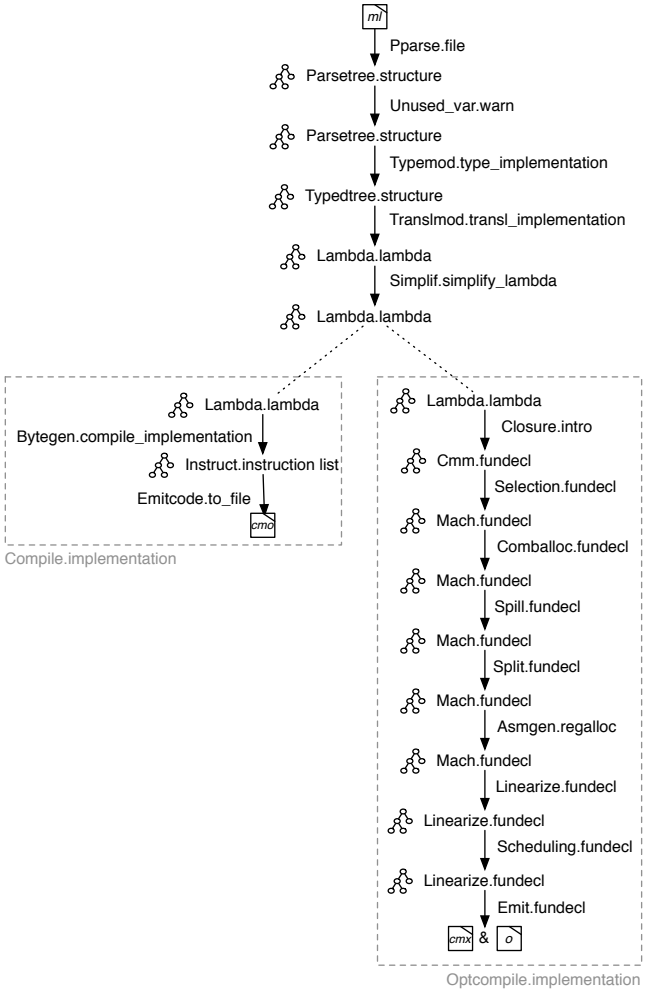


Fig. 1. Passes of OCaml compilers.

From this point, the two compilers diverge. The bytecode compiler only needs two more passes to produce its result; these passes are straightforward because the instruction set of the OCaml virtual machine was designed to provide the pieces allowing to almost execute *lambda code*. Of course, the native compiler has far more work to do because it has to accommodate an instruction set that was not specifically designed for functional programming, and has to target a register-based machine rather than a stack-based machine.

The first step, `Closure.intro`, handles the transformations associated with closures, uncurrification, and related optimizations. From this point, the code is represented by *machine code* which is an abstract representation that is still largely independent from the target platform, based on pseudo-instructions. The `Selection.fundecl` and `Comballoc.fundecl` are designed to perform the selection of pseudo-instructions for the code, and the optimization of allocations linked to a given block. Then, `Spill.fundecl`, `Split.fundecl`, and `Asmgen.regalloc` are responsible for actual register allocation, using information from the target platform. Finally, `Linearize.fundecl` reifies pseudo-instructions into actual lists of instructions, and `Scheduling.fundecl` optimizes the resulting order. The very last step is to output the assembly source code that will be used by an external assembler to produce object code.

OCaml-Java compiler

The OCaml-Java compiler can be seen as a third branch of the tree depicted by figure 1. This means that passes up to `Simplif.simplify_lambda` are shared with the original compilers. Figure 2 shows which transformations are then made on *lambda code*. First, very similarly to the native compiler, `Jclosure.jlambda_of_lambda` is responsible for the handling of closures, producing a slightly different and optimized *lambda code*. Then, `Macrogen.translate` decomposes operations from the *lambda code* into *macro instructions* that are not Java bytecode instructions but can be easily mapped to. This pass is also responsible for variable allocation which entails the choice of their actual representation, thus opening the possibility of value unboxing. Finally, `Bytecodegen.compile_function` produces actual Java bytecode using the Barista library (detailed at section 4).

The use of the Barista library provides several benefits: first, some typing discipline on instruction parameters is enforced (it would not be the case if we generated assembly source); second, the library is responsible for tedious boilerplate operations such as the computation of stack maps; third, the library provides some generic optimizations over produced bytecode. Such optimizations (removal of unnecessary operations, simplifications related to neutral or absorbing elements, strength reduction, *etc.*) allow the compiler to produce Java bytecode without having to reason on low-level optimization. As a consequence, the optimizations made by the compilers are only the ones related to the OCaml source code, not to the produced bytecode.

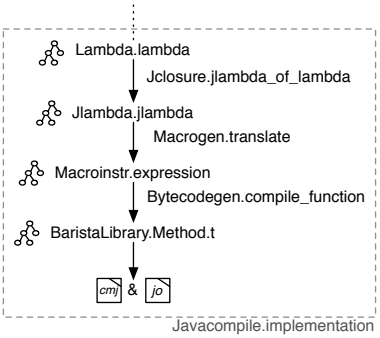


Fig. 2. Architecture of OCaml-Java compiler.

When compilation is done, two files are produced: a `.cmj` file corresponding to the `.cmx` file of the native compiler, and a `.jo` file corresponding to its `.o` file. The `.jo` file is actually a Java archive containing two entries:

- `Module.class` is the class file containing the implementation of all module functions as Java static methods;
- `Module.consts` is a binary file respecting the OCaml marshal format containing the (structured) constants used by the module.

A module is later linked to produce an executable jar file. At runtime, the initialization code for a module (located in its `entry` method) is responsible for the loading of the constants. The constants are then accessed through thread-local storage. This indirection is indeed necessary in order to allow several OCaml programs to run on the very same Java Virtual Machine.

3 Value representation

The compilation scheme of OCaml performs type erasure, meaning that *almost* every typing information is lost during the compilation process. This is of course not a problem as OCaml is statically and strongly typed, meaning that no type test has to be performed at runtime.

Basically, all the values share a common type, namely `value`. Having a common type for all values at runtime greatly simplifies the compilation process because such a common representation make polymorphism compilation trivial.

Precisely, use of the `value` type is mandatory at function boundaries (*i. e.* to call an OCaml function, or a C primitive), but a function is free to use whatever representation it prefers for local values. This freedom is indeed crucial in order to reach good performances because it allows unboxing of values. Values still

need to be boxed at function call, but this penalty can also be partially avoided by applying inlining.

In the remainder of this section, we first present the *de facto* specification of runtime values set by the original OCaml implementation, and then present how such a specification is implemented in OCaml-Java.

Original runtime

The various values manipulated at runtime by OCaml program can be specified by the following grammar.

value	::=	long unboxed value
	|	pointer to managed block
	|	pointer to unmanaged block

A long value is differentiated from a pointer value using tagging: the lowest bit is set to one for long values, while it is set to zero for pointer values. The encoding of an integer value i as a long unboxed value l is thus done according to the following equation: $l = (i \times 2) + 1$. A managed pointer (*i. e.* inside the OCaml heap) is discriminated from an unmanaged one (*i. e.* allocated by C code) by testing the actual address against the lower and higher bounds of the OCaml heap.

managed block	::=	tag $\oplus$ size $\oplus$ list of *size* blocks
	|	closure-tag $\oplus$ size $\oplus$ code pointer $\oplus$ list of *size - 1* blocks
	|	string-tag $\oplus$ size $\oplus$ array of *size* bytes
	|	double-tag $\oplus$ 64-bit float value
	|	double-array-tag $\oplus$ size $\oplus$ array of *size* 64-bit float value
	|	custom-tag $\oplus$ identifier $\oplus$ size $\oplus$ array of *size* bytes

As seen by the possible contents of a managed block, some typing information seems to be retained at runtime. However, this does not allow to recover the typing information present in the source, because several different types in the source can be mapped to the same runtime representation. Again, strong typing has been enforced at compile time, so no confusion could be made at runtime between values of different types.

OCaml-Java runtime

The representation of values is based on multiple classes for the various kinds of values. All classes inherit from a parent `Value` abstract class. This class implements the operations for all the kinds of values, possibly proposing a dummy or

failing implementation. It is then the responsibility of children classes to override that base implementation with a correct one. The guarantee that a dummy or failing implementation will never be called is based on the static and strong typing occurring at compile time.

Specialized classes are defined for long values, string values, double values, double array values, and block values. Contrary to the original runtime, all values even long ones are allocated because the Java Virtual Machine does not support tagged values. However, every creation of value has to be done through a factory method, which allows us to share values through a cache. As an example, long values are immutable and such a cache allows to share values between -128 and 255. These values are allocated once at program startup, and also allow to use reference comparison for values between the bounds.

The compilation scheme of OCaml will turn a type such as a record or a tuple of values into a mere block at runtime. Again, strong and static typing ensure that the program will not try to access to an element that does not exist (*e. g.* trying to access the third component of a couple). For this reason the original OCaml compilers will not generate code for testing such bounds. However, in Java it is not possible to remove bounds checks when accessing the elements of an array[1]. As a consequence, if the elements of a block are stored into an array, we will have to pay the price of a bound check at every access.

For this very reason, we resorted to what could be called *data inlining*. Rather than having only one class named `BasicBlockValue` storing its elements as one `Value[]` field, we define a bunch of classes named `BasicBlockValue`n that stores n elements as n `Value` fields. This allows to defines methods such as `get0()` that will return the first element of a value with no bound check. The same is done for double arrays and allows "small" tuples, record and all types sharing the same runtime representation to avoid bound checks when accessing the element at a given index.

Experimentation showed measurable speedups when growing the n value up to 8. The current version of the runtime hence contains classes with n ranging from 0 to 8. The source code for these classes is, of course, generated to avoid maintenance issues. Naturally, besides those classes, a `BasicBlockValue` (respectively a `DoubleArrayBlockValue`) is defined to be able to store an unbounded number of elements through an array. Then, array bound checks are no more avoided but experience indicates that this representation is indeed used for OCaml types that are arrays, and should test bounds at runtime for every access.

[1] The Hotspot compiler can remove such tests if it can *prove* that no illegal access will happen, but the developer can not request to remove such tests.

4 The Barista library

Overview

Barista [5], by the same author, is initially an OCaml library designed to load, construct, manipulate and save Java class files. The library supports the whole class file format as defined by Oracle (formerly Sun). Upon the library, a command-line utility (also named "barista") has been developed: both an assembler and a disassembler for the Java platform.

The assembler will turn an assembly source file into a class file to be run onto a Java Virtual Machine. The disassembler does the same work in the opposite direction: it takes the fully qualified name of a Java bytecode class file present in the classpath, and transforms it into an assembler source. Two other utilities allow to inspect the contents of a bytecode file: it is possible to just print the list of methods of a given class, and also to print the control flow of a given method as a graph. The following code sample shows how the canonical "Hello world" example can be coded in the Barista assembler:

```
.class public final pack.Test
.extends java.lang.Object

.method public static void main(java.lang.String[])
        getstatic java.lang.System.out : java.io.PrintStream
        ldc "hello world.\n"
        invokevirtual java.io.PrintStream.println(java.lang.String):void
```

One important feature of the library is that it provides two representations for the various Java element: a high-level representation, and a low-level one. The high-level representation is intended to be easily used by the developer, while the low-level one is intended to be as close as possible to the class file specification. Obviously, conversion functions between representations are provided and the developer is thus able to choose the representation level that suits her needs. Table 1 shows how Java elements are mapped to Barista types.

Table 1. Mapping of Java elements to Barista types.

Element	Low-level form	High-level form
Annotations	`Annotation.info`	`Annotation.t`
Attributes	`Attribute.into`	`Attribute.t`
Instructions	`ByteCode.t`	`Instruction.t`
Fields	`Field.info`	`Field.t`
Methods	`Method.info`	`Method.t`
Classes	`ClassFile.t`	`ClassDefinition.t`
Packages	`ClassFile.t`	`PackageDefinition.t`
Modules	`ClassFile.t`	`ModuleDefinition.t`

Hypergraph

Besides the two representation of instructions, namely types `ByteCode.t` and `Instruction.t`, that are basically lists of instructions, the code of a method can also be represented as a graph. Precisely, a method code can be represented as a rooted hypergraph. The rooted property stems from the fact that there is only one entry point for a given method. The hypergraph nature of the structure is indeed a design choice that allows to represent the conditionals by edges with one source and as many destinations as there are possible destinations.

The nodes of the hypergraph are labelled with instruction lists that contain no jump, jumps being represented by edges. Edges hence represent the control flow of the method and can be:

- classical edges with one source and one destination, in order to encode sequential execution (the edge is then with no label);
- three-legged edges with one source and two destinations, in order to encode a test and its two possible consequences (the edge is then labelled with the condition associated with the test);
- n-legged edges with one source and $n-1$ destinations, in order to encode switch instructions (the edge is then labelled with the definition of the switch, that is either a list of values or lower and upper bounds);
- *special* edges with one source and one destination, in order to indicate that the source is protected by a `try`/`catch` construct, the destination being the exception handler (the edge is then labelled with the class name of the exceptions that can be caught).

Given the hypergraph structure, there are two kinds of optimizations that can be performed by the Barista library:

- structural optimizations, modifying the hypergraph structure;
- non-structural optimizations, modifying only the labels of nodes.

In the first category, Barista currently features two optimizations: dead code elimination, and jump optimization. Dead code elimination removes all nodes that cannot possibly be reached from the root. Jump optimization short-circuits consecutive jumps with no bytecode between them, as shown by figure 3.

In the second category, Barista features several peephole optimizations that are performed independently on the hypergraph nodes. These include, among others:

- code size optimizations (*e. g.* replacing a *generic* instruction such as `aload` by a more compact `aload`n);
- removal of unnecessary load and/or store operations (*e. g.* if a loaded value is discarded or if a stored value is overwritten with no use);
- expression simplifications related to neutral or absorbing elements (*e. g.* addition to zero);
- basic strength reduction (*e. g.* shifting rather than multiplying when the multiplier is a power of 2).

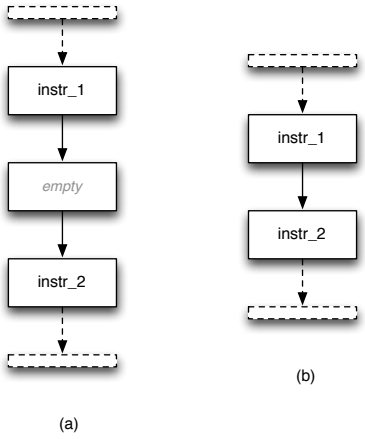


Fig. 3. Jump optimization: (a) before, and (b) after short-circuiting an empty node.

Example

As an example, we take the following Java static method, doing some computation over integer values:

```
public static int meth(final int x, final int y) {
  if (x > y) {
    try {
      return compute1(x);
    } catch (final Exception e) {
      return 0;
    }
  } else {
    return compute2(y);
  }
}
```

After compiling it with the `javac` compiler, we can dump its bytecode by invoking the `javap` utility, leading to the following output:

```
public static int meth(int, int);
  Code:
     0: iload_0
     1: iload_1
     2: if_icmple     13
     5: iload_0
     6: invokestatic  #2                  // Method compute1:(I)I
     9: ireturn
    10: astore_2
    11: iconst_0
```

```
   12: ireturn
   13: iload_1
   14: invokestatic  #4                  // Method compute2:(I)I
   17: ireturn
 Exception table:
    from    to  target type
        5     9    10   Class java/lang/Exception
```

Barista can be used to transform a method bytecode into an hypergraph by executing the `barista flow 'C.meth(int,int):int'` command where `C` is the class defining the method. The result is a graph representation in dot[2] format and is represented in figure 4.

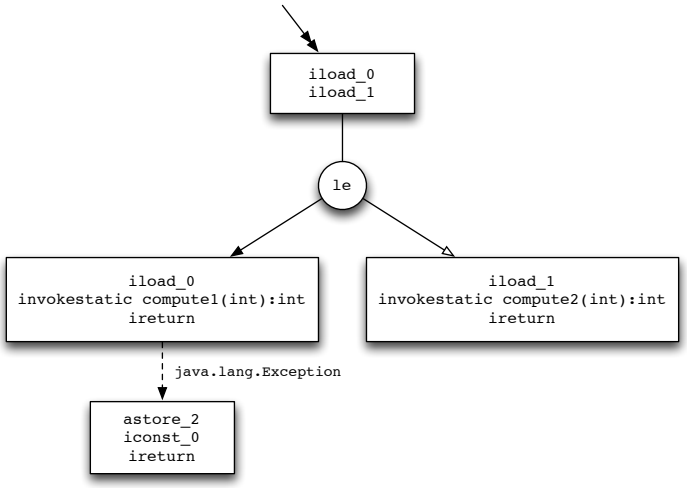


Fig. 4. Hypergraph for method `meth(int,int):int`.

Figure 4 features seven graph elements:

- four nodes (represented by boxes), containing the bytecode for the various code blocks (condition evaluation, if block, else block, and exception handler);
- a double arrow, indicating which node is the root;
- a dotted edge, from the protected node to the handler node and also labelled with the class of exceptions to be caught;
- an hyperedge, linking three nodes: (*i*) the block evaluating the condition, (*ii*) the block to execute next if condition is true, (*iii*) the block to execute next

[2] See `http://www.graphviz.org/`.

if condition is false; the hyperarc is also labelled with the kind of condition to perform.

5 Examples of bytecode generation

Tail call optimization

Our first example is a very classical one: the recursive computation of a list length using an accumulator parameter in order to make the recursive call terminal. The left column shows the OCaml code of the function, while the right one shows the generated bytecode.

```
let rec length acc = function
  | [] -> acc
  | _ :: tl -> length (succ acc) tl
```

```
 0: aload_1
 1: invokevirtual Value.asLong:()long
 4: l2i
 5: ifeq 27
 8: aload_0
 9: invokevirtual Value.getRawValue:()long
12: invokestatic NativeArithmetic.incrint:(long)long
15: invokestatic Value.createFromRawLong:(long)Value
18: aload_1
19: invokevirtual Value.get1:()Value
22: astore_1
23: astore_0
24: goto 0
27: aload_0
28: areturn
```

Instructions at offsets $0-5$ load the second parameter of the function to test whether it equals 0 (which is used to represent an empty list). The first branch of the matching, at offset 27, only loads the first parameter (*i. e.* the accumulator) in order to return its value to the caller.

The second branch of the matching perform the following operations:

offsets $9-15$ the `acc` parameter is loaded and incremented;
offsets $18-19$ the value of the second parameter is loaded, and its second component (*i. e.* the tail of the list) is extracted;
offset 22 the new value for the second parameter overwrite its previous one;
offset 23 the new value for the `acc` overwrite its previous one;
offset 24 as the new values of the parameters have replaced the previous ones, the recursive call can be safely translated as a bare jump to the entry point of the method.

Unfortunately, it is not possible to optimize every tail call using the presented scheme, as it is not allowed in Java to jump into the body of another method. The only way to transfer the execution to another method in Java is to

actually execute an `invokeXYZ` instruction, which involves placing parameters on the stack.

For this reason, most implementors of functional languages on the Java Virtual Machine only optimize tail calls when they turn to be direct recursive calls. In this respect, OCaml-Java does the same as, for example, Clojure [6] or Scala [7]. Of course, it would be possible to apply transformations to a program in order to overcome the Java restrictions over jumps (*e. g.* trampolines, or CPS could be used). However, such solutions do not only make compilation more complex, they also tend to exhibit other problems in terms of performances and/or code size.

Value unboxing

Our second example has been designed to show how the unboxing of values allows to reach good performances in the case of numerical code. The left column shows the OCaml code of the complete function, while the right one shows the generated bytecode for the loop body.

```
let float () =                          (...)
  let x = ref 1. in                     33: dload 5
  let y = ref 2. in                     35: dload_1
  let acc = ref 0. in                   36: dload_3
  for i = 1 to 1_000_000_000 do         37: dmul
    acc := !acc +. (!x *. !y);          38: dadd
    x := !x +. 1.;                      39: dstore 5
    y := !y *. 2.                       41: dload_1
  done;                                 42: dconst_1
  !acc                                  43: dadd
                                        44: dstore_1
                                        45: dload_3
                                        46: ldc2_w 2.0d
                                        49: dmul
                                        50: dstore_3
                                        (...)
```

Variables `x`, `y`, and `acc` are respectively stored at local indexes 1, 3, and 5. The compiler has determined from their initial values that they are double values. Instructions at offset $33-39$ computes the expression `!acc +. (!x *. !y)` and store its value back. Then, instructions $41-44$ update the value of the `x` variable, and instructions $45-50$ update the value of the `y` variable.
It is obvious from the instructions that all operations are done using the Java `double` primitive type, no boxing being made at all. This ensures that we get the best possible performances, and also avoid to put any pressure on the memory allocator and garbage collector.

When comparing the performances of the original OCaml compiler to the OCaml-Java compiler, we measured the code generated by the former to take 3.8 seconds and the code generated by the latter to take 5.6 seconds. Then, we changed the upper bound of the loop by multiplying it by ten, and then measured times to be respectively 38.6 seconds and 48.0 seconds. This means that in the second setting, OCaml-Java in less than 25% slower than original OCaml. Of course, the ratios are better when measuring longer runs because virtual machine startup and just-in-time compiling are amortized.

Even better, the OCaml-Java compiler is on par with the original one on benchmarks such as `nbody` from the Language Shootout[3]. However, generally speaking, it is safe to state that OCaml-Java is currently between two and three times slower than OCaml on average.

6 Future Work

Most of our short-term effort will be focused on the unboxing of values. It proved to produce large speedups in the past, and a lot of things can be done to make it more aggressive. First, currently, the kind of storage is chosen according to the initial value of a variable; we could design an heuristic also based on the uses of the variable.
Second, as previously said, boxing is mandatory at function boundaries; there are two ways to lift this restriction: (*i*) avoid such a boundary (*e. g.* by using inlining) or (*ii*) allow the compilation to functions taking unboxed parameters when typing information allows to do so. Also, unboxing is currently done only for the following OCaml types: `int`, `int32`, `int64`, `nativeint`, and `float`. It could also be done on others types, particularly ones constructed (*e. g.* records with mutable fields) over those that can already be unboxed.

Inlining itself can also be greatly improved. For example, the current version of the compiler is unable to inline recursive functions. This seems like a reasonable limitation at first, but some recursive functions can be tail-call optimized and thus be compiled as mere loops. In this case, it would be possible to inline such functions.

Another area we should definitely investigate is the possible influence of garbage collection parameters over performances. It would have had little sense for the examples presented in this paper, but we expect performances to be sensitive to garbage collector parameters in real-world applications. Indeed, the default parameters are chosen to allow good performances for typical Java applications, not OCaml ones.

Finally, we could also optimize compile-time performances by generating the Barista hypergraph directly during code generation. Currently, the compiler pro-

[3] `http://shootout.alioth.debian.org/`

duces plain bytecode that is then passed to Barista for low-level optimizations. This incurs the price of hypergraph construction from a list of bytecode instructions, which can be avoided.

To conclude, some words about optimization opportunities that are linked to the future development of the Java platform. Among those considered for inclusion in the next revision of Java, two would be particularly useful to functional languages targeting the Java Virtual Machine. The first feature is tagged values, and would allow us to avoid boxing of `int` values: it would not only allow faster operations but would also relieve the pressure over garbage collection by avoiding allocation.
The second feature is support for tail calls, and would allow us to mark a method call as terminal to indicate to the *just-in-time* compiler that a call can be optimized. It would allow, of course, faster execution, but would especially make the life of users easier because the absence of tail call optimization interacts with semantics when calls come to blow up the stack.

References

1. Clerc, X.: OCaml-Java: OCaml on the JVM. Trends in Functional Programming (2012)
2. Danelutto, M., Di Cosmo, R.: Parmap: minimalistic library for multicore programming. https://gitorious.org/parmap
3. Stolpmann, G.: Plama: Map/Reduce and distributed filesystem. http://plasma.camlcity.org/
4. Chailloux, E., Canou, B., Wang, P.: OCaml for Multicore Architectures. http://www.algo-prog.info/ocmc/web/
5. Clerc, X.: The Barista library. http://barista.x9c.fr
6. Hickey, R.: The clojure programming language. In: Proceedings of the 2008 symposium on Dynamic languages. DLS '08, New York, NY, USA, ACM (2008) 1:1–1:1
7. Odersky, M., et al.: The Scala Language. http://www.scala-lang.org/

The HERMIT in the Tree

Mechanizing Program Transformations in the GHC Core Language

Neil Sculthorpe, Andrew Farmer, and Andy Gill

Information and Telecommunication Technology Center
The University of Kansas
{neil,afarmer,andygill}@ittc.ku.edu

Abstract. There are many program transformation techniques that have been described in theory but have not been mechanized — either because they are too specialized to include in a general-purpose compiler, or because the developers' interest is in theory rather than implementation. This is unfortunate, as the mechanization process can often reveal obstacles that are glossed over in pen-and-paper proofs, yet need to be addressed before the transformation can be used in practice. In this paper, we describe our experiences of using the HERMIT toolkit to apply known transformations to the Glasgow Haskell Compiler's core language.

Keywords: GHC, mechanization, transformation, worker/wrapper

1 Introduction

There are a wide variety of transformation techniques for optimizing functional programs [19, 23, 20, 1]. Many such transformations have been implemented, and many are used by modern compilers. However, there are also techniques that have been described on paper but not mechanized, either because the transformation is too specialized to include as an optimization in a general-purpose compiler, or because the developers' interest is in theory rather than implementation.

We believe there is a lot to be gained from mechanizing program transformations. The mechanization process can often reveal obstacles that do not appear in pen-and-paper proofs, either because of implementation-specific details, or because the pen-and-paper proofs gloss over details that may seem obvious to a human, but are less obvious to a machine. And sometimes, mechanization can find genuine errors in pen-and-paper proofs [15, 6].

HERMIT (Haskell Equational Reasoning Model-to-Implementation Tunnel) is a recently implemented plugin for the Glasgow Haskell Compiler (GHC) [7] that provides an interactive interface for applying transformations directly to GHC's internal intermediate language [5]. This plugin is part of a larger HERMIT toolkit, a Haskell framework which is being developed with the aims of supporting equational reasoning and allowing custom optimizations to be applied without modifying either GHC or the Haskell source code.

In this paper we report on our experience of using HERMIT to mechanize optimization techniques, using the concatenate vanishes transformation [29], tupling transformation [18] and the worker/wrapper transformation [9, 25] as case studies. The main contributions of this work are:

- We show that it is viable to mechanize theoretical transformations in the setting of the industrial-strength Glasgow Haskell Compiler. We report on our experiences, the obstacles that arose during mechanization, and our approaches to overcoming them. (§2, 4)
- We show that the HERMIT system is sufficiently mature to be able to encode and apply these transformation techniques. Additionally, we show that it is straightforward to augment the HERMIT framework to add new specialized transformations as needed. (§4, 5)
- In the process of abstracting the common patterns of our examples, we discover that concatenate vanishes and tupling transformation are special cases of the worker/wrapper transformation. (§3)

2 HERMIT

This section briefly overviews the HERMIT toolkit; for more details consult [5].

2.1 GHC Core

GHC recently added support for custom compiler plugins that can be inserted amid GHC's optimization passes [7]. HERMIT uses this mechanism to provide a transformation system for GHC Core, GHC's internal intermediate language.

GHC Core is an implementation of System $F_C^{\uparrow}$ [27], which is System F [22] extended with let-binding, constructors, and support for type coercions. Types are explicitly passed as arguments, but never returned. Fig. 1 gives the Haskell syntax for GHC Core, presented using some inlined type synonyms for clarity.

2.2 User Interface

HERMIT provides several interfaces at different levels of abstraction. In this paper we will use just one of those interfaces: a read-eval-print loop (REPL).

The REPL allows navigation over a GHC Core abstract syntax tree (AST), displaying the current sub-tree via a choice of pretty printers. The REPL provides a statically typed monomorphic functional language with overloading. Most commands return a *rewrite* from AST to AST, and the result of executing such a command is the newly transformed AST. Older versions of the AST are maintained, and it is possible to step back and forth through the history of ASTs, or create branches to explore alternative transformation sequences. That is, HERMIT provides a version-control tree, where each node of the tree is an AST. When the user has finished applying transformations, she selects one of the ASTs for GHC to compile. See §5 for an extended example using the REPL.

```
data ModGuts  = ModGuts { _ :: [CoreBind], ... }
data CoreBind = NonRec Id CoreExpr | Rec [(Id, CoreExpr)]
data CoreExpr = Var Id
              | Lit Literal
              | App CoreExpr CoreExpr
              | Lam Id CoreExpr
              | Let CoreBind CoreExpr
              | Case CoreExpr Id Type [CoreAlt]
              | Cast CoreExpr Coercion
              | Tick (Tickish Id) CoreExpr
              | Type Type
              | Coercion Coercion
type CoreAlt  = (AltCon, [Id], CoreExpr)
data AltCon   = DataAlt DataCon | LitAlt Literal | DEFAULT
```

Fig. 1: GHC Core.

2.3 Extendability

HERMIT is designed to make the addition of new transformations straightforward. There are two ways of doing this, first by adding a new internal primitive, and secondly by leveraging the GHC RULES mechanism [21].

The former requires modifying the HERMIT source code, but our experience has been that typically they can be constructed fairly easily out of the large suite of low-level congruence combinators and strategic traversals already provided by HERMIT (see [5]). We discuss this further in §4.

The latter allows Haskell programmers to annotate their source files with directed rewrite rules. These rules are type checked, but otherwise GHC provides no guarantee as to their correctness. HERMIT exposes any such rules as rewrite commands, allowing the user to selectively apply them as desired. This provides a lightweight mechanism for allowing the user to add custom transformations to HERMIT, albeit limited to those that can be expressed by GHC RULES.

We used both methods extensively while mechanizing the examples discussed in §3. In particular, we needed to add several primitive transformations that were not obviously useful until we started mechanizing examples (see §4, 5).

3 Transformations for Mechanization

After selecting our three program transformation techniques, we chose several representative examples of each from the literature. Of about a dozen examples considered, those we have successfully mechanized in HERMIT thus far are:

- Concatenate Vanishes: Flatten [29, 13], Quicksort [29], Reverse [29, 13]
- Tupling Transformation: Fibonacci [4, 25]
- Worker/Wrapper: Last [25], Reverse [9, 5], Stream Memoization [9]

This section overviews the three techniques, and then discusses how concatenate vanishes and tupling can be viewed as special cases of worker/wrapper.

3.1 Concatenate Vanishes

The concatenate vanishes transformation (CV) [29] is a technique for increasing the efficiency of programs that make repeated use of list concatenation. Here we briefly summarize the key aspects of the transformation; for full details consult the original work by Wadler [29].

Consider the standard definition of list concatenation:

$$
\begin{array}{l}
(+\!\!+) :: [a] \to [a] \to [a] \\
[\,] \quad\;\; +\!\!+ bs = bs \\
(a : as) +\!\!+ bs = a : (as +\!\!+ bs)
\end{array}
$$

The time complexity of this definition is linear in the length of its first argument, but constant in the length of its second argument. Thus, while $+\!\!+$ is associative, $(as +\!\!+ bs) +\!\!+ cs$ will evaluate less efficiently than $as +\!\!+ (bs +\!\!+ cs)$. The essence of CV is to exploit this observation to restructure programs using repeated concatenation into a more efficient form.

The transformation is as follows. Given a function that returns a list[1]

$$
\begin{array}{l}
f :: a \to [b] \\
f\ a = expr
\end{array}
$$

define a new function that returns a list-to-list function (known as an H-list [12])

$$
\begin{array}{l}
f' :: a \to [b] \to [b] \\
f'\ a\ bs = expr +\!\!+ bs
\end{array}
$$

where *expr* is an expression that may contain f and a. Then redefine the original function f as:

$$
\begin{array}{l}
f :: a \to [b] \\
f\ a = f'\ a\ [\,]
\end{array}
$$

The efficiency gains (if any are possible) are then achieved through refactoring the definition of f': first by applying the associativity and unit laws of $+\!\!+$, and then by folding the definition of f' to eliminate any recursive calls to f.

3.2 Tupling Transformation

The tupling transformation (TT) [18] applies to functions that make duplicated recursive calls. The idea is to create a linearly recursive function that returns multiple results as a tuple, where the additional results are the values that would have been unnecessarily recomputed, but now can be reused. At the top level, a combining function computes the final result from the tuple components.

To illustrate this, consider the call dependency tree (Fig. 2a) for a naively defined Fibonacci function. Notice the duplicated calls to *fib* with the same

[1] For clarity of presentation we assume the function is in uncurried form, but CV is valid for functions that take any number of arguments; see [29].

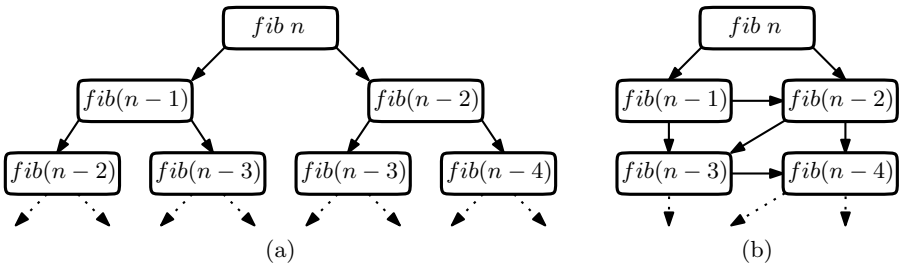


Fig. 2: Dependency graphs for *fib*, illustrating redundant calls.

arguments. Ideally we want to combine these calls to produce the dependency *graph* shown in Fig. 2b.

Informally, we tuple nodes that depend on the same recursive call. Thus, we tuple the results of *fib* n and *fib*$(n-1)$, which both depend on *fib*$(n-2)$, and *fib*$(n-1)$ and *fib*$(n-2)$, which both depend on *fib*$(n-3)$, and so forth. The task is to calculate, from the existing definition of *fib*, a recursive function t that computes each tuple from the subsequent tuple. Finally, the *fib* function is redefined in terms of t and the combining function. We perform this calculation in §5.

3.3 Worker/Wrapper Transformation

The worker/wrapper transformation (WW) [9, 25] is a technique for improving the efficiency of a recursive program by changing the data type being operated on. The basic idea is that given a program $prog :: a$, we factorize it into a more efficient *worker* program $work :: b$, and a *wrapper* function $wrap :: b \to a$ that converts the result into a value of the original type.

The first step is to rewrite the program as the fixed point of a non-recursive function f (where *expr* is an expression that may call *prog*):

$$prog = expr \qquad \Rightarrow \qquad prog = fix\ f\ \textbf{where}\ f = \lambda prog \to expr$$

Next comes the key step: choosing a more efficient data type. Once chosen, we define conversion functions between the two types:

$$\begin{aligned} unwrap &:: a \to b \\ wrap &:: b \to a \end{aligned}$$

These conversion functions are required to satisfy the property that

$$fix\ (wrap \circ unwrap \circ f) \equiv fix\ f$$

and often satisfy the stronger property $wrap \circ unwrap \equiv id$. It is then valid to redefine the original program as follows (known as WW factorization):

$$prog = wrap\ work$$

The definition of *work* can be derived in a number of ways [25]. Typically, we start from either $work = fix\ (unwrap \circ f \circ wrap)$ or $work = unwrap\ prog$, and then simplify the definition using any laws specific to the types a and b.

3.4 Concatenate Vanishes is an instance of Worker/Wrapper

CV can be expressed as an instance WW, where the original data type is a function returning a list, and the more efficient data type is a function returning an H-list. Thus we can verify the correctness of CV within the WW framework.

We start with a recursively defined function that returns a list:

$$
\begin{aligned}
&prog :: a \to [b] \\
&prog\ a = expr
\end{aligned}
$$

The first step of WW is to redefine *prog* as $fix\ f$. However, we won't need the definition of f for this particular derivation, so we omit this step and proceed to choose the type of *work*, which is a function returning an H-list:

$$work :: a \to [b] \to [b]$$

Next we define conversion functions between the two types:

$$
\begin{aligned}
&wrap :: (a \to [b] \to [b]) \to a \to [b] \\
&wrap\ h\ a = h\ a\ [\,] \\
&unwrap :: (a \to [b]) \to a \to [b] \to [b] \\
&unwrap\ h\ a\ bs = h\ a +\!\!+ bs
\end{aligned}
$$

It is straightforward to show that the WW precondition holds:

$$
\begin{aligned}
&\ \ wrap \circ unwrap \equiv id \\
\Leftrightarrow\quad &\{\ \text{extensionality}\ \} \\
&\ \ wrap\ (unwrap\ h)\ a \equiv id\ h\ a \\
\Leftrightarrow\quad &\{\ \text{unfold } wrap \text{ and } id\ \} \\
&\ \ unwrap\ h\ a\ [\,] \equiv h\ a \\
\Leftrightarrow\quad &\{\ \text{unfold } unwrap\ \} \\
&\ \ h\ a +\!\!+ [\,] \equiv h\ a \\
\Leftrightarrow\quad &\{\ [\,] \text{ is the unit of } +\!\!+\ \} \\
&\ \ True
\end{aligned}
$$

Applying WW factorization produces the following definitions:

$$
\begin{aligned}
&prog\ \ :: a \to [b] \\
&prog\ \ = wrap\ work \\
&work\ :: a \to [b] \to [b] \\
&work = unwrap\ prog
\end{aligned}
$$

η-expanding and unfolding *wrap* and *unwrap* gives:

$$
\begin{aligned}
&prog :: a \to [b] \\
&prog\ a = work\ a\ [\,] \\
&work :: a \to [b] \to [b] \\
&work\ a\ bs = prog\ a +\!\!+ bs
\end{aligned}
$$

Finally, inlining *prog* in the definition of *work* gives the CV transformation:

$$
\begin{aligned}
&work :: a \to [b] \to [b] \\
&work\ a\ bs = expr +\!\!+ bs
\end{aligned}
$$

3.5 Tupling Transformation is an instance of Worker/Wrapper

Less formally, we can observe that the TT is an instance of WW by noting that the linearly recursive function t (see §3.2) is the more efficient worker function, while the combining function is the wrapper. The unwrapping function is constructed by tupling multiple function calls on different arguments. The definitions of *wrap* and *unwrap* derived in this manner typically satisfy the strongest WW precondition. We demonstrate this in §5.

4 User Experience

This section discusses our experiences of using HERMIT to mechanize the WW and CV transformations, as well as some general experiences that aren't tied to a specific technique. We postpone discussion of TT until §5, where it serves as our more detailed case study.

4.1 Worker/Wrapper

WW was the first transformation that we mechanized. Introducing *fix*, the first step of WW, was not a transformation originally provided by HERMIT, nor was it definable in terms of other HERMIT commands. However, making use of the existing HERMIT infrastructure, we found it straightforward to add a new rewrite for this task. We did not need to add a rewrite to eliminate *fix*, as that can be achieved by using HERMIT's existing `unfold` command.

We chose to encode WW factorization using GHC RULES. Thus no modification to HERMIT was required, we just included the following pragma in the source code of each example, along with appropriate *wrap* and *unwrap* functions:

$$\{\text{-\# RULES} \quad \texttt{"ww"} \quad \forall f. \quad \mathit{fix}\ f = \mathit{wrap}\ (\mathit{fix}\ (\mathit{unwrap} \circ f \circ \mathit{wrap}))\ \text{\#-}\}$$

This works, but in the future we plan to add a HERMIT command that takes *wrap* and *unwrap* functions as parameters, thereby avoiding the need to repeat this rule for every specific *wrap* and *unwrap*.

HERMIT does not yet have a mechanism for checking preconditions, so it is up to the user to ensure that factorization is used only when the WW precondition holds. This is not ideal, and providing some mechanism within HERMIT for verifying pre-conditions, or at least for recording which pre-conditions have been assumed during the transformation, is an obvious next step in its development.

Another issue is that, unlike in a Haskell source file, the top-level bindings are not treated as a mutually recursive group. During type checking (before generating GHC Core), a dependency analysis separates the bindings into minimal recursive groups and orders these groups by their dependencies [14]. This can be problematic when applying GHC RULES, as some of the identifiers in the rule may not be in scope. To address this, we added a `flatten-module` rewrite that combines the top-level binding groups into a single recursive group, thereby ensuring all identifiers that can appear in a rule will be in scope.

Other than these issues, we found mechanizing the WW examples to be straightforward uses of HERMIT's basic transformations and GHC RULES. A detailed walk-through of the Reverse example can be found in [5].

We also encountered some unexpected behavior involving type-level universal quantification. GHC Core passes around type arguments explicitly; thus when a call is made to a polymorphic function, the type argument has to be provided. For example, the Core generated from *last* has the following type and structure:

$$
\begin{array}{l}
last :: \forall\, \tau\,.\,[\tau] \to \tau \\
last = \lambda\, \tau\ as \to ...last\ \tau\ ...
\end{array}
$$

However, we discovered that if the type signature for a top-level polymorphic function is omitted in the source code, GHC generates different Core. Specifically, it produces an outer non-recursive polymorphic function, and an inner recursive monomorphic function. That is, the type is fixed outside the recursion, avoiding the need to provide the type as an argument to each recursive call.

$$
\begin{array}{l}
last :: \forall\, \tau\,.\,[\tau] \to \tau \\
last = \lambda\, \tau \to \textbf{let}\ last :: [\tau] \to \tau \\
\qquad\qquad\qquad\qquad\ \ last = \lambda\ as \to ...last\ ... \\
\qquad\qquad\quad \textbf{in}\ last
\end{array}
$$

This difference, which is not noticeable at the level of Haskell source code, is significant enough to allow a GHC rule to fire in one case and not another. For example, WW factorization only fires for monomorphic functions, not polymorphic ones. In our opinion, HERMIT's ability to interactively display information on selected fragments of GHC Core was most helpful in understanding why the rule was not firing. Indeed, one potential application of the HERMIT system is experimenting with and debugging GHC RULES.

4.2 Concatenate Vanishes

Mechanizing CV proved straightforward. As shown in §3.4, the first step can be expressed as WW factorization, so we were able to proceed in the same manner as in §4.1. Mechanizing Flatten and QuickSort proved very similar to the Reverse example, with only a few differences in the basic transformations applied to simplify the resultant worker function. It was not necessary to add any new functionality to HERMIT.

Encouraged by the similarity of the three HERMIT scripts, we wrote a single generic script that works for all three examples, using HERMIT's higher-level commands. For this we did need to add a new command to HERMIT. The issue was that case floating (taking a function applied to a case expression and applying it to each case alternative instead) is only valid if the function is strict:

$$
\begin{array}{lcl}
f\ (\textbf{case}\ x\ \textbf{of} & & \textbf{case}\ x\ \textbf{of} \\
\quad a_1 \to e_1 & & \quad a_1 \to f\ e_1 \\
\quad a_2 \to e_2 & \Rightarrow & \quad a_2 \to f\ e_2 \\
\quad ... & & \quad ... \\
\quad a_n \to e_n) & & \quad a_n \to f\ e_n
\end{array}
$$

As HERMIT lacks a mechanism for verifying preconditions (see §4.1), it is the user's responsibility to ensure case floating is only applied to strict functions. This was fine when considering each example in isolation, as we explicitly stated when and where to float a case. But as this differed between examples, the usage in the generic script was potentially unsafe. To address this, we added a command that floats case (and let) expressions, but only past a function that it takes as a parameter. Again, adding this was straightforward.

Our generic CV script makes heavy use of GHC RULES, which encode the monoid laws for $((+\!\!+), [\,])$ and $((\circ), id)$, and a monoid homomorphism between them. We also used a rule to encode the fusion law relating the conversion functions between the list and H-list types [9]. This rule also has a precondition, and currently its usage in the generic script is unsafe in general (although in each specific example it is used safely). We are working on adding a rewrite to HERMIT that will allow us to restrict this rule to situations where the precondition is met, in a similar manner to the case-floating previously discussed.

Note that we do not claim that our generic script would work for any CV example; indeed we are quite confident it would not. Its purpose was just to test how well HERMIT copes with abstracting from multiple similar examples. HERMIT is designed as an interactive system where transformations are user-guided; we do not aim nor expect to be able to fully automate transformations in general. What we do aim for is to make HERMIT commands as robust as possible, in an effort to minimize the changes required if the source code changes, and more abstract commands help in this regard.

4.3 General Experiences

We found that we often followed inlining with the general-purpose clean-up command `bash` [5], which was serving as a crude way of unfolding [4] a definition. However, in some cases this was adversely affecting the content of the inline function, as well as its arguments. After consideration, we settled on the following HERMIT terminology, and introduced corresponding command support:

- To *inline*: to replace a value with its definition.
- To *apply*: to inline in the context of (zero or more) arguments, and perform beta-reduction (to let binding) on *all* the arguments.
- To *unfold*: to apply, then *attempt* safe/cheap substitution on all the new let bindings introduced by the apply.

Building on this terminology, we found the recursive deep-traversal combinators [5] insufficient for our needs: specifically, it was difficult to include as many arguments as possible when unfolding curried functions, while at the same time ensuring termination of unfolding. To address this, we invented a traversal strategy called `any-call`, which has the order of visiting nodes specifically tuned to maximize the number of arguments provided to any inlineable value, as well as traversing any arguments before performing the apply/unfold.

We found debugging transformation scripts to be challenging. To make it easier, we added two folklore combinators to HERMIT. The first, `trace`, identifies if a specified rewrite was attempted, and counts how many times. The second, `observe`, is similar, but also emits the node at which it was attempted. We have found both useful in practice when debugging our scripts, especially when understanding the usage of our deep higher-order combinators.

5 Example: Fibonacci Tupling

In this section we demonstrate the mechanization process in detail by performing TT on the Fibonacci function using the HERMIT REPL. Starting with the clear but inefficient (exponential time) definition over Peano naturals

$$
\begin{aligned}
&\textbf{data}\ \mathit{Nat} = \mathsf{Z} \mid \mathsf{S}\ \mathit{Nat}\\
&\mathit{fib} :: \mathit{Nat} \to \mathit{Nat}\\
&\mathit{fib}\ \mathsf{Z} = \mathsf{Z}\\
&\mathit{fib}\ (\mathsf{S}\ \mathsf{Z}) = \mathsf{S}\ \mathsf{Z}\\
&\mathit{fib}\ (\mathsf{S}\ (\mathsf{S}\ n)) = \mathit{fib}\ (\mathsf{S}\ n) + \mathit{fib}\ n
\end{aligned}
$$

we transform it into the following efficient (linear time) definition:

$$
\begin{aligned}
&\mathit{fib}' :: \mathit{Nat} \to \mathit{Nat}\\
&\mathit{fib}'\ n = \mathit{fst}\ (\mathit{work}\ n)\\
&\qquad \textbf{where}\ \mathit{work} :: \mathit{Nat} \to (\mathit{Nat}, \mathit{Nat})\\
&\qquad\qquad \mathit{work}\ \mathsf{Z} = (\mathsf{Z}, \mathsf{S}\ \mathsf{Z})\\
&\qquad\qquad \mathit{work}\ (\mathsf{S}\ m) = \textbf{let}\ (x, y) = \mathit{work}\ m\ \textbf{in}\ (y, x + y)
\end{aligned}
$$

As noted in §3.5, TT is an instance of WW, so we will make use of our existing WW infrastructure. Following [25], we choose the more efficient data type to be a function that returns a tuple of consecutive Fibonacci numbers, and define *wrap* and *unwrap* as follows:

$$
\begin{aligned}
&\mathit{wrap} :: (\mathit{Nat} \to (\mathit{Nat}, \mathit{Nat})) \to \mathit{Nat} \to \mathit{Nat}\\
&\mathit{wrap}\ h = \mathit{fst} \circ h\\
&\mathit{unwrap} :: (\mathit{Nat} \to \mathit{Nat}) \to \mathit{Nat} \to (\mathit{Nat}, \mathit{Nat})\\
&\mathit{unwrap}\ h\ n = (h\ n, h\ (\mathsf{S}\ n))
\end{aligned}
$$

Trivially, the $\mathit{wrap} \circ \mathit{unwrap} \equiv \mathit{id}$ precondition holds.

Loading the original definition of *fib* into HERMIT, we see the GHC Core that is generated:

```
hermit> set-pp-expr-type Show ; flatten-module ; consider 'fib

rec fib = λ ds → case ds of wild
                   Z → Z
                   S ds → case ds of wild
                            Z → S Z
                            S n → (+) (fib (S n)) (fib n)
```

We need to introduce the fixed-point operator, perform WW factorization, and then eliminate the fixed-point operator to return to a recursive definition. As this is a common sequence of steps (which we call the *WW split*), we have written a HERMIT script for this purpose. The script takes a recursively defined function of the form $g = expr$, and transforms it into a non-recursive call to a wrapped worker:

$$g = \textbf{let } f = \lambda g \rightarrow expr \\ \quad \textbf{in } (\textbf{let } work = unwrap\ (f\ (wrap\ work))\ \textbf{in } wrap\ work)$$

The worker is recursive, making use of the original body by coercing to and from the original computation type. The body of f is kept separate from the body of *work* by a let binding, allowing us to selectively inline f when required, but otherwise keep the details of the original computation abstract. The script relies on WW factorization being available as a GHC RULE (see §4.1).

```
hermit> load "WWSplitTactic.hss"
```

```
fib = let f = λ fib ds → case ds of wild
                           Z → Z
                           S ds → case ds of wild
                                    Z → S Z
                                    S n → (+) (fib (S n)) (fib n)
          rec work = unwrap (f (wrap work))
      in wrap work
```

This derivation will use the fold/unfold equational-reasoning technique [4]. When using fold/unfold, it is common to need access to *past* definitions of functions; a non-issue when working on paper (one simply looks up the page), but one that we needed to address. While the HERMIT kernel maintains a record of every version of the AST, we found it preferable to provide a command `remember` that explicitly saves a definition, rather than dig through the kernel's history. This also allows fold/unfold to be a lower-level notion that does not assume the existence of a version-control history, and means a definition can be saved and then applied within a single composite rewrite.

```
hermit> consider 'work ; remember origwork
```

```
work = unwrap (f (wrap work))
```

We now η-expand the body of *work* and unfold *unwrap*. Notice that the tuple constructor is polymorphic, and thus takes two types arguments (both *Nat*):

```
hermit> down ; eta-expand 'n
hermit> any-call (unfold 'unwrap)
```

```
λ n → (,) Nat Nat (f (wrap work) n) (f (wrap work) (S n))
```

We need to establish a base case for *work*. To do this we created a new command, `case-split-inline`, which takes as an argument a free variable in the expression. Using the type of the variable, a case expression is created with

one alternative per constructor, and the original expression is cloned as the right-hand side of each alternative. Finally, the constructor application making up the pattern of each alternative is inlined for the variable[2]:

```
hermit> down ; case-split-inline 'n

case n of n
  Z → (,) Nat Nat (f (wrap work) Z) (f (wrap work) (S Z))
  S a → (,) Nat Nat (f (wrap work) (S a)) (f (wrap work) (S (S a)))
```

This required modifying HERMIT's inline mechanism such that two definitions are stored for a case wildcard binder. Previously, calling `inline` on a wildcard binder would inline the case scrutinee. However, as the case analysis partially evaluates the scrutinee, we can instead inline the result of that evaluation (namely, the constructor or literal on the left-hand side of the alternative). This is now the default behavior, with only the default alternative inlining the scrutinee. The scrutinee can also be inlined explicitly using `inline-scrutinee`.

Now we selectively unfold f in three of the four places it is called, resulting in a fully simplified base case for *work*, and preparing to expose the duplicated computation in the $\mathsf{S}\ a$ case[3]:

```
hermit> { 1 ; any-call (unfold 'f) }
hermit> { 2 ; 0 ; 1 ; any-bu (unfold 'f) }
hermit> simplify

case n of n
  Z → (,) Nat Nat Z (S Z)
  S a → (,) Nat Nat (f (wrap work) (S a))
                    ((+) (wrap work (S a)) (wrap work a))
```

We move into the second case alternative for the remainder of the derivation. In the second tuple component, we unfold the saved definition of *work*:

```
hermit> 2 ; 0 ; { 1 ; any-bu (unfold origwork) }

(,) Nat Nat (f (wrap work) (S a))
            ((+) (wrap (unwrap (f (wrap work))) (S a))
                 (wrap (unwrap (f (wrap work))) a))
```

This creates an opportunity for fusing *wrap* and *unwrap* via the precondition, which we encode as a GHC RULES pragma:

$$\{\text{-\# RULES} \quad \texttt{"precondition"} \quad \forall\, w\,.\quad \mathit{wrap}\ (\mathit{unwrap}\ w) = w\ \text{\#-}\}$$

```
hermit> any-bu (unfold-rule precondition)

(,) Nat Nat (f (wrap work) (S a)) ((+) (f (wrap work) (S a))
                                       (f (wrap work) a))
```

[2] There is also a `case-split` command, which does not inline n in the alternatives.

[3] The numbers select a child node to descend into. The curly braces denote scoping: within a scope it is impossible to navigate above the node at which the scope starts, and when the scope ends the cursor returns to the starting node.

Now the duplicated computation of f (*wrap work*) (S a) is evident. We name each *distinct* call to f using let introduction, float the lets outside of the tuple, and use a newly created `fold` rewrite to combine the duplicated computation:

```
hermit> { 1 ; 1 ; let-intro 'x }
hermit> { 0 ; 1 ; let-intro 'y }
hermit> innermost (let-float-arg <+ let-float-app)
hermit> one-td (fold 'y)
let x = f (wrap work) a
    y = f (wrap work) (S a)
in (,) Nat Nat y ((+) y x)
```

Our implementation of `fold` performs a straightforward structural comparison of two expressions, attempting to instantiate one in terms of the other. It makes no attempt to convert into a normal form, so is currently limited to folding syntactically equivalent expressions. Of the new rewrites, this was the most challenging to add to HERMIT.

At this point, we would like to have the following code because it exposes an opportunity to fold *unwrap*:

$$\textbf{let } (x, y) = (f\ (wrap\ work)\ a, f\ (wrap\ work)\ (\mathsf{S}\ a))\ \textbf{in } (y, y + x)$$

However, GHC Core does not support pattern matching in let bindings. Thus we added the `let-tuple` rewrite, which combines two non-recursive let bindings into a single binding of a tuple. The original bindings are altered to project out of this tuple. This currently only works for pairs, but we plan to generalize it to n-ary tuples. The only complication in encoding this rewrite was locating GHC's tuple constructor, as the name (,) is used at both the type level and value level, and in GHC Core they live in the same name space.

```
hermit> let-tuple 'xy
let xy = (,) Nat Nat (f (wrap work) a) (f (wrap work) (S a))
    x = fst Nat Nat xy
    y = snd Nat Nat xy
in (,) Nat Nat y ((+) y x)
```

We can now fold *unwrap*:

```
hermit> one-td (fold 'unwrap)
let xy = unwrap (f (wrap work)) a
    x = fst Nat Nat xy
    y = snd Nat Nat xy
in (,) Nat Nat y ((+) y x)
```

All that remains is to fold our saved definition of *work*. This results in a definition with no calls to f, and no conversions via *wrap* and *unwrap*:

```
hermit> one-td (fold origwork)
let xy = work a
    x = fst Nat Nat xy
    y = snd Nat Nat xy
in (,) Nat Nat y ((+) y x)
```

Zooming out to see all of *fib*, we notice that f is now dead code. This would be removed by GHC's optimizer, but for presentation purposes we do so here. We also unfold the remaining call of *wrap*:

```
hermit> top ; consider 'fib
hermit> innermost dead-code-elimination
hermit> { 0 ; 1 ; any-call (unfold 'wrap) ; simplify }
fib = let rec work = λ n → case n of n
                              Z → (,) Nat Nat Z (S Z)
                              S a → let xy = work a
                                        x = fst Nat Nat xy
                                        y = snd Nat Nat xy
                                    in (,) Nat Nat y ((+) y x)
      in λ x → fst Nat Nat (work x)
```

We now have the efficient version of *fib*, and so tell GHC to resume compilation:

```
hermit> resume
```

6 Related Work

There are several refactoring tools for Haskell programs, including the Haskell Refactorer (HaRe) [3, 16], the Programming Assistant for Transforming Haskell (PATH) [28], the Ulm Transformation System (Ultra) [10], and the Haskell Equational Reasoning Assistant (HERA) [8]. HaRe and HERA allow the user to operate directly on Haskell source code, internally transforming the code into an AST (using Programatica [11] and Template Haskell [26], respectively) and then reconstructing the Haskell source code. PATH is similar, although it converts the program to its own Haskell-like language, and has the user operate on that. Ultra operates on Haskell extended with some non-deterministic operators, allowing non-executable specifications to be transformed into executable programs.

HERMIT differs from these other systems by operating on GHC Core, midway through the compilation process. The principal advantage of this approach is that GHC Core is a small language, having stripped away all of Haskell's syntactic sugar. This makes HERMIT simpler to use, implement and maintain, as there are far fewer cases to consider. Other advantages are that this automatically supports GHC language extensions, as GHC compiles them to GHC Core, and that inserting HERMIT inside the GHC optimization pipeline allows transformations to be intermixed with GHC's optimization passes. However, a disadvantage is that HERMIT cannot output Haskell source code.

One can also use proof assistants such as Coq [2] or Agda [17] to mechanize program transformations interactively. However, this requires modeling the syntax and semantics of the object language, encoding the program in that model, and then, after the transformation, transliterating the result back into the object language before it can be compiled and executed. Even were we to ignore GHC language extensions and consider only a limited subset of Haskell, the presence of partial values and lazy semantics means we cannot simply define our programs directly in the total languages provided by such proof assistants, but

instead have to model Haskell's domain theoretic setting of continuous functions over pointed ω-complete partial orders [24, 25], which in our experience is significantly harder to work with than GHC Core where that model is inherent. We emphasize that one of the aims of the HERMIT project is to make transforming Haskell programs as easy as possible for the user: we do not want familiarity with domain theory and proof assistants to be prerequisites.

7 Conclusions and Future Work

Our experience thus far has been that it is viable to mechanize basic program transformations, and that performing the transformations in HERMIT is no more complicated than on paper. However, while encoding our examples we repeatedly found it necessary to add additional transformations, and higher-level transformation strategies. This is unsurprising, as the HERMIT system is still in an early stage of development. What remains to be seen is whether, as we try more complex examples, we continue to need to add new transformations, or whether those we have now will scale. In general, we found adding new transformations to HERMIT to be a fairly simple procedure, whether by building them from HERMIT's existing low-level transformations, or by using GHC RULES. More challenging has been verifying the correctness of these transformations, and debugging our HERMIT programs when they fail to do as we expect.

Working within GHC has proved convenient. GHC Core has already been type checked before HERMIT acts on it, making all type information available. Much implementation effort was saved by using existing GHC functions such as substitution and variable de-shadowing, and safety checks such as the Core Lint pass [20] which ensures that the resultant code is type-correct and well-scoped.

More work is now needed. We have mechanized a collection of small examples as a proof of concept, but we need to try transforming larger real-world programs.

Acknowledgements

We thank Ed Komp for his work on implementing the HERMIT system. This work is supported by the National Science Foundation under Grant No. 1117569.

References

1. Appel, A.W.: Compiling with Continuations. Cambridge University Press (1992)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer (2004)
3. Brown, C.M.: Tool Support for Refactoring Haskell Programs. Ph.D. thesis, University of Kent (2008)
4. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (1977)
5. Farmer, A., Gill, A., Komp, E., Sculthorpe, N.: The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In: Haskell Symposium. ACM (2012)

6. Gammie, P.: Strict unwraps make worker/wrapper fusion totally correct. Journal of Functional Programming 21(2), 209–213 (2011)
7. GHC Team: The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.4.2 (2012), http://www.haskell.org/ghc
8. Gill, A.: Introducing the Haskell equational reasoning assistant. In: Haskell Workshop. pp. 108–109. ACM (2006)
9. Gill, A., Hutton, G.: The worker/wrapper transformation. Journal of Functional Programming 19(2), 227–251 (2009)
10. Guttmann, W., Partsch, H., Schulte, W., Vullinghs, T.: Tool support for the interactive derivation of formally correct functional programs. Journal of Universal Computer Science 9(2), 173–188 (2003)
11. Hallgren, T., Hook, J., Jones, M.P., Kieburtz, R.B.: An overview of the Programatica toolset. In: High Confidence Software and Systems (2004)
12. Hughes, R.J.M.: A novel representation of lists and its application to the function "reverse". Information Processing Letters 22(3), 141–144 (1986)
13. Hutton, G.: Programming in Haskell. Cambridge University Press (2007)
14. Jones, M.P.: Typing Haskell in Haskell. In: Haskell Workshop. pp. 1–14 (1999)
15. Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J.A., Rafkind, J., Tobin-Hochstadt, S., Findler, R.B.: Run your research: on the effectiveness of lightweight mechanization. In: Principles of Programming Languages. pp. 285–296. ACM (2012)
16. Li, H., Thompson, S.: Tool support for refactoring functional programs. In: Partial evaluation and semantics-based program manipulation. pp. 199–203. ACM (2008)
17. Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory. Ph.D. thesis, Chalmers University of Technology (2007)
18. Pettorossi, A.: A powerful strategy for deriving efficient programs by transformation. In: LISP and Functional Programming. pp. 273–281. ACM (1984)
19. Peyton Jones, S.: Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. Journal of Functional Programming 2(2), 127–202 (1992)
20. Peyton Jones, S., Santos, A.L.M.: A transformation-based optimiser for Haskell. Science of Computer Programming 32(1–3), 3–47 (1998)
21. Peyton Jones, S., Tolmach, A., Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in GHC. In: Haskell Workshop. pp. 203–233. ACM (2001)
22. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
23. Santos, A.: Compilation by Transformation in Non-Strict Functional Languages. Ph.D. thesis, University of Glasgow (1995)
24. Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. Allyn and Bacon (1986)
25. Sculthorpe, N., Hutton, G.: Work it, wrap it, fix it, fold it, in preparation.
26. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In: Haskell Workshop. pp. 1–16. ACM (2002)
27. Sulzmann, M., Chakravarty, M.M.T., Peyton Jones, S., Donnelly, K.: System F with type equality coercions. In: Types in Language Design and Implementaion. pp. 53–66. ACM (2007)
28. Tullsen, M.: PATH, A Program Transformation System for Haskell. Ph.D. thesis, Yale University (2002)
29. Wadler, P.: The concatenate vanishes. Tech. rep., University of Glasgow (1989)

Optimisation of Generic Programs through Inlining

José Pedro Magalhães*

Department of Computer Science, University of Oxford
jpm@cs.ox.ac.uk

Abstract. It is known that datatype-generic programs often run slower than type-specific variants, and this factor can prevent adoption of generic programming altogether. There can be multiple reasons for the performance penalty, but often it is caused by conversions to and from representation types that do not get eliminated during compilation. However, it is also known that generic functions can be specialised to specific datatypes, removing any overhead from the use of generic programming. In this paper, we investigate compilation techniques to specialise generic functions and remove the performance overhead of generic programs in Haskell. We pick a representative generic programming library and look at the generated code for a number of example generic functions. After understanding the necessary compiler optimisations for producing efficient generic code, we benchmark the runtime of our generic functions against handwritten variants, and conclude that all the overhead can indeed be removed automatically by the compiler.

1 Introduction

Datatype-generic programming is a form of abstraction that allows defining functions that operate on every datatype. Generic programs operate on the general structure of datatypes, therefore remaining agnostic of the individual detail of each datatype. Examples of behaviour that can be defined generically are (de)serialisation, equality testing, and traversing data. It is convenient to define such functions generically because less code has to be written, and this code has to be adapted less often. However, generic programs operate on the underlying structure of datatypes, and not on datatypes themselves directly. This indirection often causes a runtime penalty, as conversions to and from the generic representation are not always optimised away.

The performance of generic programs has been analysed before. [14] present a detailed comparison of nine libraries for generic programming in Haskell, with a brief performance analysis. This analysis indicates that the use of a generic approach could result in an increase of the running time by a factor of as much as 80. [10] also report severe performance degradation when comparing a generic approach to a similar but type-specific variant. While this is typically not a problem for smaller examples, it can severely impair adoption of generic programming in larger contexts. This problem is particularly relevant because generic programming techniques are especially applicable to large applications where performance is important, such as structure editors or compilers.

* This work has been funded by EPSRC grant number EP/J010995/1.

To understand the source of performance degradation when using a generic function from a particular generic programming library, we have to analyse the implementation of the library. The fundamental idea behind generic programming is to represent all datatypes by a small set of representation types. Equipped with conversion functions between user datatypes and their representation, we can define functions on the representation types, which are then applicable to all user types via the conversion functions. While these conversion functions are typically trivial and can be automatically generated, the overhead they impose is not automatically removed. In general, conversions to and from the generic representations are not eliminated by compilation, and are performed at run-time. These conversions are the source of inefficiency for generic programming libraries. In the earlier implementations of generic programming as code generators or preprocessors [4], optimisations (such as automatic generation of type-specialised variants of generic functions) could be implemented externally. With the switch to library approaches, all optimisations have to be performed by the compiler, as the library approach no longer generates code itself.

The Glasgow Haskell Compiler (GHC, the main Haskell compiler) compiles a program by first converting the input into a core language and then transforming the core code into more optimised versions, in a series of sequential passes. While it performs a wide range of optimisations, with the default settings it seems to be unable to remove the overhead incurred by using generic representations. Therefore generic libraries perform slower than handwritten type-specific counterparts. [1, 2] show that in many cases it is possible to remove all overhead by performing a specific form of symbolic evaluation in the Clean language. In fact, their approach is not restricted to optimising generics, and GHC performs symbolic evaluation as part of its optimisations. Our goal is to convince GHC to optimise generic functions so as to achieve the same performance as handwritten code, without requiring any additional manipulation of the compiler internals.

We have investigated this problem before [8], and concluded that tweaking GHC optimisation flags can achieve significant speedups. The problem with using compiler flags is that these apply to the entire program being compiled, and while certain flags might have a good effect on generic functions, they might adversely affect performance (or code size) of other parts of the program. In this paper we take a more fine-grained approach to the problem, looking at how to localise our performance annotations to the generic code only, by means of rewrite rules and function pragmas.[1] In this way we can improve the performance of generic functions with minimal impact on the rest of the program.

We continue by defining two representative generic functions which we focus our optimisation efforts on (Section 2). We then see how these functions can be optimised manually (Section 3), and transfer the necessary optimisation techniques to the compiler (Section 4). We confirm that our optimisations result in better runtime performance of generic programs in a benchmark in Section 5, and conclude in Section 6.

[1] `http://www.haskell.org/ghc/docs/latest/html/users_guide/pragmas.html`

2 Example generic functions

For analysing the performance of generic programs we choose the `generic-deriving` library, now integrated in GHC. Due to space considerations we cannot provide an introduction to this library; the reader is referred to related work for this purpose [8, 7]. We present two generic functions that will be the focus of our attention: equality and enumeration.

2.1 Generic equality

A notion of structural equality can easily be defined as a generic function. We first define a class for equality on the representation types:

```
class GEqRep φ where
  geqRep :: φ α → φ α → Bool
```

We can now give instances for each of the representation types:

```
instance GEqRep U1 where
  geqRep _ _ = True
instance (GEqRep α, GEqRep β) ⇒ GEqRep (α :+: β) where
  geqRep (L1 x) (L1 y) = geqRep x y
  geqRep (R1 x) (R1 y) = geqRep x y
  geqRep _      _      = False
instance (GEqRep α, GEqRep β) ⇒ GEqRep (α :×: β) where
  geqRep (x1:×:y1) (x2:×:y2) = geqRep x1 x2 ∧ geqRep y1 y2
instance (GEqRep α) ⇒ GEqRep (M1 ι γ α) where
  geqRep (M1 a) (M1 b) = geqRep a b
```

Units are trivially equal. For sums we continue the comparison recursively if both values are either on the left or on the right, and return *False* otherwise. Products are equal if both components are equal, and meta-information is ignored.

For recursive occurrences we fall back to an user-facing *GEq* class:

```
instance (GEq γ) ⇒ GEqRep (K1 ι γ) where
  geqRep (K1 a) (K1 b) = geq a b
```

This user-facing class is similar to *GEqRep*, but is used for user datatypes, and comes with a generic default method:

```
class GEq α where
  geq :: α → α → Bool
  default geq :: (Generic α, GEqRep (Rep α)) ⇒ α → α → Bool
  geq x y = geqRep (from x) (from y)
```

This class is similar to the Prelude *Eq* class, but we have left out inequality for simplicity. The generic default simply calls *from* on the arguments, and then proceeds using the generic equality function *geqRep*.

Adhoc instances for base types can reuse the Prelude implementation:

```
instance GEq Int where
  geq = (≡)
```

User datatypes, such as lists, can use the generic default:

```
instance (GEq α) ⇒ GEq [α]
```

2.2 Generic enumeration

We now define a function that enumerates all possible values of a datatype. For infinite datatypes we have to make sure that every possible value will eventually be produced. For instance, if we are enumerating integers, we should not first enumerate all positive numbers, and then the negatives. Instead, we should interleave positive and negative numbers.

While equality is a generic consumer, taking generic values as input, enumeration is a generic producer, since it generates generic values. We enumerate values by listing them with the standard list type. There is only one unit to enumerate, meta-information is ignored for enumeration purposes, and for datatype occurrences we refer to an user-facing *GEnum* class:

```
class GEnumRep φ where
  genumRep :: [φ α]

instance GEnumRep U1 where
  genumRep = [U1]
instance (GEnumRep φ) ⇒ GEnumRep (M1 ι γ φ) where
  genumRep = map M1 genumRep
instance (GEnum φ) ⇒ GEnumRep (K1 ι γ φ) where
  genumRep = map K1 genum
```

The more interesting cases are those for sums and products. For sums we enumerate both alternatives, but interleave them with a (|||) operator:

```
instance (GEnumRep α, GEnumRep β) ⇒ GEnumRep (α :+: β) where
  genumRep = map L1 genumRep ||| map R1 genumRep

infixr 5 |||
(|||) :: [α] → [α] → [α]
```

For products we generate all possible combinations of the two arguments, and diagonalise the result matrix, ensuring that all elements from each sublist will eventually be included, even if the lists are infinite:

```
instance (GEnumRep α, GEnumRep β) ⇒ GEnumRep (α :×: β) where
  genumRep = diag (map (λx → map (λy → x:×:y) genumRep) genumRep)

diag :: [[α]] → [α]
```

We omit the implementation details of $(|||)$ and *diag* as they are not important; it only matters that we have some form of fair interleaving and diagonalisation operations. The presence of $(|||)$ and *diag* throughout the generic function definition makes enumeration more complicated than equality, since equality does not make use of any auxiliary functions. We will see in Section 4.3 how this complicates the specialisation process. Note also that we do not use the more natural list comprehension syntax for defining the product instance, again to simplify the analysis of the optimisation process.

Finally, we define the user-facing class, with a default implementation:

```
class GEnum α where
  genum :: [α]
  default genum :: (Generic α, GEnumRep (Rep α)) ⇒ [α]
  genum = map to genumRep
```

3 Specialisation, by hand

We now focus on the problem of specialisation of generic functions. By specialisation we mean removing the use of generic conversion functions and representation types, replacing them by constructors of the original datatype. To convince ourselves that this task is possible we first develop a hand-written derivation of specialisation by equational reasoning. For simplicity we ignore implementation mechanisms such as the use of type classes and type families, and focus first on a very simple datatype encoding natural numbers:

```
data Nat = Ze | Su Nat
```

We give the representation of naturals with standard Haskell datatypes using a type synonym:

```
type RepNat = Either () Nat
```

We use a shallow representation (with *Nat* at the leaves, and not *RepNat*), remaining faithful with `generic-deriving`. We also need a way to convert between *RepNat* and *Nat*:

```
toNat :: RepNat → Nat
toNat     n = case n of { Left () → Ze; Right n → Su n; }
fromNat :: Nat → RepNat
fromNat n = case n of { Ze → Left (); Su n → Right n; }
```

We now analyse the specialisation of generic equality and enumeration on this datatype.

3.1 Generic equality

We start with a handwritten, type-specific definition of equality for *Nat*:

```
eqNat :: Nat → Nat → Bool
eqNat m n = case (m , n) of
```

```
(Ze    , Ze   ) → True
(Su m  , Su n) → eqNat m n
(_     , _    ) → False
```

For equality on *RepNat* we need equality on units and sums:

```
eqU :: () → () → Bool
eqU x y = case (x , y) of {(() , ()) → True; }

eqPlus :: (α → α → Bool) → (β → β → Bool) →
          Either α β → Either α β → Bool
eqPlus ea eb a b = case (a , b) of
                     (Left  x , Left  y) → ea x y
                     (Right x , Right y) → eb x y
                     (_       , _      ) → False
```

Now we can define equality for *RepNat*, and generic equality for *Nat* through conversion to *RepNat*:

```
eqRepNat :: RepNat → RepNat → Bool
eqRepNat = eqPlus eqU eqNatFromRep

eqNatFromRep :: Nat → Nat → Bool
eqNatFromRep m n = eqRepNat (fromNat m) (fromNat n)
```

Our goal now is to show that *eqNatFromRep* is equivalent to *eqNat*. In the following derivation, we start with the definition of *eqNatFromRep*, and end with the definition of *eqNat*:

```
  eqRepNat (fromNat m) (fromNat n)

≡⟨ inline eqRepNat ⟩
  eqPlus eqU eqNatFromRep (fromNat m) (fromNat n)

≡⟨ inline eqPlus ⟩
  case (fromNat m , fromNat n) of
    (Left  x , Left  y) → eqU x y
    (Right x , Right y) → eqNatFromRep x y
    _                   → False

≡⟨ inline fromNat ⟩
  case ( case m of {Ze → Left (); Su x1 → Right x1 }
       , case n of {Ze → Left (); Su x2 → Right x2 }) of
    (Left  x , Left  y) → eqU x y
    (Right x , Right y) → eqNatFromRep x y
    _                   → False

≡⟨ case-of-case transform ⟩
```

```
  case (m , n) of
    (Ze    , Ze    ) → equ () ()
    (Su x1 , Su x2) → eqNatFromRep x1 x2
    _               → False
```

≡⟨ inline *equ* and case-of-constant ⟩

```
  case (m , n) of
    (Ze    , Ze    ) → True
    (Su x1 , Su x2) → eqNatFromRep x1 x2
    _               → False
```

≡⟨ inline *eqNatFromRep*, induction ⟩

```
  case (m , n) of
    (Ze    , Ze    ) → True
    (Su x1 , Su x2) → eqNat x1 x2
    _               → False
```

This shows that the generic implementation is equivalent to the type-specific variant, and that it can be optimised to remove all conversions. We discuss the techniques used in this derivation in more detail in Section 4.1, after showing the optimisation of generic enumeration.

3.2 Generic enumeration

A type-specific enumeration function for *Nat* follows:

```
enumNat :: [Nat]
enumNat = [Ze] ||| map Su enumNat
```

To get an enumeration for *RepNat* we first need to know how to enumerate units and sums:

```
enumU :: [()]
enumU = [()]

enumPlus :: [α] → [β] → [Either α β]
enumPlus ea eb = map Left ea ||| map Right eb
```

Now we can define an enumeration for *RepNat*:

```
enumRepNat :: [RepNat]
enumRepNat = enumPlus enumU enumNatFromRep
```

With the conversion function toNat, we can use *enumRepNat* to get a generic enumeration function for *Nat*:

```
enumNatFromRep :: [Nat]
enumNatFromRep = map toNat enumRepNat
```

We now show that *enumNatFromRep* and *enumNat* are equivalent:

$$
\begin{aligned}
& \mathit{map}\ \mathit{toNat}\ \mathit{enumRepNat} \\
\equiv\langle\ & \text{inline } \mathit{enumRepNat}\ \rangle \\
& \mathit{map}\ \mathit{toNat}\ (\mathit{enumPlus}\ \mathit{enumU}\ \mathit{enumNatFromRep}) \\
\equiv\langle\ & \text{inline } \mathit{enumPlus}\ \rangle \\
& \mathit{map}\ \mathit{toNat}\ (\mathit{map}\ \mathit{Left}\ \mathit{enumU} \mid\mid\mid \mathit{map}\ \mathit{Right}\ \mathit{enumNatFromRep}) \\
\equiv\langle\ & \text{inline } \mathit{enumU}\ \rangle \\
& \mathit{map}\ \mathit{toNat}\ (\mathit{map}\ \mathit{Left}\ [()] \mid\mid\mid \mathit{map}\ \mathit{Right}\ \mathit{enumNatFromRep}) \\
\equiv\langle\ & \text{inline } \mathit{map}\ \rangle \\
& \mathit{map}\ \mathit{toNat}\ ([\mathit{Left}\ ()] \mid\mid\mid \mathit{map}\ \mathit{Right}\ \mathit{enumNatFromRep}) \\
\equiv\langle\ & \text{free theorem } (\mid\mid\mid) : \forall f\ a\ b.\mathit{map}\ f\ (a \mid\mid\mid b) = \mathit{map}\ f\ a \mid\mid\mid \mathit{map}\ f\ b\ \rangle \\
& \mathit{map}\ \mathit{toNat}\ [\mathit{Left}\ ()] \mid\mid\mid \mathit{map}\ \mathit{toNat}\ (\mathit{map}\ \mathit{Right}\ \mathit{enumNatFromRep}) \\
\equiv\langle\ & \text{inline } \mathit{map}\ \rangle \\
& [\mathit{toNat}\ (\mathit{Left}\ ())] \mid\mid\mid \mathit{map}\ \mathit{toNat}\ (\mathit{map}\ \mathit{Right}\ \mathit{enumNatFromRep}) \\
\equiv\langle\ & \text{inline } \mathit{toNat} \text{ and case-of-constant}\ \rangle \\
& [\mathit{Ze}] \mid\mid\mid \mathit{map}\ \mathit{toNat}\ (\mathit{map}\ \mathit{Right}\ \mathit{enumNatFromRep}) \\
\equiv\langle\ & \text{functor composition law: } \forall f\ g\ l.\mathit{map}\ f\ (\mathit{map}\ g\ l) = \mathit{map}\ (f \circ g)\ l\ \rangle \\
& [\mathit{Ze}] \mid\mid\mid \mathit{map}\ (\mathit{toNat} \circ \mathit{Right})\ \mathit{enumNatFromRep} \\
\equiv\langle\ & \text{inline } \mathit{toNat} \text{ and case-of-constant}\ \rangle \\
& [\mathit{Ze}] \mid\mid\mid \mathit{map}\ \mathit{Su}\ \mathit{enumNatFromRep}
\end{aligned}
$$

Like equality, generic enumeration can also be specialised to a type-specific variant without any overhead.

4 Specialisation, by the compiler

After the manual specialisation of generic functions, let us now analyse how to convince the compiler to automatically perform the specialisation.

4.1 Optimisation techniques

Our calculations in Section 3 rely on a number of lemmas and techniques that the compiler will have to use. We review them here:

Inlining Inlining replaces a function call with its definition. It is a crucial optimisation technique because it can expose other optimisations. However, inlining causes code duplication, and care has to be taken to avoid non-termination through infinite inlining.

GHC uses a number of heuristics to decide when to inline a function or not, and loop breakers for preventing infinite inlining [11]. The programmer can provide explicit inlining annotations with the *INLINE* and *NOINLINE* pragmas, of the form:

$$\{-\#\ \mathit{INLINE}\ [n]\ f\ \#-\}$$

In this pragma, f is the function to be inlined, and n is a phase number. GHC performs a number of optimisation phases through a program, numbered in decreasing order until zero. Setting n to 1, for instance, means "be keen to inline f in phase 1 and after". For a *NOINLINE* pragma, this means "do not inline f in phase 1 or after". The phase can be left out, in which case the pragma applies to all phases.[2]

Application of free theorems and functor laws Free theorems [15] are theorems that arise from the type of a polymorphic function, regardless of the function's definition. Each polymorphic function is associated with a free theorem, and functions with the same type share the same theorem. The functor laws arise from the categorical nature of functors. Every *Functor* instance in Haskell should obey the functor laws.

GHC does not compute and use the free theorem of each polymorphic function, also because it may not be clear which direction of the theorem is useful for optimisation purposes. However, we can add special optimisation rules to GHC via a *RULES* pragma [13]. For instance, the rewrite rule corresponding to the free theorem of $(|||)$ follows:

$$\{-\#\ \mathit{RULES}\ \texttt{"ft |||"}\ \forall f\ a\ b.\ \mathit{map}\ f\ (a\ |||\ b) = \mathit{map}\ f\ a\ |||\ \mathit{map}\ f\ b\ \#-\}$$

This pragma introduces a rule named "ft |||" telling GHC to replace appearances of the application $\mathit{map}\ f\ (a\ |||\ b)$ with $\mathit{map}\ f\ a\ |||\ \mathit{map}\ f\ b$. GHC does not perform any confluence checking on rewrite rules, so the programmer should ensure confluence or GHC might loop during compilation.

Optimisation of case statements Case statements drive evaluation in GHC's core language, and give rise to many possible optimisations. [12] provide a detailed account of these; in our derivation in Section 3.2 we used a "case of constant" rule to optimise a statement of the form:

$$\textbf{case}\ (\mathit{Left}\ ())\ \textbf{of}\ \{\mathit{Left}\ () \to \mathit{Ze};\ \mathit{Right}\ n \to \mathit{Su}\ n;\ \}$$

Since we know what we are case-analysing, we can replace this case statement by the much simpler expression *Ze*. Similarly, in Section 3.1 we used a case-of-case transform to eliminate an inner case statement. Consider an expression of the form:

$$\textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{e1 \to e2;\ \})\ \textbf{of}\ \{e2 \to e3;\ \}$$

Here, *e1*, *e2*, and *e3* are expressions starting with a constructor. We can simplify this to:

$$\textbf{case}\ x\ \textbf{of}\ \{e1 \to e3;\ \}$$

[2] See the GHC User's Guide for more details: `http://www.haskell.org/ghc/docs/latest/html/users_guide/pragmas.html`.

This rule naturally generalises to case statements with multiple branches.

4.2 Generic equality

We have seen that we have a good number of tools at our disposal for directing the optimisation process in GHC: inline pragmas, rewrite rules, phase distinction, and all the standard optimisations for the functional core language. We will now annotate our generic functions and evaluate the quality of the core code generated by GHC.

We start by defining a *Generic* instance for the *Nat* type:

```
instance Generic Nat where
  type Rep Nat = U1 :+: Rec0 Nat
  {-# INLINE [1] to #-}
  to (L1 U1)      = Ze
  to (R1 (K1 n)) = Su n
  {-# INLINE [1] from #-}
  from Ze      = L1 U1
  from (Su n) = R1 (K1 n)
```

We give inline pragmas *to* and *from* to guarantee that these functions will be inlined. However, we ask the inliner to only inline them on phase 1 and after; this is to guarantee that we first inline the generic function definitions, and to prevent partial fusion.

We can now provide a generic definition of equality for *Nat*:

```
instance GEq Nat
```

Compiling this code with the standard optimisation flag `-O` gives us the following core code:

```
$GEqNat_geq :: Nat → Nat → Bool
$GEqNat_geq = λ(x :: Nat) (y :: Nat) →
                case x of
                  Ze   → case y of {Ze → True;  Su m → False; }
                  Su m → case y of {Ze → False; Su n → $GEqNat_geq m n; }
```

The core language is a small, explicitly typed language in the style of System F [16]. The function $\$GEqNat_{geq}$ is prefixed with a \$ because it was generated by the compiler, representing the *geq* method of the *GEq* instance for *Nat*. We can see that the generic representation was completely removed.

The same happens for lists, as evidenced by the generated core code:

```
$GEq[]_geq :: ∀α.GEq α ⇒ [α] → [α] → Bool
$GEq[]_geq = λα (eqA :: GEq α) (l1 :: [α]) (l2 :: [α]) →
               case l1 of
                 []       → case l2 of {[] → True; (h : t) → False; }
                 (h1 : t1) → case l2 of
                               []       → False
                               (h2 : t2) → case eqA h1 h2 of
```

$$\begin{aligned} &\mathit{False} \to \mathit{False} \\ &\mathit{True} \ \to \$\mathit{GEq}[]_{\mathit{geq}}\ \alpha\ \mathit{eqA}\ t_1\ t_2 \end{aligned}$$

Note that type abstraction and application is explicit in core. There is syntax to distinguish type and value application and abstraction from each other, but we suppress the distinction since it is clear from the colour. Note also that constraints (to the left of the $\Rightarrow$ arrow) become just ordinary parameters, so $\$\mathit{GEq}[]_{\mathit{geq}}$ takes a function to compute equality on the list elements, *eqA*.[3]

Perhaps surprisingly, GHC performs all the required steps of Section 3.1 without requiring any annotations to the generic function itself. In general, however, we found that it is sensible to provide *INLINE* pragmas for each instance of the representation datatypes when defining a generic function. In the case of *geqRep*, the methods are small, so GHC inlines them eagerly. For more complicated generic functions, the methods may become larger, and GHC will avoid inlining them. Supplying an *INLINE* pragma tells GHC to inline the methods anyway.

4.3 Generic enumeration

Generic consumers, such as equality, are, in our experience, more easily optimised by GHC. A generic producer such as enumeration, in particular, is challenging because it requires map fusion, and lifting auxiliary functions through maps using free theorems. As such, we encounter some difficulties while optimising enumeration. We start by looking at the natural numbers:

```
instance GEnum Nat where
  genum = map to genumRep
```

Note that instead of using the default definition we directly inline its definition; this is to circumvent a limitation in the current implementation of defaults that prevents later rewrite rules from applying. GHC then generates the following code:

$$\begin{aligned} &\$x_2 :: [\,U_1 :+: \mathit{Rec}_0\ \mathit{Nat}\,] \\ &\$x_2 = \mathit{map}\ \$x_4\ \$\mathit{GEnumNat}_{\mathit{genum}} \\ &\$x_1 :: [\,U_1 :+: \mathit{Rec}_0\ \mathit{Nat}\,] \\ &\$x_1 = \$x_3 \,|||\, \$x_2 \\ &\$\mathit{GEnumNat}_{\mathit{genum}} :: [\mathit{Nat}\,] \\ &\$\mathit{GEnumNat}_{\mathit{genum}} = \mathit{map}\ \mathit{to}\ \$x_1 \end{aligned}$$

We omit the definitions of $\$x_3$ and $\$x_4$ for brevity. To make progress we need to tell GHC to move the *map to* expression in $\$\mathit{GEnumNat}_{\mathit{genum}}$ through the $(|||)$ operator. We use a rewrite rule for this:

$$\{-\#\ \mathit{RULES}\ \texttt{"ft |||"}\ \forall f\ a\ b.\ \mathit{map}\ f\ (a \,|||\, b) = \mathit{map}\ f\ a \,|||\, \mathit{map}\ f\ b\ \#-\}$$

With this rule in place, GHC generates the following code:

[3] The type of *eqA* is $\mathit{GEq}\ \alpha$, but we use it as if it had type $\alpha \to \alpha \to \mathit{Bool}$. In the generated core there is also a coercion around the use of *eqA* to transform the class type into a function. This is the standard way class methods are desugared into core; we elide these details as they are not relevant to optimisation itself.

$$\$x_2 :: [U_1 :+: Rec_0\ Nat]$$
$$\$x_2 = map\ \$x_4\ \$GEnumNat_{genum}$$
$$\$x_1 :: [Nat]$$
$$\$x_1 = map\ to\ \$x_2$$
$$\$GEnumNat_{genum} :: [Nat]$$
$$\$GEnumNat_{genum} = \$x_3 \mid\mid\mid \$x_1$$

We now see that the $\$x_1$ term is *map* applied to the result of a *map*. The way *map* is optimised in GHC (by conversion to *build*/*foldr* form) interferes with our `"ft |||"` rewrite rule, and map fusion is not happening. We can remedy this with an explicit map fusion rewrite rule:

$$\{-\#\ RULES\ \texttt{"map/map1"}\ \forall f\ g\ l.\ map\ f\ (map\ g\ l) = map\ (f \circ g)\ l\ \#-\}$$

This rule results in much improved generated code:

$$\$x_3 :: [U_1 :+: Rec_0\ Nat]$$
$$\$x_3 = \$x_4 : []$$
$$\$x_2 :: [Nat]$$
$$\$x_2 = map\ to\ \$x_3$$
$$\$x_1 :: [Nat]$$
$$\$x_1 = map\ Su\ \$GEnumNat_{genum}$$
$$\$GEnumNat_{genum} :: [Nat]$$
$$\$GEnumNat_{genum} = \$x_2 \mid\mid\mid \$x_1$$

The only thing we are missing now is to optimise $\$x_3$; note that its type is $[U_1 :+: Rec_0\ Nat]$, and not $[Nat]$. For this we simply need to tell GHC to eagerly map a function over a list with a single element:

$$\{-\#\ RULES\ \texttt{"map/map2"}\ \forall f\ x.\ map\ f\ (x : []) = (f\ x) : []\ \#-\}$$

With this, GHC can finally generate the fully specialised enumeration function on *Nat*:

$$\$x_2 :: [Nat]$$
$$\$x_2 = Ze : []$$
$$\$x_1 :: [Nat]$$
$$\$x_1 = map\ Su\ \$GEnumNat_{genum}$$
$$\$GEnumNat_{genum} :: [Nat]$$
$$\$GEnumNat_{genum} = \$x_2 \mid\mid\mid \$x_1$$

Compelling GHC to optmise generic enumeration for lists proves to be more difficult.[4] Since lists use products, we need to introduce a rewrite rule for the free theorem of *diag*, allowing *map* to be pushed inside *diag*:

$$\{-\#\ RULES\ \texttt{"ft/diag"}\ \forall f\ l.\ map\ f\ (diag\ l) = diag\ (map\ (map\ f)\ l)\ \#-\}$$

[4] We believe, however, that this is only due to bugs in the inliner, and have filed bug reports #7109, #7112, and #7114 to address these issues.

With this rule, and the extra optimisation flag `-fno-full-laziness` to maximise the chances for rewrite rules to apply, we get the following code:

$$
\begin{aligned}
&\$GEnum[]_{genum} :: \forall\alpha.GEnum\ \alpha \Rightarrow [[\alpha]] \\
&\$GEnum[]_{genum} = \lambda(gEnumA :: GEnum\ \alpha) \rightarrow \\
&\quad ([] : []) \,|||\, \textbf{let}\ \$x_1 :: [Rec_0\ [\alpha]] \\
&\qquad\qquad\qquad \$x_1 = map\ K_1\ (\$GEnum[]_{genum}\ gEnumA) \\
&\qquad\qquad \textbf{in}\ diag\ (map\ (\lambda(\$x_3 :: \alpha) \rightarrow \\
&\qquad\qquad\qquad\qquad map\ (\lambda(\$x_2 :: Rec_0\ [\alpha]) \rightarrow \textbf{case}\ \$x_2\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad K_1\ \$x_4 \rightarrow \$x_3 : \$x_4)\ \$x_1) \\
&\qquad\qquad\qquad\qquad gEnumA)
\end{aligned}
$$

Most of the generic overhead is optimised away, but one problem remains: $\$x_1$ maps K_1 over the recursive enumeration elements, but this K_1 is immediately eliminated by a **case** statement. If $\$x_1$ was inlined, GHC could perform a map fusion, and then eliminate the use of K_1 altogether. However, we have no way to specify that $\$x_1$ should be inlined; the compiler generated it, so only the compiler can decide when to inline it. Also, we had to use the compiler flag `-fno-full-laziness` to prevent some let-floating, but the flag applies to the entire program and might have unintended side-effects.

Reflecting on our developments in this section, we have seen that:

- Convincing GHC to optimise *genum* for a simple datatype such as *Nat* requires the expected free theorem of $(|||)$. However, due to interaction between phases of application of rewrite rules, we are forced to introduce new rules for optimisation of *map*.
- Optimising *genum* for a more complicated datatype like lists requires the expected free theorem of *diag*. However, even after further tweaking of optimisation flags, we are currently unable to derive a fully optimised implementation. In any case, the partial optimisation achieved is certainly beneficial.
- More generally, we see that practical optimisation of generic functions is hard because of subtle interactions between the different optimisation mechanisms involved, such as inlining, rewrite rule application, **let** floating, **case** optimisation, etc.

These experiments have been performed with GHC version 7.4.1. We have observed that the behavior of the optimiser changes between compiler versions. In particular, some techniques which resulted in better code in some versions (e.g. the use of *SPECIALISE* pragmas) result in worse code in other versions. We are working together with GHC developers to ensure that generic code, at least for the `generic-deriving` library, is specialised adequately, guaranteeing performance equivalent to type-specific code.

5 Benchmarking

We have confirmed the runtime behaviour of our code by benchmarking it. Benchmarking is, in general, a complex task, and a lazy language imposes even more challenges

on the design of a benchmark. We designed a benchmark suite that ensures easy repeatability of tests, calculating the average running time and the standard deviation for statistical analysis. It is portable across different operating systems and can easily be run with different compiler versions. To ensure reliability of the benchmark we use profiling, which gives us information about which computations last longer. For each of the tests, we ensure that at least 50% of the time is spent on the function we want to benchmark. A top-level Haskell script takes care of compiling all the tests with the same flags, invoking them a given number of times, parsing and accumulating results as each test finishes, and calculating and displaying the average running time at the end, along with some system information.

We have a detailed benchmark suite over different datatypes, generic functions, and generic programming libraries.[5] Due to space constraints, we present here only a general overview of the results. Confirming the findings of Section 4, the benchmark finds no difference between the running times of generic versus type-specific equality. We have also benchmarked a traversal that updates the values in a tree, and a conversion to *String*; in both cases, the generic function performs as fast as the handwritten code. As for enumeration, we find no overheard for the *Nat* datatype. Enumeration for a binary tree datatype runs about 1.63 times slower than a type-specific variant, probably because the optimiser fails to remove all generic representation overhead.

6 Conclusion

In this paper we have looked at the problem of optimising generic functions. With their representation types and associated conversions, generic programs tend to be slower than their type-specific handwritten counterparts, and this can limit adoption of generic programming in situations where performance is important. We have picked one specific library, `generic-deriving`, and investigated the code generation for generic programs, and the necessary optimisation techniques to fully remove any overhead from the library. We concluded that the overhead can be fully removed most of the time, using only already available optimisations that apply to functional programs in general. However, due to the difficulty of managing the interaction between several different optimisations, in some cases we are not able to fully remove the overhead. We are confident, however, that this is only a matter of further tweaking of GHC's optimisation strategies.

6.1 Automatic inlining and generation of rewrite rules

Some work remains to be done in terms of improving the user experience. We have mentioned that the *to* and *from* functions should be inlined; this should be automatically established by the mechanism for deriving *Generic* instances. Additionally, inserting *INLINE* pragmas for each case in the generic function is a tedious process, which should also be automated. Finally, it would be interesting to see if the definition of rewrite rules based on free theorems of auxiliary functions used could be automated; it is easy

[5] `https://bitbucket.org/dreixel/public/src/7d32c569e678/benchmark`

to generate free theorems, but it is not always clear how to use these theorems for optimisation purposes.

6.2 Optimising other libraries

The library we have used for the development in this paper, `generic-deriving`, is practical, realistic, and representative of many other libraries. In particular, our techniques readily apply to `regular` [9] and `instant-generics` [3], for instance.

Other approaches to generic programming, such as Scrap Your Boilerplate [SYB, 5, 6], use different implementation mechanisms and require different optimisation strategies. SYB, in particular, cannot be optimised using the same techniques we have seen, because it relies on (type-safe) runtime casts. Since type comparisons are performed at runtime, the compiler does not have enough information to automatically specialise generic functions. It remains to be seen how to optimise other approaches, and to establish general guidelines for optimisation of generic programs.

In any case, it is now clear that generic programs do not have to be slow, and their optimisation up to handwritten code performance is not only possible but also achievable using only standard optimisation techniques. This opens the door for a future where generic programs are not only general, elegant, and concise, but also as efficient as type-specific code.

Bibliography

[1] Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference*, volume 3125 of *LNCS*, pages 16–31. Springer, 2004. doi: 10.1007/978-3-540-27764-4_3.

[2] Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium*, volume 3350 of *LNCS*, pages 203–218. Springer, 2005. doi: 10.1007/978-3-540-30557-6_16.

[3] Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy, 2009. Available at `http://www.cse.unsw.edu.au/~chak/papers/CDL09.html`.

[4] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approches to generic programming in Haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 72–149. Springer-Verlag, 2007. doi: 10.1007/978-3-540-76786-2_2.

[5] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 26–37. ACM, 2003. doi: 10.1145/604174.604179.

[6] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming*, pages 244–255. ACM, 2004. doi: 10.1145/1016850.1016883.

[7] José Pedro Magalhães. *Less Is More: Generic Programming Theory and Practice*. PhD thesis, Universiteit Utrecht, 2012.

[8] José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. Optimizing generics is easy! In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 33–42. ACM, 2010. doi: 10.1145/1706356.1706366.

[9] Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 13–24. ACM, 2008. doi: 10.1145/1411318.1411321.

[10] Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren, and José Pedro Magalhães. A lightweight approach to datatype-generic rewriting. *Journal of Functional Programming*, 20(Special Issue 3–4):375–413, 2010. doi: 10.1017/S0956796810000183.

[11] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4&5):393–433, 2002. doi: 10.1017/S0956796802004331.

[12] Simon Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32:3–47, September 1998. doi: 10.1016/S0167-6423(97)00029-4.

[13] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop 2001*, pages 203–233, 2001.

[14] Alexey Rodriguez Yakushev, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C.d.S. Oliveira. Comparing libraries for generic programming in Haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, pages 111–122. ACM, 2008. doi: 10.1145/1411286.1411301.

[15] Philip Wadler. Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404.

[16] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012. doi: 10.1145/2103786.2103795.

The Nax programming language (work in progress)

Ki Yung Ahn[1], Tim Sheard[1], Marcelo Fiore[2], and Andrew M. Pitts[2]

[1] Portland State University, Portland, Oregon, USA *
kya@cs.pdx.edu sheard@cs.pdx.edu
[2] University of Cambridge, Cambridge, UK
{Marcelo.Fiore,Andrew.Pitts}@cl.cam.ac.uk

1 Introduction

During the past decade, the functional programming community achieved partial success in their goal of maintaining fine-grained properties by moderate extensions to the type system of functional languages[6, 5, 11]. This approach is often called *"lightweight"* (e.g., lightweight dependent types[3], lightweight program verification), since using a full blown proof assistant to maintain similar properties is likely to require much more effort (heavyweight).

The Generalized Algebraic Data Type (GADT) extension, implemented in the Glasgow Haskell Compiler (GHC), has made this approach possible even when performing everyday functional programming tasks. Unfortunately, implementations supporting a lightweight approach (e.g., GHC) lack **correctness guarantees** and **type inference** in general. In addition, practical implementations often lack support for GADTs indexed by terms, so **term indices are faked** (or simulated) by additional type structure replicating the requisite term structure. We believe that the lightweight approach can become more productive and reliable if we can resolve these problems.

Problem 1. ***Correctness guarantee***

Proof assistants based on dependent types can *express fine-grained properties* and also *guarantee correctness* since these calculi are based on strongly normalizing and logically consistent systems. For instance, Coq is based on the Calculus of Inductive Constructions, which is a dependently-typed λ-calculi known to be strongly normalizing and logically consistent.

But, the same fine-grained properties are not expressible in ordinary functional programming languages with only simple polymorphic types, since such languages lack the expressivity of dependent types. Even if these fine-grained properties were somehow expressible, one would not have any guarantee of correctness. Recall that general purpose programming languages are neither strongly normalizing nor logically consistent, because they are (by design) capable of expressing diverging computations. So, the lightweight approach in conventional functional languages can only raise programmers confidence of correctness

* supported by NSF grant 0910500.

[3] http://okmij.org/ftp/Computation/lightweight-dependent-typing.html

(assuming that the inconsistent fragment of the type system was never used for reasoning about the desired properties) but cannot guarantee the correctness of the desired properties as is the norm in proof assistants.

Problem 2. ***Type inference***

Type inference makes type-safe programming pleasant when performomng everyday programming tasks, since programmers are freed from including tedious type annotations. Many typed functional programming languages including Haskell 98 and SML are based on the Hindley-Milner type system (HM), which is not only *type-safe* but also supports *type inference*.

An essential feature of the lightweight approach is *indexed datatypes*, which are datatypes with *heterogeneous type parameters* (a.k.a. *indices*) which are made possible by the GADT extension. Such datatypes, often used in lightweight approaches, are beyond the capabilities of polymorphic type schemes used in HM. Inclusion of just a simple subset of the indexed datatypes, such as nested datatypes [3], already make type inference undecidable [8]. More sophisticated uses add even more complication[9].

So, functional language implementations, that support the lightweight approach, require type annotations on both indexed datatype declarations and on function definitions that pattern match indexed datatypes, in order to recover a semblence of type inference.[4] Type annotations on datatype declarations are absolutely necessary when either the result types of their data constructors have indices or when the argument types of their data constructors have existential indices. However, it still an open question of *where and how much type annotations are needed on function definitions*.

Problem 3. ***Faked term indices***

The indexed datatypes can only have static dependencies (i.e., indices must be compleytely known at type checking time), unlike full-fledged dependent types, as used in proof assistants, which can depend on both static and dynamic values. Therefore, having term indices does not imply full-fledged dependent types.

Indexed datatypes can be indexed by either types or terms, or both. Type representations [6] (Fig. 1) used in datatype generic programming are typical examples of type-indexed datatypes. The length-indexed list, or `Vector`, (Fig. 2) is an example of a term-indexed datatype. However, the `Vector` datatype declaration in Fig. 2 uses faked term indices. These indexes are faked because rather than use the real term constructors of the natural numbers (defined by `data Nat = Succ Nat | Zero`) it uses the uninhabited types `Succ` and `Zero` to simulate the data constructors of `Nat`. Such faked term indices are problematic since they (1) duplicate code (i.e., operations on `Nat` must be redefined at type level) and (2) have less precise semantics than true term indices (e.g., cannot prevent ill-typed types such as `Succ Bool`).

[4] Type inference aided by type annotation is also called partial type inference or type reconstruction.

Although indexed datatypes with true term indices have been studied[12], including term indices in practical functional languages is not trivial. Allowing arbitrary terms at the type level breaks the decidability of type checking due to diverging terms. Term indices in types implies that type equality depends on term equality. And, obviously, term equality will loop when one of the terms being compared diverges. Undecidability of type checking can be lifted once we have resolved *Problem 1*, and make sure that indices are normalizing.

```
data Rep t where
  R_Int  :: Rep Int
  R_Char :: Rep Char
  R_List :: Rep a -> Rep [a]
  R_Pair :: Rep a -> Rep b -> Rep (a,b)
```

Fig. 1. A type representation for `Int`, `Char`, `[]`, and `(,)` in Haskell with GADTs

```
data Succ n
data Zero

data Vector a n where
  VCons :: a -> Vector m a -> Vector a (Succ m)
  VNil  :: Zero
```

Fig. 2. Length indexed list datatype `Vector` in Haskell with GADTs

To resolve these problems, we have designed and implemented a prototype of the Nax programming language. The current proptotype of Nax is a strongly normalizing functional language supporting the following features (which are all illustrated in §2):

Two level datatypes. Recursive dataypes are introduced in two stages. First a non-recursive *structure* is introduced which abstracts over where recursive sub-components will appear. Then a *fix-point* is taken to define the recursive types (§2.1). To minimize the extra notation necessary to program in this manner an extensive *macro-facility* is provided. The most common macro forms can be automatically derived. This is illustrated in §2.3.

Indexed types with static term indices. A type constructor is applied to arguments. Arguments are either parameters or indices. A datatype is polymorphic over its parameters (in the sense that parametricity theorems hold

over parameters). Parameters are always types. Indexed arguments can be either types or terms. An index usually encodes a static property about the shape or form of a value with that type. We use different kinding rules to separate term indices from type indices. For instance a length indexed list, `x`, might have type (`List Int 2`). The `Int` is a parameter, indicating the list contains integers, but the `2` is an index indicating that the list, `x`, has exactly two elements. Types are static in Nax. Types are only used for type checking and are computationally irrelevant, even though some parts of a type might include terms. In other words, Nax supports *index erasure*,

Recursive types of unrestricted polarity but restricted elimination. It is well known that unrestricted recursive types enable diverging computation even without any recursion at the term level. To design a normalizing language that supports recursive types, we must make a design choice that limits the use of recursive types. There are two possible design choices. We may restrict the formation of recursive types (i.e., type definition) or we may restrict the elimination of recursive types (i.e., pattern matching). In Nax, we make the latter design choice, so that we can define *all the recursive datatypes* available in modern functional languages.

Mendler style iteration and recursion combinators. Any useful normalizing language should support principled recursion operators that guarantee normalization. Such operators should be easy to use, and expressive over datatypes with both parameters and indices. Mendler style combinators meet both requirements. So, we adopt them in Nax.

Type inference (reconstruction) from minimal annotation. When we extend the Hindley-Milner type system with indexed data types, we no longer have type inference for completely unannotated terms. For example, this restriction shows up in languages which support GADTs, which support a kind of type indexing. Although complete type inference is not possible, partial type inference (reconstruction of missing type information) is still possible when sufficient type annotations are provided. Nax's systematic partition of type parameters from type indices provides a mechanism where it is possible to decide exactly where additional type annotations are needed, and to enforce that the programmer supply such annotations. This system faithfully extends the Hindley-Milner type inference (i.e., no additional annotations are needed for the programs that are already inferable by Hindley-Milner).

2 Nax by Example

We introduce programming in our implementation of Nax by providing examples. An example usually consists of several parts.

- Introducing data definitions to describe the data of interest. Recursive data is introduced in two stages. We must be careful to separate parameters from indices when using indices to describe static properties of data.

– Introduce macros, either by explicit definition, or by automatic fixpoint derivation to limit the amount of explicit notation that must be supplied by the programmer.
– Write a series of definitions that describe how the data is to be manipulated. Deconstruction of recursive data can only be performed with Mendler-style combinators to ensure strong normalization.

2.1 Two-level types

Non recursive datatypes are introduced by the **data** declaration. The data declaration can include arguments. The kind and separation of arguments into parameters and a indices is inferred. For example, the three non-recursive data types, *Bool*, *Either*, and *Maybe*, familiar to many functional programmers, are introduced by declaring the kind of the type, and the type of each of the constructors. This is similar to the way GADTs are introduced in Haskell.

data $Bool : *$	**data** $Either : * \to * \to *$	**data** $Maybe : * \to *$
where	**where**	**where**
$False : Bool$	$Left : a \to Either\ a\ b$	$Nothing : Maybe\ a$
$True : Bool$	$Right : b \to Either\ a\ b$	$Just : a \to Maybe\ a$

Note the kind information ($Bool : *$) declares *Bool* to be a type, ($Either : * \to * \to *$) declares *Either* to be a type constructor with two type arguments, and ($Maybe : * \to *$) declares *Maybe* to be a type constructor with one type argument.

To introduce a recursive type, we first introduce a non recursive datatype that uses a parameter where the usual recursive components occur. By design, normal parameters of the introduced type are written first (a in L below) and the type argument to stand for the recursive component is written last (the r of N, and the r of L below).

-- The fixpoint of N will	-- The fixpoint of $(L\ a)$ will
-- be the natural numbers.	-- be the polymorphic lists
data $N : * \to *$ **where**	**data** $L : * \to * \to *$ **where**
$Zero : N\ r$	$Nil\ \ : L\ a\ r$
$Succ : r \to N\ r$	$Cons : a \to r \to L\ a\ r$

A recursive type can be defined as the fixpoint of a (perhaps partially applied) non-recursive type constructor. Thus the traditional natural numbers are typed by $\mu^{[*]}\ N$ and the traditional lists with components of type a are typed by $\mu^{[*]}\ (L\ a)$. Note that the recursive type operator $\mu^{[\kappa]}$ is itself specialized with a kind argument inside square brackets ($[\kappa]$). The recursive type $(\mu^{[\kappa]}\ f)$ is well kinded only if the operand f has kind $\kappa \to \kappa$, in which case the recursive type $(\mu^{[k]}\ f)$ has kind κ. Since both N and $(L\ a)$ have kind $* \to *$, the recursive types $\mu^{[*]}\ N$ and $\mu^{[*]}\ (L\ a)$ have kind $*$. That is, they are both types, not type constructors.

2.2 Creating values

Values of a particular data type are created by use of constructor functions. For example *True* and *False* are nullary constructors (or, constants) of type *Bool*. (*Left* 4) is a value of type (*Either Int a*). Values of recursive types (i.e., those values with types such as $(\mu^{[k]}\ f)$ are formed by using the special $\mathsf{In}^{[\kappa]}$ constructor expression. Thus *Nil* has type *L a* and $(\mathsf{In}^{[*]}\ Nil)$ has type $(\mu^{[*]}\ (L\ a))$. In general, applying the operator $\mathsf{In}^{[k]}$ injects a term of type $f\ (\mu^{[k]}\ f)$ to the recursive type $(\mu^{[k]}\ f)$. Thus a list of *Bool* could be created using the term $(\mathsf{In}^{[*]}\ (Cons\ True\ (\mathsf{In}^{[*]}\ (Cons\ False\ (\mathsf{In}^{[*]}\ Nil)))))$. A general rule of thumb, is to apply $\mathsf{In}^{[k]}$ to terms of non-recursive type to get terms of recursive type. Writing programs using two level types, and recursive injections has definite benefits, but it surely makes programs rather annoying to write. Thus, we have provided Nax with a simple but powerful synonym (macro) facility.

2.3 Synonyms, constructor functions, and fixpoint derivation

We may codify that some type is the fixpoint of another, once and for all, by introducing a type synonym (macro).

synonym $Nat = \mu^{[*]}\ N$

synonym $List\ a = \mu^{[*]}\ (L\ a)$

In a similar manner we can introduce constructor functions that create recursive values without explicit mention of $\mathsf{In}^{\cdot}$ at their call sites (potentially many), but only at their site of definition (exactly once).

$zero = \mathsf{In}^{[*]}\ Zero$

$succ\ n = \mathsf{In}^{[*]}\ (Succ\ n)$

$nil = \mathsf{In}^{[*]}\ Nil$

$cons\ x\ xs = \mathsf{In}^{[*]}\ (Cons\ x\ xs)$

This is such a common occurrence that recursive synonyms and recursive constructor functions can be automatically derived. Automatic synonym and constructor derivation using Nax is both concise and simple. The clause "**deriving fixpoint** *List*" (below right) causes the **synonym** for *List* to be automatically defined. It also defines the constructor functions *nil* and *cons*. By convention, the constructor functions are named by dropping the initial upper-case letter in the name of the non-recursive constructors to lower-case. To illustrate, we provide side-by-side comparisons of Haskell and two different uses of Nax.

Haskell	*Nax with synonyms*	*Nax with derivation*
data *List a*	**data** $L : * \to * \to *$ **where**	**data** $L : * \to * \to *$ **where**
$= Nil$	$Nil : L\ a\ r$	$Nil : L\ a\ r$
$\mid Cons\ a\ (List\ a)$	$Cons : a \to r \to L\ a\ r$	$Cons : a \to r \to L\ a\ r$
	synonym $List\ a = \mu^{[*]}\ (L\ a)$	**deriving fixpoint** *List*
	$nil = \mathsf{In}^{[*]}\ Nil$	
	$cons\ x\ xs = \mathsf{In}^{[*]}\ (Cons\ x\ xs)$	
$x = Cons\ 3\ (Cons\ 2\ Nil)$	$x = cons\ 3\ (cons\ 2\ nil)$	$x = cons\ 3\ (cons\ 2\ nil)$

2.4 Mendler combinators for non-indexed types

There are no restrictions on what kind of datatypes can be defined in Nax. There are also no restrictions on the creation of values. Values are created using constructor functions, and the recursive injection ($\mathsf{In}^{[k]}$). To ensure strong normalization, analysis of constructed values has some restrictions. Values of non-recursive types can be freely analysed using pattern matching. Values of recursive types must be analysed using one of the Mendler-style combinators. By design, we limit pattern matching to values of non-recursive types, by *not* providing any mechanism to match against the recursive injection ($\mathsf{In}^{[k]}$).

To illustrate simple pattern matching over non-recursive types, we give a Nax multi-clause definition for the $\neg$ function over the (non-recursive) *Bool* type, and a function that strips off the *Just* constructor over the (non-recursive) *Maybe* type using a case expression.

$$\begin{array}{l|l} \neg\ \mathit{True} = \mathit{False} & \mathit{unJust0}\ x = \mathbf{case}^{\{\}}\ x\ \mathbf{of}\ \mathit{Just}\ x \to x \\ \neg\ \mathit{False} = \mathit{True} & \qquad\qquad\qquad\quad \mathit{Nothing} \to 0 \end{array}$$

Analysis of recursive data is performed with Mendler-style combinators. In our implementation we provide 5 Mendler-style combinators: $\mathsf{MIt}^{\cdot}$ (fold or catamorphism or iteration), $\mathsf{MPr}^{\cdot}$ (primitive recursion), $\mathsf{McvIt}^{\cdot}$ (courses of values iteration), and $\mathsf{McvPr}^{\cdot}$ (courses of values primitive recursion), and $\mathsf{MsfIt}^{\cdot}$ (fold or catamorphism or iteration for recursive types with negative occurrences).

A Mendler-style combinator is written in a manner similar to a case expression. A Mendler-style combinator expression contains patterns, and the variables bound in the patterns are scoped over a term. This term is executed if the pattern matches. A mendler-style combinator expression differs from a case expression in that it also introduces additional names (or variables) into scope. These variables play a role similar in nature to the operations of an abstract datatype, and provide additional functionality in addition to what can be expressed using just case analysis.

For a visual example, compare the **case** expression to the $\mathsf{MIt}^{\cdot}$ expression. In the **case**, each *clause* following the **of** indicates a possible match of the scrutinee x. In the $\mathsf{MIt}^{\cdot}$, each *equation* following the **with**, binds the variable f, and matches the pattern to a value related to the scrutinee x.

$$\begin{array}{rl|l} \mathbf{case}^{\{\}}\ x\ \mathbf{of}\ \mathit{Nil} & \to e_1 & \mathsf{MIt}^{\{\}}\ x\ \mathbf{with}\ f\ (\mathit{Cons}\ x\ xs) = e_1 \\ \quad \mathit{Cons}\ x\ xs & \to e_2 & \qquad\qquad\quad f\ \mathit{Nil} = e_2 \end{array}$$

The number and type of the additional variables depends upon which family of Mendler combinators is used to analyze the scrutinee. Each equation specifies (a potential) computation in an abstract datatype depending on whether the pattern matches. For the $\mathsf{MIt}^{\cdot}$ combinator (above) the abstract datatype has the following form. The scrutinee, x is a value of some recursive type ($\mu^{[*]}\ T$) for a non-recursive type constructor T. In each clause, the pattern has type ($T\ r$), for some abstract type r. The additional variable introduced (f) is an operator over the abstract type, r, that can safely manipulate only abstract values of type r.

Different Mendler-style combinators are implemented by different abstract types. Each abstraction safely describes a class of provably terminating computations over a recursive type. The number (and type) of abstract operations differs from one family of Mendler combinators to another. We give descriptions of three families of Mendler combinators, their abstractions, and the types of the operators within the abstraction, below. In each description, the type *ans* represents the result type, when the Mendler combinator is fully applied.

$\mathsf{MIt}^{\{\,\}}\ x$ **with**
$\quad f\ \ p_i = e_i$

$x\ : \mu^{[*]}\ T$
$f\ : r \to ans$

$p_i : T\ r$
$e_i : ans$

$\mathsf{MIt}^{\{\psi\}}\ \varphi\ (\mathsf{In}^{[*]}\ x)$
$\quad = \varphi\ (\mathsf{MIt}^{\{\psi\}}\ \varphi)\ x$

$\mathsf{MPr}^{\{\,\}}\ x$ **with**
$\quad f\ \ cast\ \ p_i = e_i$

$x\ : \mu^{[*]}\ T$
$f\ : r \to ans$
$cast : r \to \mu^{[*]}\ T$
$p_i : T\ r$
$e_i : ans$

$\mathsf{MPr}^{\{\psi\}}\ \varphi\ (\mathsf{In}^{[*]}\ x)$
$\quad = \varphi\ (\mathsf{MPr}^{\{\psi\}}\ \varphi)\ (\mathsf{In}^{[*]})\ x$

$\mathsf{McvIt}^{\{\,\}}\ x$ **with**
$\quad f\ \ project\ \ p_i = e_i$

$x\ \ : \mu^{[*]}\ T$
$f\ \ : r \to ans$
$project : r \to T\ r$
$p_i\ \ : T\ r$
$e_i\ \ : ans$

$\mathsf{McvIt}^{\{\psi\}}\ \varphi\ (\mathsf{In}^{[*]}\ x)$
$\quad = \varphi\ (\mathsf{McvIt}^{\{\psi\}}\ \varphi)\ out\ x$
$\quad$ **where** $out\ (\mathsf{In}^{[*]}\ x) = x$

A Mendler-style combinator implements a (provably terminating) recursive function applied to the scrutinee. The abstract type and its operations ensure termination. Note that every operation above includes an abstract operator, $f : r \to ans$. This operation represents a recursive call in the function defined by the Mendler-style combinator. Other operations, such as *cast* and *project*, support additional functionality within the abstraction in which they are defined ($\mathsf{MPr}^{\cdot}$ and $\mathsf{McvIt}^{\cdot}$ respectively). The equations at the bottom of each section provide an operational understanding of how the operator works. These can be safely ignored until after we see some examples of how a Mendler-style combinator works in practice.

$$
\begin{array}{lll}
length\ y & = \mathsf{MIt}^{\{\,\}}\ y & \textbf{with}\ len\ Nil = zero \\
 & & \quad len\ (Cons\ x\ xs) = (succ\ zero) + len\ xs \\
tail\ x & = \mathsf{MPr}^{\{\,\}}\ x & \textbf{with}\ tl\ cast\ Nil = nil \\
 & & \quad tl\ cast\ (Cons\ y\ ys) = cast\ ys \\
factorial\ x & = \mathsf{MPr}^{\{\,\}}\ x & \textbf{with}\ fact\ cast\ Zero = succ\ zero \\
 & & \quad fact\ cast\ (Succ\ n) = times\ (succ\ (cast\ n))\ (fact\ n) \\
fibonacci\ x & = \mathsf{McvIt}^{\{\,\}}\ x & \textbf{with}\ fib\ out\ Zero = succ\ zero \\
 & & \quad fib\ out\ (Succ\ n) = \textbf{case}^{\{\,\}}\ (out\ n)\ \textbf{of} \\
 & & \qquad Zero \to succ\ zero \\
 & & \qquad Succ\ m \to fib\ n + fib\ m
\end{array}
$$

The *length* function uses the simplest kind of recursion where each recursive call is an application to a direct subcomponent of the input. Operationally,

length works as follows. The scrutinee, y, has type $(\mu^{[*]}\ (L\ a))$, and has the form $(\mathsf{In}^{[*]}\ x)$. The type of y implies that x must have the form *Nil* or $(Cons\ x\ xs)$. The $\mathsf{MIt}^{\cdot}$ strips off the $\mathsf{In}^{[*]}$ and matches x against the *Nil* and $(Cons\ x\ xs)$ patterns. If the *Nil* pattern matches, then 0 is returned. If the $(Cons\ x\ xs)$ pattern matches, x and xs are bound. The abstract type mechanism gives the pattern $(Cons\ x\ xs)$ the type $(L\ a\ r)$, so $(x : a)$ and $(xs : r)$ for some abstract type r. The abstract operation, $(len : r \to Int)$, can safely be applied to xs, obtaining the length of the tail of the original list. This value is incremented, and then returned. The $\mathsf{MIt}^{\cdot}$ abstraction provides a safe way to allow the user to make recursive calls, *len*, but the abstract type, r, limits its use to direct subcomponents, so termination is guaranteed.

Some recursive functions need direct access, not only to the direct subcomponents, but also the original input as well. The Mendler-style combinator $\mathsf{MPr}^{\cdot}$ provides a safe, yet abstract mechanism, to support this. The Mendler $\mathsf{MPr}^{\cdot}$ abstraction provides two abstract operations. The recursive caller with type $(r \to ans)$ and a casting function with type $(r \to \mu^{[k]}\ T)$. The casting operation allows the user to recover the original type from the abstract type r, but since the recursive caller only works on the abstract type r, the user cannot make a recursive call on one of these cast values. The functions *factorial* (over the natural numbers) and *tail* (over lists) are both defined using $\mathsf{MPr}^{\cdot}$.

Note how in *factorial* the original input is recovered (in constant time) by taking the successor of casting the abstract predecessor, n. In the *tail* function, the abstract tail, ys, is cast to get the answer, and the recursive caller is not even used.

Some recursive functions need direct access, not only to the direct subcomponents, but even deeper subcomponents. The Mendler-style combinator $\mathsf{McvIt}^{\cdot}$ provides a safe, yet abstract mechanism, to support this. The function *fibonacci* is a classic example of this kind of recursion. The Mendler $\mathsf{McvIt}^{\cdot}$ provides two abstract operations. The recursive caller with type $(r \to ans)$ and a projection function with type $(r \to T\ r)$. The projection allows the programmer to observe the hidden T structure inside a value of the abstract type r. In the *fibonacci* function above, we name the projection *out*. It is used to observe if the abstract predecessor, n, of the input, x, is either zero, or the successor of the second predecessor, m, of x. Note how recursive calls are made on the direct predecessor, n, and the second predecessor, m.

Each recursion combinator can be defined by the equation at the bottom of its figure. Each combinator can be given a naive type involving the concrete recursive type $(\mu^{[*]}\ T)$, but if we instead give it a more abstract type, abstracting values of type $(\mu^{[*]}\ T)$ into some unknown abstract type r, one can safely guarantee a certain pattern of use that insures termination. Informally, if the combinator works for some unknown type r it will certainly also work for the actual type $(\mu^{[*]}\ T)$, but because it cannot assume that r has any particular structure, the user is forced to use the abstract operations in carefully proscribed ways.

2.5 Types with static indices

Recall that a type can have both parameters and indices, and that indices can be either types or terms. We define three types below each with one or more indices. Each example defines a non-recursive type, and then uses derivation to define synonyms for its fix point and recursive constructor functions. By convention, in each example, the argument that abstracts the recursive components is called r. By design, arguments appearing before r are understood to be parameters, and arguments appearing after r are understood to be indices. To define a recursive type with indices, it is necessary to give the argument, r, a higher-order kind. That is, r should take indices as well, since it abstracts over a recursive type which takes indices.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nest} : (* \to *) \to * \to * \to * \ \textbf{where} \\
\quad \mathit{Tip} \quad : a \to \mathit{Nest}\ r\ a \\
\quad \mathit{Fork} \quad : r\ (a, a) \to \mathit{Nest}\ r\ a \\
\quad\quad \textbf{deriving fixpoint}\ \mathit{PowerTree} \\
\textbf{data}\ V : * \to (\mathit{Nat} \to *) \to \mathit{Nat} \to * \ \textbf{where} \\
\quad \mathit{Vnil} \quad : V\ a\ r\ \{\text{`}\mathit{zero}\} \\
\quad \mathit{Vcons} : a \to r\ \{n\} \to V\ a\ r\ \{\text{`}\mathit{succ}\ n\} \\
\quad\quad \textbf{deriving fixpoint}\ \mathit{Vector} \\
\textbf{data}\ \mathit{Tag} = E \mid O \\
\textbf{data}\ P : (\mathit{Tag} \to \mathit{Nat} \to *) \to \mathit{Tag} \to \mathit{Nat} \to * \ \textbf{where} \\
\quad \mathit{Base} \quad : P\ r\ \{E\}\ \{\text{`}\mathit{zero}\} \\
\quad \mathit{StepO} : r\ \{O\}\ \{i\} \to P\ r\ \{E\}\ \{\text{`}\mathit{succ}\ i\} \\
\quad \mathit{StepE} : r\ \{E\}\ \{i\} \to P\ r\ \{O\}\ \{\text{`}\mathit{succ}\ i\} \\
\quad\quad \textbf{deriving fixpoint}\ \mathit{Proof}
\end{array}
$$

Note, to distinguish type indices from term indices (and to make parsing unambiguous), we enclose term indices in braces ($\{...\}$). We also backquote (`) variables in terms that we expect to be bound in the current environment. Unbackquoted variables are taken to be universally quantified. By backquoting *succ*, we indicate that we want terms, which are applications of the successor function, but not some universally quantified function variable[5]. For non-recursive types without parameters, the kind of the fixpoint is the same as the kind of the recursive argument r. If the non-recursive type has parameters, the kind of the fixpoint will be composed of the parameters $\to$ the kind of the recursive argument r. For example, study the kinds of the fixpoints for the non-recursive types declared above in the table below.

[5] In the design of Nax we had a choice. Either, explicitly declare each universally quantified variable, or explicitly mark those variables not universally quantified. Since quantification is much more common than referring to variables already in scope, the choice was easy.

non-recursive type	*Nest*	*V*	*P*
recursive type	*PowerTree*	*Vector*	*Proof*
kind of T	$* \to *$	$* \to Nat \to *$	$Tag \to Nat \to *$
kind of r	$* \to *$	$Nat \to *$	$Tag \to Nat \to *$
number of parameters	0	1	0
number of indices	1 (type)	1 (term)	2 (term,term)

Recall, indices are used to track static properties about values with those types. A well-formed (*PowerTree* x) contains a balanced set of parenthesized binary tuples of elements. The index, x, describes what kind of values are nested in the parentheses. The invariant is that the number of items nested is always an exact power of 2. A (*Vector* a $\{n\}$) is a list of elements of type a, with length exactly equal to n, and a (*Proof* $\{E\}$ $\{n\}$) witnesses that the natural number n is even, and a (*Proof* $\{O\}$ $\{m\}$) witnesses that the natural number m is odd. Some example value with these types are given below.

$$
\begin{aligned}
&\mathit{tree1} : \mathit{PowerTree}\ \mathit{Int} = \mathit{tip}\ 3 \\
&\mathit{tree2} : \mathit{PowerTree}\ \mathit{Int} = \mathit{fork}\ (\mathit{tip}\ (2,5)) \\
&\mathit{tree3} : \mathit{PowerTree}\ \mathit{Int} = \mathit{fork}\ (\mathit{fork}\ (\mathit{tip}\ ((4,7),(0,2)))) \\
&\mathit{v2} : \mathit{Vector}\ \mathit{Int}\ \{\mathit{succ}\ (\mathit{succ}\ \mathit{zero})\} = (\mathit{vcons}\ 3\ (\mathit{vcons}\ 5\ \mathit{vnil})) \\
&\mathit{p1} : P\ \{O\}\ \{\mathit{succ}\ \mathit{zero}\} \qquad\quad = \mathit{stepE}\ \mathit{base} \\
&\mathit{p2} : P\ \{E\}\ \{\mathit{succ}\ (\mathit{succ}\ \mathit{zero})\} = \mathit{stepO}\ (\mathit{stepE}\ \mathit{base})
\end{aligned}
$$

Note that in the types of the terms above, the indices in braces ($\{...\}$) are ordinary terms (not types). In these example we use natural numbers (e.g., *succ* (*succ zero*)) and elements (E and O) of the two-valued type *Tag*. It is interesting to note that sometimes the terms are of recursive types (e.g., *Nat* which is a synonym for $\mu^{[*]}\ N$), and some are non-recursive types (e.g., *Tag*).

2.6 Mendler-style combinators for indexed types

Mendler-style combinators generalize naturally to indexed types. The key observation that makes this generalization possible is that the types of the operations within abstraction have to be generalized to deal with the type indices in a consistent manner. How this is done is best first explained by example, and then later abstracted to its full general form.

Recall, a value of type (*PowerTree Int*) is a set of integers. This set is constructed as a balanced binary tree with pairs at the leaves (see *tree2* and *tree3* above). The number of integers in the set is an exact power of 2. Consider a function that adds up all those integers. One wants a function of type (*PowerTree Int* $\to$ *Int*). One strategy to writing this function is to write a more general function of type (*PowerTree* $a \to (a \to \mathit{Int}) \to \mathit{Int}$). In Nax, we can do this as follows:

$$
\begin{aligned}
\mathit{genericSum}\ t = \mathsf{MIt}^{\{\,a.\,(a \to \mathit{Int}) \to \mathit{Int}\}}\ t\ \ &\textbf{with} \\
\mathit{sum}\ (\mathit{Tip}\ x)\ \ &= \lambda f \to f\ x
\end{aligned}
$$

$$sum\ (Fork\ x) = \lambda f \to sum\ x\ (\lambda(a,b) \to f\ a + f\ b)$$
$$sumTree\ t = genericSum\ t\ (\lambda x \to x)$$

In general, the type of the result of a function over an indexed type, can depend upon what the index is. Thus, a Mendler-style combinator over a value with an indexed type, must be type-specialized to that value's index. Different values of the same general type, will have different indices. After all, the role of an index is to witness an invariant about the value, and different values might have different invariants. Capturing this variation is the role of the clause $\{\, a.\ (a \to Int) \to Int \,\}$ following the keyword $\mathsf{MIt}^{\cdot}$. We call such a clause, an *index transformer*. In the same way that the type of the result depends upon the index, the type of the different components of the abstract datatype implementing the Mendler-style combinator also depend upon the index. In fact, everything depends upon the index in a uniform way. The index transformer captures this uniformity. One cannot abstract over the index transformer in Nax. Each Mendler-style combinator, over an indexed type, must be supplied with a concrete clause (inside the braces) that describe how the results depend upon the index. To see how the transformer is used, study the types of the terms in the following paragraph. Can you see the relation between the types and the transformer?

The scrutinee t has type $(PowerTree\ a)$ which is a synonym for $((\mu^{[*\to *]}\ Nest)\ a)$. The recursive caller *sum* has type $(\forall\ a\ .\ r\ a \to (a \to Int) \to Int)$, for some abstract type constructor r. Recall r has an index, so it must be a type constructor, not a type. The patterns $(Tip\ x)$ and $(Fork\ x)$ have type $(Nest\ r\ a)$ and the right hand sides of the equations: $(\lambda f \to f\ x)$ and $(\lambda f \to sum\ x\ (\lambda(a,b) \to f\ a + f\ b))$, have type $((a \to Int) \to Int)$. Note that the dependency of $((a \to Int) \to Int)$ on the index a, appears in both the result type, and the type of the recursive caller. If we think of an index transformer, like $\{\, a\ .\ (a \to Int) \to Int \,\}$, as a function: $\psi\ a = (a \to Int) \to Int$, we can succinctly describe the types of the abstract operations in the $\mathsf{MIt}^{\cdot}$ Mendler abstraction. In the table below, we put the general case on the left, and terms from the *genericSum* example, that illustrate the general case, on the right.

$\mathsf{MIt}^{\{\psi\}}\ x$ **with**
$f\ p_i = e_i$

$\psi : \kappa \to *$	$\{\, a\ .\ (a \to Int) \to Int \,\}$	$: * \to *$
$T : (\kappa \to *) \to \kappa \to *$	$Nest$	$: (* \to *) \to * \to *$
$x : (\mu^{[\kappa \to *]}\ T)\ a$	t	$: (\mu^{[* \to *]}\ Nest)\ a$
$f : \forall\ (a : \kappa)\ .\ r\ a \to \psi\ a$	sum	$: \forall\ (a : *)\ .\ r\ a \to (a \to Int) \to Int$
$p_i : T\ r\ a$	$Fork\ x$	$: Nest\ r\ a$
$e_i : \psi\ a$	$\lambda f \to f\ x$	$: (a \to Int) \to Int$

The same scheme for $\mathsf{MIt}^{\cdot}$ generalizes to type constructors with term indices, and with multiple indices. To illustrate this we give the generic schemes for type constructors with 2 or 3 indices. In the table the variables κ_1, κ_2, and κ_3, stand

for arbitrary kinds (either kinds for types, like $*$, or kinds for terms, like *Nat* or *Tag*).

$$
\begin{array}{l|l}
T : (\kappa_1 \to \kappa_2 \to *) \to (\kappa_1 \to \kappa_2 \to *) & T : (\kappa_1 \to \kappa_2 \to \kappa_3 \to *) \to (\kappa_1 \to \kappa_2 \to \kappa_3 \to^*) \\
\psi \ : \kappa_1 \to \kappa_2 \to * & \psi \ : \kappa_1 \to \kappa_2 \to \kappa_3 \to * \\
x \ : (\mu^{[\kappa_1 \to \kappa_2 \to *]} \ T) \ a \ b & x \ : (\mu^{[\kappa_1 \to \kappa_2 \to \kappa_3 \to *]} \ T) \ a \ b \ c \\
f \ : \forall \ (a : \kappa_1) \ (b : \kappa_2) \ . \ r \ a \ b \to \psi \ a \ b & f \ : \forall \ (a : \kappa_1) \ (b : \kappa_2) \ (c : \kappa_3) \ . \ r \ a \ b \ c \to \psi \ a \ b \ c \\
p_i : T \ r \ a \ b & p_i : T \ r \ a \ b \ c \\
e_i : \psi \ a \ b & e_i : \psi \ a \ b \ c
\end{array}
$$

The simplest form of index transformation, is where the transformation is a constant function. This is the case of the function that computes the integer length of a natural-number, length-indexed, list (what we called a *Vector*). Independent of the length the result is an integer. Such a function has type: $Vector \ a \ \{n\} \to Int$. We can write this as follows:

$$
\begin{array}{lll}
vlen \ x = \mathsf{MIt}^{\{\{i\}.\ Int\}} \ x \ \textbf{with} & len \ Vnil & = 0 \\
& len \ (Vcons \ x \ xs) & = 1 + len \ xs
\end{array}
$$

Let's study an example with a more interesting index transformation. A term with type $(Proof \ \{E\} \ \{n\})$, which is synonymous with the type $(\mu^{[\,Tag \to Nat \to *]} \ P \ \{E\} \ \{n\})$, witnesses that the term n is even. Can we transform such a term into a proof that $n + 1$ is odd? We can generalize this by writing a function which has both of the types below:
$Proof \ \{E\} \ \{n\} \to Proof \ \{O\} \ \{`succ \ n\}$, and
$Proof \ \{O\} \ \{n\} \to Proof \ \{E\} \ \{`succ \ n\}$.
We can capture this dependency by defining the term-level function *flip*, and using an $\mathsf{MIt}^{\cdot}$ with the index transformer: $\{\{t\} \ \{i\} \ . \ Proof \ \{`flip \ t\} \ \{`succ \ i\}\}$.

$$
\begin{array}{l}
flip \ E = O \\
flip \ O = E \\
flop \ x = \mathsf{MIt}^{\{\{t\} \ \{i\}.\ Proof \ \{`flip \ t\} \ \{`succ \ i\}\}} \ x \ \textbf{with} \\
\qquad \begin{array}{ll}
f \ Base & = stepE \ base \\
f \ (StepO \ p) & = stepE \ (f \ p) \\
f \ (StepE \ p) & = stepO \ (f \ p)
\end{array}
\end{array}
$$

For our last term-indexed example, every length-indexed list has a length, which is either even or odd. We can witness this fact by writing a function with type: $Vector \ a \ \{n\} \to Either \ (Even \ \{n\}) \ (Odd \ \{n\})$. Here, *Even* and *Odd* are synonyms for particular kinds of *Proof*. To write this function, we need the index transformation: $\{\{n\} \ . \ Either \ (Even \ \{n\}) \ (Odd \ \{n\})\}$.

```
synonym Even {x} = Proof {E} {x}
synonym Odd  {x} = Proof {O} {x}
proveEvenOrOdd x = MIt^{{n}. Either (Even {n}) (Odd {n})} x with
                     prEOO Vnil          = Left base
                     prEOO (Vcons x xs) = case^{} prEOO xs of
                                            Left p  -> Right (stepE p)
                                            Right p -> Left (stepO p)
```

2.7 Recursive types of unrestricted polarity but restricted elimination

In Nax, programmers can define recursive data structures with both positive and negative polarity. The classic example is a datatype encoding the syntax of λ-calculus, which uses higher-order abstract syntax (HOAS). Terms in the λ-calculus are either variables, applications, or abstractions. In a HOAS representation, one uses Nax functions to encode abstractions. We give a two level description for recursive λ-calculus *Term*s, by taking the fixpoint of the non-recursive *Lam* datatype.

```
data Lam : * -> * where
  App :: r -> r -> Lam r
  Abs :: (r -> r) -> Lam r
    deriving fixpoint Term
apply = abs (\f -> abs (\x -> app f x))
```

Note that we don't need to include a constructor for variables, as variables are represented by Nax variables, bound by Nax functions. For example the lambda term: $(\lambda f.\lambda x.f\ x)$ is encoded by the Nax term *apply* above.

Note also, the recursive constructor: $abs : (Term \to Term) \to Term$, introduced by the **deriving fixpoint** clause, has a negative occurrence of the type *Term*. In a language with unrestricted analysis, such a type could lead to non-terminating computations. The Mendler MIt$^{\cdot}$ and MPr$^{\cdot}$ combinators limit the analysis of such types in a manner that precludes non-terminating computations. The Mendler-style combinator, McvIt$^{\cdot}$, is too expressive to exclude non-terminating computations, and must be restricted to recursive datatypes with no negative occurrences.

Even though MIt$^{\cdot}$ and MPr$^{\cdot}$ allow us to safely operate on values of type *Term*, they are not expressive enough to write many interesting functions. Fortunately, there is a more expressive Mendler-style combinator that is safe over recursive types with negative occurrences. We call this combinator MsfIt$^{\cdot}$. This combinator is based upon an interesting programming trick, first described by Sheard and Fegaras [7], hence the "sf" in the name MsfIt$^{\cdot}$. The abstraction supported by MsfIt$^{\cdot}$ is as follows:

$$\begin{array}{l|l} \mathsf{Msflt}^{\{\}}\ x\ \textbf{with} & x\ \ : \mu^{[*]}\ T \\ \quad f\ inv\ p_i = e_i & f\ \ : r \to ans \\ & inv : ans \to r \\ & p_i\ : T\ r \\ & e_i\ : ans \end{array}$$

To use $\mathsf{Msflt}^{\cdot}$ the inverse allows one to cast an answer into an abstract value. To see how this works, study the function that turns a *Term* into a string. The strategy is to write an auxiliary function, *showHelp* that takes an extra integer argument. Every time we encounter a lambda abstraction, we create a new variable, `x`n (see the function *new*), where n is the current value of the integer variable. When we make a recursive call, we increment the integer. In the comments (the rest of a line after --), we give the type of a few terms, including the abstract operations *sh* and *inv*.

```
                                  -- cat : List String → String
                                  -- new : Int → String
new n = cat ["x", show n]
                                  -- showHelp : Term → (Int → String)
                                  -- sh : r → (Int → String)
                                  -- inv : (Int → String) → r
                                  -- (λn → new m) : Int → String
showHelp x =
  Msflt{} x with
    sh inv (App x y) = λm → cat ["(", sh x m, " ", sh y m, ")"]
    sh inv (Abs f)   = λm → cat ["(fn ", new m, " => ",
                                 sh (f (inv (λn → new m))) (m + 1), ")"]
showTerm x = showHelp x 0
showTerm apply : List Char = "(fn x0 => (fn x1 => (x0 x1)))"
```

The final line of the example above illustrates applying *showTerm* to *apply*. Recall that $apply = abs\ (\lambda f \to abs\ (\lambda x \to app\ f\ x))$, which is the HOAS representation of the λ-calculus term $(\lambda f.\lambda x.f\ x)$.

2.8 Lessons from Nax

Nax is our first attempt to build a strongly normalizing, sound and consistent logic, based upon Mendler-style iteration. We would like to emphasize the lessons we learned along the way.

- Writing types as the fixed point of a non-recursive type constructor is quite expressive. It supports a wide variety of types including the regular types (*Nat* and *List*), nested types (*PowerTree*), GADTs (*Vector*), and mutually recursive types (*Even* and *Odd*).
- Two-level types, while expressive, are a pain to program with (all those $\mu^{[\kappa]}$ and $\mathsf{In}^{[\kappa]}$ annotations), so a strong synonym or macro facility is necessary. With syntactic support, one hardly even notices.

– The use of term-indexed types allows programmers to write types that act as logical relations, and form the basis for reasoning about programs. We have formalized this is the paper *System* F_i*: a higher-order polymorphic* λ*-calculus with erasable term indices*[2] which we have submitted to POPL.
– Using Mendler-style combinators is expressive, and with syntactic support (the **with** equations of the Mendler combinators), is easy to use. In fact Nax programs are often no more complicated than their Haskell counterparts.
– Type inference is an important feature of a programming language. We hope you noticed, that apart from index transformers, no type information is supplied in any of the Nax examples. The Nax compiler reconstructs all type information.
– Index transformers are the minimal information needed to extend Hindley-Milner type inference over GADTs. One can always predict where they are needed, and the compiler can enforce that the programmer supplies them. They are never needed for non-indexed types. Nax faithfully extends Hindley-Milner type inference.

3 Nax

We describe Nax by comparing it to System F_i, which is the calculus Nax is based on (see §4 for additional information). Along with this submission, a draft of the paper on System F_i, which we submitted to POPL recently, is included as a supplementary material. For simplicity, we Nax with just one Mendler-style combinator MIt.

3.1 Language definition

The Nax language definition is described in Fig. 3 and Fig. 4. Fig. 3 illustrates syntax, reduction rules, and well-formedness conditions for typing contexts. Fig. 4 illustrates sorting, kinding, and typing rules.

Typing contexts The typing context of Nax is separated into three zones. In addition to the two zones of F_i(the type level context Δ and the term level context Γ), we have top level contexts (Σ). The top level contexts can contain three kinds of bindings: type constructor bindings ($T : \kappa$), data constructor bindings ($C : \sigma$), and top level variable bindings ($`x : \sigma$). These bindings are introduced from declarations ($D$). Type constructor bindings ($T : \kappa$) and data constructor bindings ($C : \sigma$) are introduced from datatype declarations (**data** $T : \ldots$ **where** $\ldots$). Top level variable bindings ($`x : \sigma$) are introduced from top level definitions ($`x = t$). The rules for well-formed contexts in Nax are similar to those rules in F_i.

Kinds and their sorting rules The kind syntax of Nax is exactly the same as the kind syntax of F_i. The sorting rules are the same as F_i except we judge the sorts of kinds under the top level context (Σ).

Syntax:

$$
\begin{array}{lrcl}
\text{Kinds} & \kappa & ::= & * \mid \kappa \to \kappa \mid A \to \kappa \\
\text{Type Constructors} & F, G, A, B & ::= & X \mid T \mid \mu^{\kappa}(T\,\overline{\tau}) \mid F\ G \mid F\,\{s\} \mid A \to B \\
\text{Type Schemes} & \sigma & ::= & A \mid \forall X.\sigma \mid \forall i.\sigma \\
\text{Terms} & r, s, t & ::= & x \mid \text{`}x \mid i \mid \lambda x.t \mid r\ s \mid \mathbf{let}\ x = s\ \mathbf{in}\ t \mid \varphi^{\psi} \mid \mathsf{Mlt}\ x.\varphi^{\psi} \mid \mathsf{In}^{\kappa} \\
\text{Program} & \mathit{Prog} & ::= & \overline{D}; t \\
\text{Declarations} & D & ::= & \mathbf{data}\ T : \overline{K} \to * \ \mathbf{where}\ \overline{C : \overline{A} \to T\,\overline{\tau}} \quad \mid \text{`}x = t \\
\text{List of Declarations} & \overline{D} & ::= & \cdot \mid D, \overline{D} \\
\text{Kind Arguments} & K & ::= & \kappa \mid A \\
\text{Type Arguments} & \tau & ::= & G \mid \{s\} \\
\text{Type Argument Variables} & \iota & ::= & X \mid i \\
\text{Index Transformers} & \psi & ::= & \cdot \mid \overline{\iota}.B \\
\text{Case Branches} & \varphi & ::= & \overline{C\,\overline{x} \to t} \\
\text{Contexts} & \Sigma & ::= & \cdot \mid \Sigma, T : \kappa \mid \Sigma, C : \sigma \mid \Sigma, \text{`}x : \sigma \\
 & \Delta & ::= & \cdot \mid \Delta, X^{\kappa} \mid \Delta, i^{\sigma} \\
 & \Gamma & ::= & \cdot \mid \Gamma, x : \sigma
\end{array}
$$

Well-formed contexts:

$$\boxed{\vdash \Sigma} \qquad \frac{}{\vdash \cdot} \qquad \frac{\vdash \Sigma \qquad \Sigma \vdash \kappa : \square}{\vdash \Sigma, T : \kappa}\big(T \notin \mathsf{dom}(\Sigma)\big)$$

$$\frac{\vdash \Sigma \qquad \Sigma; \cdot \vdash \sigma : *}{\vdash \Sigma, C : \sigma}\big(C \notin \mathsf{dom}(\Sigma)\big) \qquad \frac{\vdash \Sigma \qquad \Sigma; \cdot \vdash \sigma : *}{\vdash \Sigma, \text{`}x : \sigma}\big(\text{`}x \notin \mathsf{dom}(\Sigma)\big)$$

$$\boxed{\Sigma \vdash \Delta} \quad \frac{\vdash \Sigma}{\Sigma \vdash \cdot} \qquad \frac{\Sigma \vdash \Delta \qquad \Sigma \vdash \kappa : \square}{\Sigma \vdash \Delta, X^{\kappa}}\big(X \notin \mathsf{dom}(\Delta)\big) \qquad \frac{\Sigma \vdash \Delta \qquad \Sigma; \cdot \vdash \sigma : *}{\Sigma \vdash \Delta, i^{\sigma}}\big(i \notin \mathsf{dom}(\Delta)\big)$$

$$\boxed{\Sigma; \Delta \vdash \Gamma} \qquad \frac{\Sigma \vdash \Delta}{\Sigma; \Delta \vdash \cdot} \qquad \frac{\Sigma; \Delta \vdash \Gamma \qquad \Sigma; \Delta \vdash A : *}{\Sigma; \Delta \vdash \Gamma, x : A}\big(x \notin \mathsf{dom}(\Gamma)\big)$$

Reduction: $\boxed{t \rightsquigarrow t'}$

$$\frac{}{(\lambda x.t)\ s \rightsquigarrow [s/x]t} \qquad \frac{}{\mathbf{let}\ x = s\ \mathbf{in}\ t \rightsquigarrow [s/x]t}$$

$$\frac{C\,\overline{x} \to t \in \varphi}{\varphi^{\psi}(C\,\overline{t}) \rightsquigarrow [\overline{t}/\overline{x}]t} \qquad \frac{}{\mathsf{Mlt}\ x.\varphi^{\psi}\ (\mathsf{In}^{\kappa} t) \rightsquigarrow [\mathsf{Mlt}\ x.\varphi^{\psi}/x]\varphi^{\psi}\ t} \qquad \frac{\text{`}x = t \in \overline{D}}{\text{`}x \rightsquigarrow t}$$

$$\frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'} \qquad \frac{r \rightsquigarrow r'}{r\ s \rightsquigarrow r'\ s} \qquad \frac{s \rightsquigarrow s'}{r\ s \rightsquigarrow r\ s'} \qquad \frac{t_i \rightsquigarrow t_i'}{C\ t_1 \cdots t_i \cdots t_n \rightsquigarrow C\ t_1 \cdots t_i' \cdots t_n}$$

$$\frac{s \rightsquigarrow s'}{\mathbf{let}\ x = s\ \mathbf{in}\ t \rightsquigarrow \mathbf{let}\ x = s'\ \mathbf{in}\ t} \qquad \frac{t \rightsquigarrow t'}{\mathbf{let}\ x = s\ \mathbf{in}\ t \rightsquigarrow \mathbf{let}\ x = s\ \mathbf{in}\ t'} \qquad \frac{\varphi^{\psi} \rightsquigarrow \varphi'^{\psi}}{\mathsf{Mlt}\ x.\varphi^{\psi} \rightsquigarrow \mathsf{Mlt}\ x.\varphi'^{\psi}}$$

$$\frac{t_i \rightsquigarrow t_i'}{(C_1\,\overline{x_1} \to t_1; \cdots; C_i\,\overline{x_i} \to t_i; \cdots; C_n\,\overline{x_n} \to t_n)^{\psi} \ \rightsquigarrow\ (C_1\,\overline{x_1} \to t_1; \cdots; C_i\,\overline{x_i} \to t_i'; \cdots; C_n\,\overline{x_n} \to t_n)^{\psi}}$$

Fig. 3. Syntax and Reduction rules of Nax

Sorting:

$$\boxed{\Sigma \vdash \kappa : \Box} \quad (A)\frac{}{\Sigma \vdash * : \Box} \quad (R)\frac{\Sigma \vdash \kappa : \Box \quad \Sigma \vdash \kappa' : \Box}{\Sigma \vdash \kappa \to \kappa' : \Box} \quad (Ri)\frac{\Sigma; \cdot \vdash A : * \quad \Sigma \vdash \kappa : \Box}{\Sigma \vdash A \to \kappa : \Box}$$

Kinding: $\boxed{\Sigma; \Delta \vdash \sigma : \kappa}$ $\quad (\forall)\frac{\Sigma; \Delta, X^\kappa \vdash \sigma : *}{\Sigma; \Delta \vdash \forall X.\sigma : *} \quad (\forall i)\frac{\Sigma; \Delta, i^A \vdash \sigma : *}{\Sigma; \Delta \vdash \forall i.\sigma : *}$

$$\boxed{\Sigma; \Delta \vdash F : \kappa} \quad (Var)\frac{X^\kappa \in \Delta \quad \Sigma \vdash \Delta}{\Sigma; \Delta \vdash X : \kappa} \quad (TCon)\frac{T : \kappa \in \Sigma \quad \Sigma \vdash \Delta}{\Sigma; \Delta \vdash T : \kappa} \quad (\mu)\frac{\Sigma; \Delta \vdash T\,\overline{\tau} : \kappa \to \kappa}{\Sigma; \Delta \vdash \mu^\kappa(T\,\overline{\tau}) : \kappa}$$

$$(@)\frac{\Sigma; \Delta \vdash F : \kappa \to \kappa' \quad \Sigma; \Delta \vdash G : \kappa}{\Sigma; \Delta \vdash F\,G : \kappa'} \quad (@i)\frac{\Sigma; \Delta \vdash F : A \to \kappa \quad \Sigma; \Delta; \cdot \vdash s : A}{\Sigma; \Delta \vdash F\,\{s\} : \kappa}$$

$$(\to)\frac{\Sigma; \Delta \vdash A : * \quad \Sigma; \Delta \vdash B : *}{\Sigma; \Delta \vdash A \to B : *} \quad (Conv)\frac{\Sigma; \Delta \vdash A : \kappa \quad \Sigma \vdash \kappa = \kappa' : \Box}{\Sigma; \Delta \vdash A : \kappa'}$$

Typing:

$$\boxed{\Sigma \vdash Prog : A} \quad (\cdot; t)\frac{\Sigma; \cdot; \cdot \vdash t : A}{\Sigma \vdash \cdot; t : A} \quad (D)\frac{\Sigma \vdash D \Rrightarrow \Sigma' \quad \Sigma' \vdash \overline{D};\ t : A}{\Sigma \vdash D, \overline{D};\ t : A}$$

$$\boxed{\Sigma; \Delta; \Gamma \vdash t : A} \quad (=)\frac{\Sigma; \Delta; \Gamma \vdash t : A \quad \Sigma; \Delta \vdash A = B : *}{\Sigma; \Delta; \Gamma \vdash t : B}$$

$$(:)\frac{x : \sigma \in \Gamma \quad \Sigma; \Delta \vdash A \prec \sigma \quad \Sigma; \Delta \vdash \Gamma}{\Sigma; \Delta; \Gamma \vdash x : A} \quad (: i)\frac{i^\sigma \in \Delta \quad \Sigma; \Delta \vdash A \prec \sigma \quad \Sigma; \Delta \vdash \Gamma}{\Sigma; \Delta; \Gamma \vdash i : A}$$

$$(: C)\frac{C : \sigma \in \Sigma \quad \Sigma; \Delta \vdash A \prec \sigma \quad \Sigma; \Delta \vdash \Gamma}{\Sigma; \Delta; \Gamma \vdash C : A} \quad (: \text{`})\frac{\text{`}x : \sigma \in \Sigma \quad \Sigma; \Delta \vdash A \prec \sigma \quad \Sigma; \Delta \vdash \Gamma}{\Sigma; \Delta; \Gamma \vdash \text{`}x : A}$$

$$(\to I)\frac{\Sigma; \Delta; \Gamma, x : A \vdash t : B}{\Sigma; \Delta; \Gamma \vdash \lambda x.t : A \to B} \qquad (\to E)\frac{\Sigma; \Delta; \Gamma \vdash r : A \to B \quad \Sigma; \Delta; \Gamma \vdash s : A}{\Sigma; \Delta; \Gamma \vdash r\ s : B}$$

$$(\text{let})\frac{\Sigma; \Delta, \overline{\iota^K}; \Gamma \vdash s : A \quad \Sigma; \Delta; \Gamma, x : \forall \bar{\iota}.A \vdash t : B}{\Sigma; \Delta; \Gamma \vdash \textbf{let}\ x = s\ \textbf{in}\ t : B}\ \left(\bar{\iota} \cap \mathrm{FV}(s) = \emptyset\right)\ \left(\bar{\iota} \cap \mathrm{FV}(\Gamma) = \emptyset\right) \quad (\text{case})\frac{\Sigma; \Delta; \Gamma \vdash^\psi \varphi : \forall \bar{\iota}.F\,\bar{\iota} \to \psi(\bar{\iota})}{\Sigma; \Delta; \Gamma \vdash \varphi^\psi : F\,\overline{\tau} \to \psi(\overline{\tau})}$$

$$(\text{MIt})\frac{\Sigma; \Delta, X^\kappa; \Gamma, x : \forall \overline{\iota'}.X\overline{\iota'} \to \psi(\overline{\iota'}) \vdash^\psi \varphi : \forall \bar{\iota}.F\,X\,\bar{\iota} \to \psi(\bar{\iota})}{\Sigma; \Delta; \Gamma \vdash \mathsf{MIt}\ x.\varphi^\psi : \mu^\kappa\,F\,\overline{\tau} \to \psi(\overline{\tau})}\left(X \notin \mathrm{FV}(\Gamma)\right)$$

$$(\text{In})\frac{}{\Sigma; \Delta; \Gamma \vdash \mathsf{In}^\kappa : F(\mu^\kappa F)\,\overline{\tau} \to \mu^\kappa F\,\overline{\tau}}$$

$$\boxed{\Sigma; \Delta; \Gamma \vdash^\psi \varphi : \sigma} \quad \frac{\Sigma|_T = \overline{C_k : \sigma_k}^{\,k=1..n} \quad \overline{\Sigma; \Delta \vdash \overline{A} \to T\,\overline{\tau}\,\overline{\tau}_k \prec \sigma_k\ \Sigma; \Delta; \Gamma, \overline{x : A} \vdash t : \psi(\overline{\tau}_k)}^{\,k=1..n}}{\Sigma; \Delta; \Gamma \vdash^\psi \overline{C_k\ \overline{x} \to t}^{\,k=1..n} : \forall \bar{\iota}.T\,\overline{\tau}\,\bar{\iota} \to \psi(\bar{\iota})}$$

Extending the Global Context: $\boxed{\Sigma \vdash D \Rrightarrow \Sigma'}$

$$(\Sigma, T)\frac{\overline{\Sigma, T : \kappa;\ \overline{\iota^K} \vdash \overline{A} \to T\,\overline{\tau} : *}}{\Sigma \vdash \textbf{data}\ T : \kappa\ \textbf{where}\ \overline{C : \overline{A} \to T\,\overline{\tau}} \ \Rrightarrow\ \Sigma, T : \kappa, \overline{C : \forall \bar{\iota}.\overline{A} \to T\,\overline{\tau}}}$$

$$(\Sigma, \text{`}x)\frac{\Sigma; \overline{\iota^K}; \cdot \vdash t : A}{\Sigma \vdash \text{`}x = t \Rrightarrow \Sigma, \text{`}x : \forall \bar{\iota}.A}\left(\bar{\iota} \cap \mathrm{FV}(t) = \emptyset\right)$$

Fig. 4. Typing rules of Nax

Type constructors and their kinding rules The syntax for type constructors of Nax is similar to F_i, but different from F_i in two aspects.

Firstly, polymorphic types are separate out as type schemes (σ) in Nax since the type system of Nax is in flavour of Hindley-Milner to support type inference (or, reconstruction).

Secondly, there are no type level abstractions and index abstractions in Nax. Instead of defining type constructors expecting type arguments by abstraction and index abstraction at type level, Nax supports datatype declarations (**data** $T : \kappa$ **where** $\ldots$) and recursive type operators (μ^κ) as language constructs.

Intuitively, the kinding rule for the recursive type operator should be $\Sigma \vdash \mu^\kappa : (\kappa \to \kappa) \to \kappa$. However, we restrict the recursive type operator (μ^κ) only to be applied to datatypes ($T\,\overline{\tau}$). This restriction is evident in both the type constructor syntax in Fig. 3 and the kinding rule (μ) in Fig. 4. What this restriction really excludes are nested applications of recursive type operators. For instance, $\mu^\kappa(\mu^{\kappa\to\kappa}F)$ where $F : (\kappa \to \kappa) \to \kappa \to \kappa$ is not allowed although it would be well-kinded under the less restrictive kinding rule ($\Sigma \vdash \mu^\kappa : (\kappa \to \kappa) \to \kappa$). Motivation behind this restriction is type inference. In order to infer a type for a Mendler style iterator, we need to restrict the form of its body since the body must be polymorphic over the indices (see (MIt) rule). In general, we do not want polymorphic types to be first class since we want type inference. One simple design choice is to allow case branches (φ) to have polymorphic types, or type schemes, and annotate case branches with index transformers (φ^ψ). For the exact same reason (i.e., type inference), we restrict the body of the Mendler style iterators be case terms (i.e., ($\mathsf{MIt}\ x.\varphi^\psi$) instead of ($\mathsf{MIt}\ x.t$)).

Terms and their typing rules The term syntax of Nax has six additional term constructs than F_i: data constructors (C), top level variables ($`x$), polymorphic let bindings (**let** $x = s$ **in** t), eliminators for datatypes (φ^ψ), Mendler style iterators ($\mathsf{MIt}\ x.\varphi^\psi$), and constructors for recursive types (In^κ). Typing rules for them are provided in Fig. 4.

The typing rules ($: C$) and ($: `$) are for data constructors ($C$) and top level variables ($`x$) bound in the top level context (Σ). Data constructors (C) are introduced from datatype declarations (**data** $T : \kappa$ **where** $\ldots$) by the rule (Σ, T), and top level variables ($`x$) are introduced from top level definitions ($`x = t$) by the rule ($\Sigma, `x$). The typing rules ($: C$) and ($: `$) behave similar to the rule ($:$) for the variables and the rule ($: i$) for index variables. All these four rules ($:$), ($: i$), ($: C$), and ($: `$) for identifiers look up a certain context (one of the three zones Σ, Δ, and Γ). Since the Nax type system is in flavour of Hindley-Milner, identifiers are bound to type schemes (σ) and the typing rules for the identifiers instantiate type schemes to types (A). Note, a type instantiation ($\Sigma; \Delta \vdash A \prec \sigma$) is a judgement under the top level context (Σ) and the type level context (Δ), since the instantiated type needs to be well-kinded under Σ and Δ.

Polymorphic let bindings in Nax are just the usual polymorphic bindings of Hindley-Milner type system for generalizing types of local definitions into type

schemes. In Nax, we generalize over term indices as well as types. The typing rule for let bindings is the (let) rule.

Eliminators for datatypes (φ^{ψ}), or case-terms, are case branches (φ) annotated by index transformers (ψ). For non-indexed types, case-terms are the usual single level pattern matching expressions in functional languages. For example, a Nax case-term applied to non-indexed typed term ($\varphi^{\cdot}\ s$) corresponds to a Haskell case-expression over that term (**case** s **of** $\{\varphi\}$). Note, we give trivial index transformer annotation (i.e., $\psi = \cdot$) for non-indexed types (e.g., booleans, natural numbers) since there are no indices to worry about. For indexed types, the indexed transformer annotations proivide useful information for type reconstruction. For example, consider the following datatype declaration:

$$\textbf{data}\ \texttt{Judgement} : \texttt{Bool} \to * \ \textbf{where}\ \ \begin{array}{r}\texttt{TJ} : \texttt{Formula} \to \texttt{Judgement}\ \{\texttt{True}\};\\ \texttt{FJ} : \texttt{Formula} \to \texttt{Judgement}\ \{\texttt{False}\}\end{array}$$

The datatype `Judgement` is index by boolean terms (e.g., `True` and `False` of type `Bool`). The data constructor `TJ` contains a formula expected to be true and the data constructor `FJ` contains a formula expected to be false. We can define a function, which produces an inverted judgement by negating the formula contained in a given judgement, as follows[6]:

$$\begin{pmatrix}\texttt{TJ}\ x \to \texttt{FJ}(\mathit{neg}\ \ x);\\ \texttt{FJ}\ x \to \texttt{TJ}(\mathit{neg}\ \ x)\end{pmatrix}^{i\,.\,\texttt{Judgement}\ \{`\mathit{not}\ i\}}$$

where *neg* is a function that produces negated formula and '*not* is a top level function that negates booleans.

Note that the index transformer ($i\,.\,\texttt{Judgement}\ \{`\mathit{not}\ i\}$) captures the idea that the resulting inverted judgement has opposite expectations from the given judgement. The types of such case-terms involving indexed types can also be inferred when we annotate the case-terms with appropriate index transformers. Reduction rules for case-terms (Fig. 3) are standard.

Mendler style iterators ($\mathsf{MIt}\ x.\varphi^{\psi}$) are eliminators for recursive types. A case-term expects a datatype argument (of type $T\,\overline{\tau}$). A Mendler style iterator expects a recursive type argument (of type $\mu^{\kappa}(T\,\overline{\tau})$). Intuitively, Mendler style iterators open up the recursive type ($\mu^{\kappa}(T\,\overline{\tau})$) and case branch over its base datatype structure (of type $T\,\overline{\tau}$). This intuition is captured by the reduction rule for Mendler style iterators (Fig. 3): $\mathsf{MIt}\ x.\varphi^{\psi}\ (\mathsf{In}^{\kappa} t) \rightsquigarrow [\mathsf{MIt}\ x.\varphi^{\psi}/x]\varphi^{\psi}\ t$. Note that a Mendler style iterator ($\mathsf{MIt}\ x.\varphi^{\psi}$) applied to a term of recursive type ($\mathsf{In}^{\kappa} t$) constructed by the In^{κ} constructor reduces to a case-term ($[\mathsf{MIt}\ x.\varphi^{\psi}/x]\varphi^{\psi}$) applied to the base structure (t) contained in the In^{κ} constructor. The variable (x) bound by MIt is a label for the recursive call. Note that the case-term ($[\mathsf{MIt}\ x.\varphi^{\psi}/x]\varphi^{\psi}$) appearing in the reduction rule substitutes x with the Mendler style iterator itself. However, unlike the fixpoint operator for unrestricted general recursion, Mendler style iterators are guaranteed to normalize because of their carefully designed typing rule (MIt) due to Mendler.

The constructors for recursive types (In^{κ}) are standard (see rule (In) in Fig. 4). The kind annotation κ on the In^{κ} constructor aids kind inference. If

[6] case branches are laid out in multiple lines for better readability

we were to simulate the recursive type operator μ^κ and its constructor In^κ in a functional language like Haskell (with GADT and kind annotation extensions), we would simulate them by the following recursive datatype:

$$\textbf{data}\ \mu^\kappa : (\kappa \to \kappa) \to \kappa\ \textbf{where}\ \mathsf{In}^\kappa : X(\mu^\kappa X)\bar{\iota} \to \mu^\kappa X\,\bar{\iota}$$

However, such a simulation of μ^κ by a recursive datatype cannot guarantee normalization of the language, since unlimited elimination of In^κ via case branches is already powerful enough to encode non-terminating computation even without using any recursion at term level. Thus, Nax provides μ^κ and In^κ as primitive language constructs, and only allow elimination of In^κ via Mendler style iteration.

Nax programs and their typing rules A Nax program $(\overline{D}; t)$ is a list of declarations $(\overline{D})$ followed by a term (t). A declaration can be either a datatype declaration $(\textbf{data}\ T : \kappa\ \textbf{where}\ \ldots)$ or a top level definition $(`x = t)$. The list of declarations $(\overline{D})$ are processed by the rules $(\Sigma, T)$ and $(\Sigma, `x)$ before type checking the term (t). The kinding and typing information from the datatype declarations and the top level definitions preceding the term are captured into the top level context (Σ) according to the rules (Σ, T) and $(\Sigma, `x)$. The top level context extended by these rules are used for type checking the term following the list of declarations. Therefore, the sorting, kinding, and typing rules of Nax (Fig. 4) involves Σ in addtion to Δ and Γ, while the corresponding rule of F_i (Fig. **??**) involves Δ and Γ only.

Reduction rules Reduction rules are defined in Fig. 3. First five rules are the redex rules that makes an actual reduction step on redexes. A redex is be one of the following: a lambda term applied to an argument, a let binding, a case term applied to a constructor term, a Mendler style iterator applied to an In^κ-constructed term, and a top level variable.

Note, the reduction rule for $`x$ mentions $\overline{D}$). Although we illustrate the reduction as a relation on terms $(t \rightsquigarrow t')$, we implicitly assume that there exists some fixed list of declarations $(\overline{D})$ for the reduction relation $(\rightsquigarrow)$. In order to make a reduction step for top level variables, we need to know the top level definition for $`x$, which should be contained in $\overline{D}$. Since the list of declarations are given by the input program $(\overline{D}; t)$ to type check, it is non-ambiguous which $\overline{D}$ to use for reducing $`x$. In case when it is ambiguous, we could use a notation like $t \overset{\overline{D}}{\rightsquigarrow} t'$ to make it more precise.

The other rules, following the top level variable reduction rule $(`x \rightsquigarrow t)$, are context rules to make a reduction step for the terms whose redexes appear inside their subterms.

3.2 Syntax-directed type system and type inference

The kinding and typing rules of Nax illustrated in Fig. 4 is not syntax directed since the conversion rules $(Conv)$ and $(=)$ are not syntax directed. These con-

version rules can apply to anywhere regardless of the syntactic category of terms and types.

We can easily adapt the system to be syntax directed by embedding the conversion rules into application-like rules (e.g., $(@i)$, $(\to E)$). Among the kinding rules, the only place where conversion is truly necessary is in the index application rule $(@i)$. We can define the syntax directed application rule $(@i)_s$ as follows:

$$(@i)_s \frac{\Sigma; \Delta \vdash_s F : A \to \kappa \qquad \Sigma; \Delta; \cdot \vdash_s s : A' \qquad \Sigma; \cdot \vdash_s A = A' : *}{\Sigma; \Delta \vdash_s F\,\{s\} : \kappa}$$

Among the typing rules, we need to embed conversion into the $(\to E)$ rule and probably the rules (let) and (MIt), and the branch checking rule as well. Once we finish describing the syntax directed type system, we should prove that it is equivalent to the typing rules in Fig. 4. We are still working on describing the type inference algorithm. Then, we would need to prove the correctness of the type inference algorithm with respect to the syntax directec typing rules.

4 Embedding Nax into strongly normalizing calculus

Our approach to formalizing the Nax as a logical language, is to embed each logical feature of Nax into a lower level language, System F_i, which we have proven to be strongly normalizing and logically consistent[2]. We have designed System F_i, which is an extension of F_ω with erasable term indices, and proved its properties.[7] Our approach of distinguishing type and term indices is unique, and requires the extension of some previous work on normalizing calculi.

5 Implementation

We used Haskell and its libraries (e.g., unbound [10]) to implement a prototype of Nax as an interpreter.

6 Future Work

Nax is one thread of research in the Trellys project, a collaborative initiative to design a dependently-typed programming language with simple support for general recursion and other convenient but logically unsound features, yet still maintain a logically sound core. Here is a partial list of ongoing and future work in the Nax thread.

– *Embedding all of Nax into a strongly normalizing calculus*
We can embed datatypes of Nax and the MIt and Msfit combinator families

[7] We submitted a paper on System F_i on POPL recently. Along with this submission, the draft of that paper on System F_i is included as a supplimentary material.

into System F_i. But, other combinator families, such as MPr, are only known to be embeddable into a different calculus, System Fix_ω [1], which does not support term indices. We believe we can similarly extend Fix_ω with erasable term indices – we call this calculus Fix_i, which can then embed MPr over datatypes with term indices. Properties of Fix_i needs to be checked but we strongly believe that the desired properties, i.e., strong normalization and logical consistency, will hold in Fix_i as well as in F_i.

– *Including both a programatic and a logical fragment*
In the future we want Nax programs to include both a logical fragment and a non-logical (or programatic) fragment, and we want the type system to separate the two. We believe that we can extend the proof principles outlined in the paper *Step-Indexed Normalization for a Language with General Recursion* [4].

– *Large eliminations*
The current prototype of Nax only supports a limited form of large elimination (i.e. mapping indices from argument types to result types) due to the limited syntax of the index transformer. We hope to enrich the index transformer syntax to support more expressive large eliminations (e.g., if-then-else, or more generally, case expressions in index transformers) and still maintain our design goals of having both type inference and a logically consistent type system. Again, we will make sure that such new features are safe by embedding the new feature into the calculus Nax is based on, such as System F_i.

Bibliography

[1] Abel, A., Matthes, R.: Fixed points of type constructors and primitive recursion. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL. Lecture Notes in Computer Science, vol. 3210, pp. 190–204. Springer (2004)

[2] Ahn, K.Y., Sheard, T., Fiore, M., Pitts, A.M.: System F_i: a higher-order polymorphic λ-calculus with erasable term indices (July 2012), submitted to POPL 2013

[3] Bird, R.S., Meertens, L.G.L.T.: Nested datatypes. In: Proceedings of the Mathematics of Program Construction. pp. 52–67. MPC '98, Springer-Verlag, London, UK, UK (1998)

[4] Casinghino, C., Sjöberg, V., Weirich, S.: Step-indexed normalization for a language with general recursion. In: MSFP '12 (2012)

[5] Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. pp. 90–104. Haskell '02, ACM, New York, NY, USA (2002)

[6] Cheney, J., Hinze, R.: First-class phantom types. Tech. rep., Cornell University (2003)

[7] Fegaras, L., Sheard, T.: Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 284–294. POPL '96, ACM (1996)

[8] Henglein, F.: Type inference with polymorphic recursion. ACM Trans. Program. Lang. Syst. 15(2), 253–289 (Apr 1993)

[9] Lin, C.K.: Practical Type Inference for the GADT Type System. Ph.D. thesis, Portland State University (2010)

[10] Weirich, S., Yorgey, B.A., Sheard, T.: Binders unbound. In: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming. pp. 333–345. ICFP '11, ACM, New York, NY, USA (2011)

[11] Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 224–235. POPL '03, ACM, New York, NY, USA (2003)

[12] Zenger, C.: Indexed types. Theoretical Computer Science 187, 147–165 (1997)

A Supplimentary material

Material following from next page is a supplimentary material on System F_i, which is a draft submitted to POPL '13.

System F_i

a higher-order polymorphic λ-calculus with erasable term indices

Ki Yung Ahn Tim Sheard
Portland State University
{kya,sheard}@cs.pdx.edu

Marcelo Fiore Andrew M. Pitts
University of Cambridge
{Marcelo.Fiore,Andrew.Pitts}@cl.cam.ac.uk

Abstract

The purpose of this paper is to introduce a foundational type system, System F_i, for the design of programming languages with first-class term-indexed datatypes – higher-order datatypes whose parameters range over data such as Natural Numbers or Lists.

To do this, we have devised a minimal extension of System F_ω that incorporates term indices. While term-indexed datatypes are expressible in rich type theories, like the Implicit Calculus of Constructions (ICC), these systems typically come coupled with orthogonal features such as large eliminations and full type dependency. We argue that there are important pedagogical benefits of isolating the minimal features to support term-indexing. We show that System F_i provides a theory for analysing programs with term-indexed types and also argue that it constitutes a basis for the design of logically-sound light-weight dependent programming languages.

In terms of expressivity, System F_i sits in between System F_ω (the prototypical logical calculus for functional programming) and ICC (a full-featured dependent type theory). Indeed, we relate System F_i to System F_ω and ICC as follows. We establish erasure properties of F_i-types that capture the idea that term indices are discardable in that they are irrelevant for computation. Index erasure projects typing in System F_i to typing in System F_ω; so System F_i inherits the strong-normalisation property from System F_ω. The logical consistency of System F_i is established by embedding it into a subset of ICC.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—data types and structures; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—lambda calculus and related systems

General Terms Languages, Theory

Keywords term-indexed data types, generalized algebraic data types, higher-order polymorphism, type-constructor polymorphism, higher-kinded types, impredicative encoding

1. Introduction

We wish to incorporate dependent types into ordinary programming languages. We are interested in two kinds of dependent types. Full dependency, where the type of a function can depend upon the value of its run-time parameters, and static dependency, where the type of a function can depend only upon static (or compile-time) parameters. Static dependency is sometimes referred to as indexed typing. The first is very expressive, while the second is often much easier to learn and use especially for those who are familiar to functional programming languages like Haskell or ML. Indexed types come in two flavors: type-indexed and term-indexed types. Type indexing includes parametric polymorphism, but it also includes more sophisticated typing as found in Generalized Algebraic Datatypes (GADTs). An example of type indexing using GADTs is a type representation:

```
data TypeRep t where
  Int :: TypeRep Int
  Bool:: TypeRep Bool
  Pair:: TypeRep a -> TypeRep b -> TypeRep(a,b)
```

Here, a value of type `(TypeRep t)` is isomorphic in "shape" with the type `t`. For example `(Pair Int Bool)` is isomorphic in shape with its type `(Int,Bool)`.

On the other hand, term-indexed types include indices that range over data structures, such as Natural Numbers (like `Z`, `(S Z)`) or Lists (like `Nil` or `(Cons Z Nil)`). The classic example of a term index is the second parameter to the length-indexed list type `Vec` (as in `(Vec Int (S Z))`). In languages such as Haskell, which support GADTs with type indexing, term-indices are not first-class; they are "faked" by reflecting data at the type level with uninhabited type constructors (see §6 for a very recent GHC extension, which enable term-indices be first class). For example,

```
data Succ n
data Zero

data Vector t n where
  Cons :: a -> Vector a n -> Vector a (Succ n)
  Nil  :: Vector a Zero
```

This comes with a number of problems. First, there is no way to say that types such as `(Succ Int)` are ill-formed, and second the costs associated with duplicating the constructor functions of data to be used as term-indices. Nevertheless, "faked" term-indexed GADTs have become extremely popular as a light-weight, type-based mechanism to raise the confidence of users that software systems maintain important properties.

A salient example is Guillemette's thesis [11] encoding the classic paper by Morrisett et al. [18] completely in Haskell. This impressive system embeds a multi-stage compiler, from System F all the way to typed assembly language using indexed datatypes (many of them "faked" term-indices) to show that every stage preserves type information. As such, it provides confidence but no guarantees. Indeed, since in Haskell the non-terminating computation can be assigned any type, it is in principle possible that the type-preservation property is a consequence of a non-terminating computation in the program code.

This drawback is absent in approaches based on strongly normalizing logical calculi; like, for instance, System F_ω, the higher-order polymorphic lambda calculus, which is rich enough to express a wide collection of data structures. Unfortunately, the *term-indexed datatypes* that are necessary to support Guillemette's system are not known to be expressible in System F_ω.

In his CompCert system, Leroy [13] showed that the much richer logical Calculus of Inductive Constructions (CIC), which constitutes the basis of the Coq proof assistant, is expressive enough to guarantee type preservation (and more) between compiler stages. This approach, however, comes at a cost. Programmers must learn to use both dependent types and a new programming paradigm, programming by code extraction.

Some natural questions thus arise: Is there an expressive system supporting term-indexed types, say, sitting somewhere in between System F_ω and fully dependent calculi? If only term-indexed types are needed to maintain properties of interest, is there a language one can use? Can one program, rather than extract code? The goal of this paper is to develop the theory necessary to begin answering these and related questions.

Our approach in this direction is to design a new foundational calculus, System F_i, for functional programming languages with term-indexed datatypes. In a nutshell, System F_i is obtained by minimally extending System F_ω with type-indexed kinds. Notably, this yields a logical calculus that is expressive enough to embed non-dependent *term-indexed datatypes* and their eliminators. Our contributions in this development are as follows.

- Identifying the features that are needed for a higher-order polymorphic λ-calculus to embed term-indexed datatypes (§2), in isolation from other features normally associated with such calculi (e.g., general recursion, large elimination, dependent types).
- The design of the calculus, System F_i (§3), and its use to study properties of languages with term-indexed datatypes, by embedding these into the calculus (§4). For instance, one can use System F_i to prove that the Mendler-style eliminators for GADTs of [3] are normalizing.
- Showing that System F_i enjoys a simple erasure property and inherits meta-theoretic results (strong normalization and logical consistency) from well-known calculi (System F_ω and ICC) that enclose System F_i (§5).

2. From System F_ω to System F_i, and back

It is well known that datatypes can be embedded into polymorphic lambda calculi by means of functional encodings (e.g., [1]), such as the Church and Boehm-Berarducci encodings.

In System F, for instance, one can embed *regular datatypes* [4], like homogeneous lists:

Haskell: `data List a = Cons a (List a) | Nil`
System F: $\texttt{List}\ A \triangleq \forall X.(A \to X \to X) \to X \to X$

In such regular datatypes, constructors have algebraic structure that directly translates into polymorphic operations on abstract types as encapsulated by universal quantification.

In the more expressive System F_ω, one can encode more general *type-indexed datatypes* that go beyond the algebraic class. For example, one can embed powerlists with heterogeneous elements in which an element of type `a` is followed by an element of the product type `(a,a)`:

Haskell: `data Powl a = PCons a (Powl(a,a)) | PNil`
System F_ω: $\texttt{Powl} \triangleq \lambda A^*.\forall X^{*\to *}.$
$(A \to X(A \times A) \to XA) \to XA \to XA$

Note the non-regular occurrence (`Powl(a,a)`) in the definition of (`Powl a`), and the use of universal quantification over higher-order kinds.

What about term-indexed datatypes? What extension to System F_ω is needed to embed these, as well as type-indexed ones? Our answer is System F_i.

In a functional language supporting term-indexed datatypes, we envisage that the classic example of homogeneous length-indexed lists would be defined along the following lines:

```
data Nat = S Nat | Z

data Vec (a:*) {i:Nat} where
  VCons : a -> Vec a {i} -> Vec a {S i}
  VNil  : Vec a {Z}
```

Here the type constructor `Vec` is defined to admit parameterisation by both type and term indices. For instance, the type (`Vec (List Nat) {S (S Z)}`) is that of two-dimensional vectors of lists of natural numbers. By design, our syntax directly reflects the different type and term indexing by indicating the latter in curly brackets. This feature has been directly transferred from System F_i, where it is used as a mechanism for guaranteeing the static nature of term indexing.

The encoding of the vector datatype in System F_i is as follows:

$$\texttt{Vec} \triangleq \lambda A^*.\lambda i^{\texttt{Nat}}.\forall X^{\texttt{Nat}\to *}.$$
$$(\forall j^{\texttt{Nat}}.A \to X\{j\} \to X\{\texttt{S}\ j\}) \to X\{\texttt{Z}\} \to X\{i\}$$

where `Nat`, `Z`, and `S` respectively encode the Natural Numbers, zero and successor. Without going into the details of the formalism, which are given in the next section, one sees that such a calculus incorporating term-indexing structure needs four additional constructs.

1. Type-indexed kinding $(A \to \kappa)$ (as in $(\texttt{Nat}\to *)$ in the example above) where the compile-time nature of term-indexing will be reflected in the enforcement that A be a closed type (rule (Ri) in Figure 1).
2. Term-index abstraction $\lambda i^A.F$ (as $\lambda i^{\texttt{Nat}}.\cdots$ in the example above) for constructing (or introducing) type-indexed kinds (rule (λi) in Figure 1).
3. Term-index application $F\{s\}$ (as $X\{\texttt{Z}\}$, $X\{j\}$, and $X\{\texttt{S}\ j\}$ in the example above) for destructing (or eliminating) type-indexed kinds, where the compile-time nature of indexing will be reflected in the enforcement that the index be statically typed (rule $(@i)$ in Figure 1) .
4. Term-index polymorphism $\forall i^A.B$ (as $\forall j^{\texttt{Nat}}.\cdots$ in the example above) where the compile-time nature of polymorphic term-indexing will be reflected in the enforcement that the variable i be static of closed type A (rule $(\forall Ii)$ in Figure 1).

As exemplified above, System F_i maintains a clear-cut separation between higher-order kinding and term indexing. This adds a level of abstraction to System F_ω and yields types that in addition to structural invariants also keep track of indexing invariants. Being static, all term-index information can be erased. This projects System F_i into System F_ω fixing the latter. For instance, the erasure of the F_i-type `Vec` is the F_ω-type `List`, the erasure of which (when regarded as an F_i-type that is) is in turn itself. Since, as already mentioned, typing in System F_i imposes structural and indexing constraints on terms one expects that the structural projection from System F_i to System F_ω provided by index erasure preserves typing. This is established in §5 and used to deduce the strong normalization of System F_i.

3. System F_i

System F_i is a higher-order polymorphic lambda calculus designed to extend System F_ω by the inclusion of term indices. The syntax and rules of System F_i are described in Figures 1 and 2. The extensions new to System F_i, which are not originally part of System F_ω, are highlighted by grey boxes. Eliding all the grey boxes from Figures 1 and 2, one obtains a version of System F_ω with Curry-style terms and the typing context separated into two parts (type-level context Δ and term-level context Γ).

In this section, we first discuss the rational for our design choices (§3.1) and then introduce the new constructs of System F_i (§3.2).

3.1 Design of System F_i

Terms in F_i are Curry style. That is, term level abstractions are unannotated ($\lambda x.t$), and type generalizations ($\forall I$) and type instantiations ($\forall E$) are implicit at term level. A Curry-style calculus generally has an advantage over its Church-style counterpart when reasoning about properties of reduction. For instance, the Church-Rosser property naturally holds for β-, η-, and $\beta\eta$-reduction in the Curry style, but may not hold in the Church style. This is due to the presence of annotations in abstractions [16].[1]

Type constructors, on the other hand, remain Church style in F_i. That is, type level abstractions are annotated by kinds ($\lambda X^\kappa.F$). Choosing type constructors to be Church style makes the kind of a type constructor visually explicit. The choice of style for type constructors is not as crucial as the choice of style for terms, since the syntax and kinding rules at type level are essentially a simply typed lambda calculus. Annotating the type level abstractions with kinds makes kinds explicit in the type syntax. Since F_i is essentially an extension of F_ω with a new formation rule for kinds, making kinds explicit is a pedagogical tool to emphasize the consequences of this new formation rule. As a notational convention, we write A and B, instead of F and G, where A and B to are expected to be types (i.e., nullary type constructors) of kind $*$.

In a language with term indices, terms appear in types (e.g., the length index $(n + m)$ in the type *Vec Nat* $\{n + m\}$). Such terms contain variables. The binding sites of these variables matter. In F_i, we expect such variables to be statically bound. Dynamically bound index variables would require a dependently typed calculus, such as the calculus of constructions. To reflect this design choice, typing contexts are separated into type level contexts (Δ) and term level contexts (Γ). Type level (static) variables (X , i) are bound in Δ and term (dynamic) variables (x) are bound in Γ. Type level variables are either type constructor variables (X) or term variables to be used as indices (i). As a notational convention, we write i, instead of x, when term variables are to be used as indices (i.e., introduced by either index abstraction or index polymorphism).

In contrast to our design choice, System F_ω is most often formalized using a single context, which binds both type variables (X) and term variables (x). In such a formalization, the free type variables in the typing of the term variable must be bound earlier in the context. For example, if X and Y appear free in the type of f, they must appear earlier in the single context (Γ) as below:

$$\Gamma = \ldots, X^*, \ldots, Y^*, \ldots, (f : \forall Z^*.X \to Y \to Z), \ldots$$

In such a formalization, the side condition ($X \notin \Gamma$) in the ($\forall I$) rule of Figure 1 is not necessary, since such a condition is already a part of the well-formedness condition for the context (i.e., Γ, X^κ is well-formed when $X \notin \mathrm{FV}(\Gamma)$). Thus, for F_ω, it is only a matter of taste whether to formalize the system using a single context or two contexts, since they are equivalent formalizations with comparable complexity.

However, in F_i, we separate the context into two parts to distinguish term variables used in types (which we call index variables, or indices, and are bound as Δ, i^A) from the ordinary use of term variables (which are bound as $\Gamma, x : A$). The expectation is that indices should have no effect on reduction at the term level. Although it is imaginable to formalize F_i with a single typing context and distinguish index variables from ordinary term variables using more general concepts (e.g., capability, modality), we think that splitting the typing context into two parts is the simplest solution.

3.2 System F_i compared to System F_ω

We assume readers to be familiar with System F_ω and focus on describing the new constructs of F_i. These appear in grey boxes.

Kinds. The key extension to F_ω is the addition of term-indexed arrow kinds of the form $A \to \kappa$. This allows type constructors to have terms as indices. The rest of the development of F_i flows naturally from this single extension.

Sorting. The formation of indexed arrow kinds is governed by the sorting rule (Ri). The rule (Ri) specifies that an indexed arrow kind $A \to \kappa$ is well-sorted when A has kind $*$ under the empty type level context $(\cdot)$ and κ is well-sorted.

Requiring the use of the empty context avoids dependent kinds (i.e., kinds depending on type level or value level bindings). The type A appearing in the index arrow kind $A \to \kappa$ must be well-kinded under the empty type level context $(\cdot)$. That is, A should to be a closed type of kind $*$, which does not contain any free type variables or index variables. For example, $(\mathit{List}\, X \to *)$ is not a well-sorted kind, while $((\forall X^*.\, \mathit{List}\, X) \to *)$ is a well-sorted kind.

Typing contexts. Typing contexts are split into two parts. Type level contexts (Δ) for type level (static) bindings, and term level contexts (Γ) for term level (dynamic) bindings. A new form of index variable binding (i^A) can appear in type level contexts in addition to the traditional type variable bindings (X^κ). There is only one form of term level binding ($x : A$) that appears in term level contexts.

Well formed typing contexts. A type level context Δ is well-formed if (1) it is either empty, or (2) extended by a type variable binding X^κ whose kind κ is well-sorted under Δ, or (3) extended by an index binding i^A whose type A is well-kinded under the empty type level context at kind $*$. This restriction is similar to the one that occurs in the sorting rule (Ri) for term-indexed arrow kinds (see the paragraph ***Sorting***). The consequence of this is that, in typing contexts and in sorts, A must be a closed type (not a type constructor!) without free variables.

A term level context Γ is well-formed under a type level context Δ when it is either empty or extended by a term variable binding $x : A$ whose type A is well-kinded under Δ.

Type constructors and their kinding rules. We extend the type constructor syntax by three constructs, and extend the kinding rules accordingly for these new constructs.

$\lambda i^A.F$ is the type level abstraction over an index (or, index abstraction). Index abstractions introduce indexed arrow kinds by the kinding rule (λi). Note, the use of the new form of context extension, i^A, in the kinding rule (λi).

$F\,\{s\}$ is the type level index application. In contrast to the ordinary type level application ($F\,G$) where the argument (G) is a type constructor, the argument of an index application ($F\,\{s\}$)

[1] The Church-Rosser property, in its strictest sense (i.e., α-equivalence over terms), generally does not hold in Church-style calculi , but may hold under certain approximations, such as modulo ignoring the annotations in abstractions.

Syntax:

Sort	$\Box$
Term Variables	x, i
Type Constructor Variables	X
Kinds	$\kappa ::= * \mid \kappa \to \kappa \mid A \to \kappa$
Type Constructors	$A, B, F, G ::= X \mid A \to B \mid \lambda X^\kappa.F \mid F\,G \mid \forall X^\kappa.B \mid \lambda i^A.F \mid F\,\{s\} \mid \forall i^A.B$
Terms	$r, s, t ::= x \mid \lambda x.t \mid r\ s$
Typing Contexts	$\Delta ::= \cdot \mid \Delta, X^\kappa \mid \Delta, i^A$
	$\Gamma ::= \cdot \mid \Gamma, x : A$

Well-formed typing contexts:

$\boxed{\vdash \Delta}$ $\quad \dfrac{}{\vdash \cdot} \quad \dfrac{\vdash \Delta \quad \vdash \kappa : \Box}{\vdash \Delta, X^\kappa}\,(X \notin \mathsf{dom}(\Delta)) \quad \dfrac{\vdash \Delta \quad \cdot \vdash A : *}{\vdash \Delta, i^A}\,(i \notin \mathsf{dom}(\Delta))$

$\boxed{\Delta \vdash \Gamma}$ $\quad \dfrac{\vdash \Delta}{\Delta \vdash \cdot} \quad \dfrac{\Delta \vdash \Gamma \quad \Delta \vdash A : *}{\Delta \vdash \Gamma, x : A}\,(x \notin \mathsf{dom}(\Gamma))$

Sorting: $\boxed{\vdash \kappa : \Box}$ $\quad (A)\dfrac{}{\vdash * : \Box} \quad (R)\dfrac{\vdash \kappa : \Box \quad \vdash \kappa' : \Box}{\vdash \kappa \to \kappa' : \Box} \quad (Ri)\dfrac{\cdot \vdash A : * \quad \vdash \kappa : \Box}{\vdash A \to \kappa : \Box}$

Kinding: $\boxed{\Delta \vdash F : \kappa}$ $\quad (Var)\dfrac{X^\kappa \in \Delta \quad \vdash \Delta}{\Delta \vdash X : \kappa} \quad (\to)\dfrac{\Delta \vdash A : * \quad \Delta \vdash B : *}{\Delta \vdash A \to B : *}$

$(\lambda)\dfrac{\vdash \kappa : \Box \quad \Delta, X^\kappa \vdash F : \kappa'}{\Delta \vdash \lambda X^\kappa.F : \kappa \to \kappa'} \quad (@)\dfrac{\Delta \vdash F : \kappa \to \kappa' \quad \Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'} \quad (\forall)\dfrac{\vdash \kappa : \Box \quad \Delta, X^\kappa \vdash B : *}{\Delta \vdash \forall X^\kappa.B : *}$

$(\lambda i)\dfrac{\cdot \vdash A : * \quad \Delta, i^A \vdash F : \kappa}{\Delta \vdash \lambda i^A.F : A \to \kappa} \quad (@i)\dfrac{\Delta \vdash F : A \to \kappa \quad \Delta; \cdot \vdash s : A}{\Delta \vdash F\,\{s\} : \kappa} \quad (\forall i)\dfrac{\cdot \vdash A : * \quad \Delta, i^A \vdash B : *}{\Delta \vdash \forall i^A.B : *}$

$(Conv)\dfrac{\Delta \vdash A : \kappa \quad \Delta \vdash \kappa = \kappa' : \Box}{\Delta \vdash A : \kappa'}$

Typing: $\boxed{\Delta; \Gamma \vdash t : A}$ $\quad (:)\dfrac{(x : A) \in \Gamma \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash x : A} \quad (:i)\dfrac{i^A \in \Delta \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i : A}$

$(\to I)\dfrac{\Delta \vdash A : * \quad \Delta; \Gamma, x : A \vdash t : B}{\Delta; \Gamma \vdash \lambda x.t : A \to B} \quad (\to E)\dfrac{\Delta; \Gamma \vdash r : A \to B \quad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash r\ s : B}$

$(\forall I)\dfrac{\vdash \kappa : \Box \quad \Delta, X^\kappa; \Gamma \vdash t : B}{\Delta; \Gamma \vdash t : \forall X^\kappa.B}\,(X \notin \mathrm{FV}(\Gamma)) \quad (\forall E)\dfrac{\Delta; \Gamma \vdash t : \forall X^\kappa.B \quad \Delta \vdash G : \kappa}{\Delta; \Gamma \vdash t : B[G/X]}$

$(\forall Ii)\dfrac{\cdot \vdash A : * \quad \Delta, i^A; \Gamma \vdash t : B}{\Delta; \Gamma \vdash t : \forall i^A.B}\begin{pmatrix} i \notin \mathrm{FV}(t), \\ i \notin \mathrm{FV}(\Gamma) \end{pmatrix} \quad (\forall Ei)\dfrac{\Delta; \Gamma \vdash t : \forall i^A.B \quad \Delta; \cdot \vdash s : A}{\Delta; \Gamma \vdash t : B[s/i]}$

$(=)\dfrac{\Delta; \Gamma \vdash t : A \quad \Delta \vdash A = B : *}{\Delta; \Gamma \vdash t : B}$

Reduction: $\boxed{t \rightsquigarrow t'}$ $\quad \dfrac{}{(\lambda x.t)\ s \rightsquigarrow t[s/x]} \quad \dfrac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'} \quad \dfrac{r \rightsquigarrow r'}{r\ s \rightsquigarrow r'\ s} \quad \dfrac{s \rightsquigarrow s'}{r\ s \rightsquigarrow r\ s'}$

Figure 1. Syntax, Typing rules, and Reduction rules of F_i

Kind equality: $\boxed{\vdash \kappa = \kappa' : \square}$

$$\frac{}{\vdash * = * : \square} \qquad \frac{\vdash \kappa_1 = \kappa_1' : \square \quad \vdash \kappa_2 = \kappa_2' : \square}{\vdash \kappa_1 \to \kappa_2 = \kappa_1' \to \kappa_2' : \square} \qquad \frac{\cdot \vdash A = A' : * \quad \vdash \kappa = \kappa' : \square}{\vdash A \to \kappa = A' \to \kappa' : \square}$$

$$\frac{\vdash \kappa = \kappa' : \square}{\vdash \kappa' = \kappa : \square} \qquad \frac{\vdash \kappa = \kappa' : \square \quad \vdash \kappa' = \kappa'' : \square}{\vdash \kappa = \kappa'' : \square}$$

Type constructor equality: $\boxed{\Delta \vdash F = F' : \kappa}$

$$\frac{\Delta, X^\kappa \vdash F : \kappa' \quad \Delta \vdash G : \kappa}{\Delta \vdash (\lambda X^\kappa.F)\, G = F[G/X] : \kappa'} \qquad \frac{\Delta, i^A \vdash F : \kappa \quad \Delta; \cdot \vdash s : A}{\Delta \vdash (\lambda i^A.F)\, \{s\} = F[s/i] : \kappa}$$

$$\frac{\Delta \vdash X : \kappa}{\Delta \vdash X = X : \kappa} \qquad \frac{\Delta \vdash A = A' : * \quad \Delta \vdash B = B' : *}{\Delta \vdash A \to B = A' \to B' : *}$$

$$\frac{\vdash \kappa : \square \quad \Delta, X^\kappa \vdash F = F' : \kappa'}{\Delta \vdash \lambda X^\kappa.F = \lambda X^\kappa.F' : \kappa \to \kappa'} \qquad \frac{\Delta \vdash F = F' : \kappa \to \kappa' \quad \Delta \vdash G = G' : \kappa}{\Delta \vdash F\, G = F'\, G' : \kappa'} \qquad \frac{\vdash \kappa : \square \quad \Delta, X^\kappa \vdash B = B' : *}{\Delta \vdash \forall X^\kappa.B = \forall X^\kappa.B' : *}$$

$$\frac{\cdot \vdash A : * \quad \Delta, i^A \vdash F = F' : \kappa}{\Delta \vdash \lambda i^A.F = \lambda i^A.F' : A \to \kappa} \qquad \frac{\Delta \vdash F = F' : A \to \kappa \quad \Delta; \cdot \vdash s = s' : A}{\Delta \vdash F\, \{s\} = F'\, \{s'\} : \kappa} \qquad \frac{\cdot \vdash A : * \quad \Delta, i^A \vdash B = B' : *}{\Delta \vdash \forall i^A.B = \forall i^A.B' : *}$$

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F' = F : \kappa} \qquad \frac{\Delta \vdash F = F' : \kappa \quad \Delta \vdash F' = F'' : \kappa}{\Delta \vdash F = F'' : \kappa}$$

Term equality: $\boxed{\Delta; \Gamma \vdash t = t' : A}$

$$\frac{\Delta; \Gamma, x : A \vdash t : B \quad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash (\lambda x.t)\, s = t[s/x] : B} \qquad \frac{\Delta; \Gamma \vdash x : A}{\Delta; \Gamma \vdash x = x : A}$$

$$\frac{\Delta \vdash A : * \quad \Delta; \Gamma, x : A \vdash t = t' : B}{\Delta; \Gamma \vdash \lambda x.t = \lambda x.t' : B} \qquad \frac{\Delta; \Gamma \vdash r = r' : A \to B \quad \Delta; \Gamma \vdash s = s' : A}{\Delta; \Gamma \vdash r\, s = r'\, s' : B}$$

$$\frac{\vdash \kappa : \square \quad \Delta, X^\kappa; \Gamma \vdash t = t' : B}{\Delta; \Gamma \vdash t = t' : \forall X^\kappa.B}\,(X \notin \mathrm{FV}(\Gamma)) \qquad \frac{\Delta; \Gamma \vdash t = t' : \forall X^\kappa.B \quad \Delta \vdash G : \kappa}{\Delta; \Gamma \vdash t = t' : B[G/X]}$$

$$\frac{\cdot \vdash A : * \quad \Delta, i^A; \Gamma \vdash t = t' : B}{\Delta; \Gamma \vdash t = t' : \forall i^A.B}\begin{pmatrix} i \notin \mathrm{FV}(t), \\ i \notin \mathrm{FV}(t'), \\ i \notin \mathrm{FV}(\Gamma) \end{pmatrix} \qquad \frac{\Delta; \Gamma \vdash t = t' : \forall i^A.B \quad \Delta; \cdot \vdash s : A}{\Delta; \Gamma \vdash t = t' : B[s/i]}$$

$$\frac{\Delta; \Gamma \vdash t = t' : A}{\Delta; \Gamma \vdash t' = t : A} \qquad \frac{\Delta; \Gamma \vdash t = t' : A \quad \Delta; \Gamma \vdash t' = t'' : A}{\Delta; \Gamma \vdash t = t'' : A}$$

Figure 2. Equality rules of F_i

is a term (s). We use the curly bracket notation around an index argument in a type to emphasize the transition from ordinary type to term, and to emphasize that s is an index term, which is erasable. Index applications eliminate indexed arrow kinds by the kinding rule $(@i)$. Note, we type check the index term (s) under the current type level context paired with the empty term level context $(\Delta; \cdot)$ since we do not want the index term (s) to depend on any term level bindings. Allowing such a dependency would admit true dependent types.

$\forall i^A.B$ is an index polymorphic type. The formation of indexed polymorphic types is governed by the kinding rule $\forall i$, which is very similar to the formation rule ($\forall$) for ordinary polymorphic types.

In addition to the rules (λi), ($@i$), and ($\forall i$), we need a conversion rule $(Conv)$ at kind level. This is because the new extension to the kind syntax $A \to \kappa$ involves types. Since kind syntax involves types, we need more than simple structural equality over kinds. The new equality over kinds is the usual structural equality extended by type constructor equality when comparing indexed arrow kinds (see Figure 2).

Terms and their typing rules The term syntax is exactly the same as other Curry-style calclui. We write x for ordinary term variables introduced by term level abstractions ($\lambda x.t$). We write i for index variables introduced by index abstractions ($\lambda i^A.F$) and by index polymorphic types ($\forall i^A.B$). As discussed earlier, the distinction between x and i is for the convenience of readability.

Since F_i has index polymorphic types ($\forall i^A.B$), we need typing rules for index polymorphism: $(\forall Ii)$ for index generalization and $(\forall Ei)$ for index instantiation.

The index generalization rule ($\forall Ii$) is similar to the type generalization rule ($\forall I$), but generalizes over index variables (i) rather than type consturctor variables (X). The rule ($\forall Ii$) has two side conditions while the rule ($\forall I$) has only one side conditions. The additional side condition $i \notin \mathrm{FV}(t)$ in the ($\forall Ii$) rule prevents terms from accessing the type level index variables introduced by index polymorphism. Without this side condition, $\forall$-binder would no longer behave polymorphicaly, but instead would behave as a dependent function, which are usually denoted by the Π-binder in dependent type theories. The rule ($\forall I$) for ordinary type generalization does not need such additional side condition because type variables cannot appear in the syntax of terms. The side conditions on generalization rules for polymorphism is fairly standard in de-

$$(\lambda i)\dfrac{\cdot \vdash A : *\qquad (@i)\dfrac{\Delta, i^A \vdash F : A \to \kappa \qquad (:i)\dfrac{i^A \in \Delta, i^A \qquad \Delta \vdash \cdot}{\Delta, i^A; \cdot \vdash i : A}}{\Delta, i^A \vdash F\{i\} : \kappa}}{\Delta \vdash \lambda i^A.F\{i\} : A \to \kappa}$$

Figure 3. Kinding derivation for an index abstraction

pendently typed languages supporting distinctions between polymorphism (or, erasable arguments) and dependent functions (e.g., IPTS[17], ICC[16]).

The index instantiation rule ($\forall Ei$) is similar to the type instantiation rule ($\forall Ei$), except that we type check the index term s to be instantiated for i in the current type level context paired with the empty term level context $(\Delta; \cdot)$ rather than the current term level context. Since index terms are at type level, they should not depend on term level bindings.

In addition to the rules ($\forall Ii$) and ($\forall Ei$) for index polymorphism, we need an additional variable rule $(:i)$ to be able to access the index variables already in scope. Terms (s) used at type level in index applications ($F\{s\}$) should be able to access index variables already in scope. For example, $\lambda i^A.F\{i\}$ should be well-kinded under a context where F is well-kinded, justified by the derivation in Figure 3.

4. Embedding datatypes and their eliminators

System F_i can express a rich collection of datatypes. First, we illustrate embeddings for both non-recursive and recursive datatypes using Church encodings [6] to define data constructors (§4.1). Second, we illustrate a more involved embedding for recursive datatypes based on two-level types (§4.2). Lastly, we discuss an encoding of equality over term indices (§4.3).

4.1 Embedding datatypes using Church-encoded terms

Church [6] invented an embedding of the natural numbers into the untyped λ-calculus, which he used to argue that the λ-calculus was expressive enough for the foundation of logic and arithmetic. Church encoded the data constructors of natural numbers, successor and zero, as higher-order functions, $\texttt{succ} = \lambda x.\lambda x_s.\lambda x_z.x_s(x\,x_s x_z)$ and $\texttt{zero} = \lambda x_s.\lambda x_z.x_z$. The heart of the Church encoding is that a value is encoded as an elimination function. The bound variables x_s and x_z (of both `succ` and `zero`) stand for the operations needed to eliminate the successor case and the zero case respectively. The Church encodings of successor states: to eliminate $\texttt{succ}\,x$, "apply x_s to the elimination of the predecessor $(x\,x_s x_z)$"; and, to eliminate `zero`, just "return x_z". Since values *are* elimination functions, the eliminator can be defined as applying the value itself to the needed operations. One for each of the data constructors. For instance, we can define an eliminator for the natural numbers as $\texttt{elim}_{\texttt{Nat}} = \lambda x.\lambda x_s.\lambda x_z.x\,x_s x_z$. This is just an η-expansion of the identity function $\lambda x.x$. The Church encoded natural numbers are typable in a polymorphic λ-calculi, such as System F_ω, as follows:

$$\begin{array}{lll}
\texttt{Nat} & = & \forall X^*.(X \to X) \to X \to X \\
\texttt{S} & : \texttt{Nat} \to \texttt{Nat} \; = & \lambda x.\lambda x_s.\lambda x_z.x_s(x\,x_s x_z) \\
\texttt{Z} & : \texttt{Nat} \; = & \lambda x_s.\lambda x_z.x_z \\
\texttt{elim}_{\texttt{Nat}} & : \texttt{Nat} \to \forall X^*.(X \to X) \to X \to X & \\
 & = & \lambda x.\lambda x_s.\lambda x_z.x\,x_s x_z
\end{array}$$

In a Similar fashion, other datatypes are also embeddable into polymorphic λ-calculi. Embeddings of some well-known non-recursive datatypes are illustrated in Figure 4, and embeddings

$$\begin{array}{lll}
\texttt{Bool} & = & \forall X.X \to X \to X \\
\texttt{true} & : \texttt{Bool} \; = & \lambda x_1.\lambda x_2.x_1 \\
\texttt{false} & : \texttt{Bool} \; = & \lambda x_1.\lambda x_2.x_2 \\
\texttt{elim}_{\texttt{Bool}} & : \texttt{Bool} \to \forall X.X \to X \to X & \\
 & = & \lambda x.\lambda x_1.\lambda x_2.x\;x_1\;x_2 \qquad (\textbf{if}\ x\ \textbf{then}\ x_1\ \textbf{else}\ x_2) \\
\hline
A_1 \times A_2 & = & \forall X.(A_1 \to A_2 \to X) \to X \\
\texttt{pair} & : \forall A_1^*.\forall A_2^*.A_1 \times A_2 \; = & \lambda x_1.\lambda x_2.\lambda x'.x'\,x_1\,x_2 \\
\texttt{elim}_{(\times)} & : \forall A_1^*.\forall A_2^*.A_1 \times A_2 \to \forall X.(A_1 \to A_2 \to X) \to X & \\
 & = & \lambda x.\lambda x'.x\;x' \\
 & & \text{(by passing appropriate values to } x', \text{ we get} \\
 & & \texttt{fst} = \lambda x.x(\lambda x_1.\lambda x_2.x_1),\ \texttt{snd} = \lambda x.x(\lambda x_1.\lambda x_2.x_2)\ ) \\
\hline
A_1 + A_2 & = & \forall X^*.(A_1 \to X) \to (A_2 \to X) \to X \\
\texttt{inl} & : \forall A_1^*.\forall A_2^*.A_1 \to A_1 + A_2 \; = & \lambda x.\lambda x_1.\lambda x_2.x_1\,x \\
\texttt{inr} & : \forall A_1^*.\forall A_2^*.A_2 \to A_1 + A_2 \; = & \lambda x.\lambda x_1.\lambda x_2.x_2\,x \\
\texttt{elim}_{(+)} & : \forall A_1^*.\forall A_2^*.(A_1 + A_2) \to & \\
 & \quad \forall X^*.(A_1 \to X) \to (A_2 \to X) \to X & \\
 & = & \lambda x.\lambda x_1.\lambda x_2.x\;x_1\;x_2 \\
 & & (\textbf{case}\ x\ \textbf{of}\ \{\texttt{inl}\ x' \to x_1\ x'; \texttt{inr}\ x' \to x_2\ x'\})
\end{array}$$

Figure 4. Embedding non-recursive datatypes

of the list-like recursive datatypes, which we discussed earlier as motivating examples (§2), are illustrated in Figure 5. Note that the term encodings for the constructors and eliminators of the list-like datatypes in Figure 5 are exactly the same. For instance, the term encodings for `nil`, `pnil`, and `vnil` are all the same term: $\lambda x_s.\lambda x_z.x_z$. The nil and cons terms capture the linear nature of lists, so they are the same for all list like structures. But, the types differ, capturing different invariants about lists – shape of the elements (`Powl`), and length of the list (`Vec`).

4.2 Embedding recursive datatypes as two-level types

We can divide a recursive datatype definition into two parts – a recursive type operator and a base structure. The operator "weaves" recursion into the datatype definition, and the base structure describes its shape (i.e., number of data constructors and their types). One can program with two-level types in any functional language that supports higher-order polymorphism[2], such as Haskell. In Figure 6, we illustrate this by giving an example of a two level definition for ordinary lists (all the other types in this paper have similar definitions).

The use of two-level types has been recognized as a useful functional programming pearl [21], since two-level types separate the two concerns of (1) recursion on recursive sub components and (2) handling different cases (by pattern matching over the shape of the (non-recursive) base structure). An advantage of such an approach, is that a single eliminator can be defined once for all datatypes of the same kind. For example, the function $\texttt{mit}_\kappa$ describes Mendler-style iteration[3] for the recursive types defined by μ_κ. Although it is possible to write programs using two level datatypes in a general purpose functional language, one could not expect logical consistency in such systems.

[2] a.k.a. higher-kinded polymorphism, or type-constructor polymorphism

[3] An iteration is a principled recursion scheme guaranteed to terminate for any well-founded input. Also known as fold, or catamorphism.

$$
\begin{array}{lcl}
\texttt{List} & = & \lambda A^*.\forall X^*.(A \to X \to X) \to X \to X \\
\texttt{cons} & : & \forall A^*.A \to \texttt{List}\,A \to \texttt{List}\,A \\
 & & \qquad = \lambda x_a.\lambda x.\lambda x_c.\lambda x_n.x_c\,x_a\,(x\;x_c\,x_n) \\
\texttt{nil} & : & \forall A^*.\texttt{List}\,A \;=\; \lambda x_c.\lambda x_n.\lambda x_n \\
\texttt{elim}_{\texttt{List}} & : & \forall A^*.\texttt{List}\,A \to \forall X^*.(A \to X \to X) \to X \to X \\
 & = & \lambda x.\lambda x_c.\lambda x_n.x\;x_c\,x_n \qquad (\textit{foldr}\;x_z\;x_c\;x \text{ in Haskell}) \\
\hline
\texttt{Powl} & = & \lambda A^*. \\
 & & \forall X^{*\to *}.(A \to X(A \times A) \to XA) \to XA \to XA \\
\texttt{pcons} & : & \forall A^*.A \to \texttt{Powl}(A \times A) \to \texttt{Powl}\,A \\
 & & \qquad = \lambda x_a.\lambda x.\lambda x_c.\lambda x_n.x_c\,x_a\,(x\;x_c\,x_n) \\
\texttt{pnil} & : & \forall A^*.\texttt{Powl}\,A \;=\; \lambda x_c.\lambda x_n.\lambda x_n \\
\texttt{elim}_{\texttt{Powl}} & : & \forall A^*.\texttt{Powl}\,A \to \\
 & & \forall X^{*\to *}.(A \to X(A \times A) \to XA) \to XA \to XA \\
 & = & \lambda x.\lambda x_c.\lambda x_n.x\;x_c\,x_n \\
\hline
\texttt{Vec} & = & \lambda A^*.\lambda i^{\texttt{Nat}}. \\
 & & \forall X^{\texttt{Nat}\to *}.(\forall i^{\texttt{Nat}}.A \to X\{i\} \to X\{\texttt{S}\,i\}) \to \\
 & & \qquad X\{\texttt{Z}\} \to X\{i\} \\
\texttt{vcons} & : & \forall A^*.\forall i^{\texttt{Nat}}.A \to \texttt{Vec}\,A\,\{i\} \to \texttt{Vec}\,A\,\{\texttt{S}\,i\} \\
 & & \qquad = \lambda x_a.\lambda x.\lambda x_c.\lambda x_n.x_c\,x_a\,(x\;x_c\,x_n) \\
\texttt{vnil} & : & \forall A^*.\texttt{Vec}\,A\,\{\texttt{Z}\} \;=\; \lambda x_c.\lambda x_n.x_n \\
\texttt{elim}_{\texttt{Vec}} & : & \forall A^*.\forall i^{\texttt{Nat}}.\texttt{Vec}\,A\,\{i\} \to \\
 & & \forall X^{\texttt{Nat}\to *}.(\forall i^{\texttt{Nat}}.A \to X\{i\} \to X\{\texttt{S}\,i\}) \to \\
 & & \qquad X\{\texttt{Z}\} \to X\{i\} \\
 & = & \lambda x.\lambda x_c.\lambda x_n.x\;x_c\,x_n
\end{array}
$$

Figure 5. Embedding recursive datatypes

Interestingly, there exist embeddings of the recursive type operator μ_κ, its data constructor $\texttt{In}_\kappa$, and the Mendler-style iterator $\texttt{mit}_\kappa$ for each kind κ into the higher-order polymorphic λ-calculus F_i, as illustrated in Figure 7. In addition to illustrating the general form of embedding μ_κ, we also fully expand the embeddings for some instances (μ_*, $\mu_{*\to *}$, $\mu_{\texttt{Nat}\to *}$), which are used in Figure 6. These embeddings support the embedding of arbitrary type- and term-indexed recursive datatypes into System F_i. Thus we can reason about these datatypes in a logically consistent calculus.

However, it is important to note that there does not exist an embedding of the arbitrary destruction (or, pattern matching away) of the $\texttt{In}_\kappa$ constructor. It is known that combining arbitrary recursive datatypes with the ability to destruct (or, unroll) their values is powerful enough to define non-terminating computations in a type safe way, leading to logical inconsistency. Some systems maintain consistency by restricting which recursive datatypes can be defined, but allow arbitrary unrolling. In System F_i, we can define any datatype, but restrict unrolling to Mendler style operators definable in F_i. Such operators are quite expressive, capturing at least iteration, primitive recursion, and courses of values recursion.

Example 1. *The datatype of λ-terms in context*

```
data Lam ( C: Nat -> * ) { i: Nat } where
  LVar : C{i} -> Lam{i}
  LApp : Lam{i} -> Lam{i} -> Lam{i}
  LAbs : Lam{S i} -> Lam{i}
```

```
newtype μ* (f :: * -> *)
  = In* (f (μ* f))

data ListF (a::*) (r::*)
  = Cons a r        | Nil

type List a = μ* (ListF a)
cons x xs = In* (Cons x xs)
nil       = In* Nil

mit* :: (∀ r.(r->x) -> f r -> x) -> Mu0 f -> x
mit* phi (In* z) = phi (mit* phi) z

newtype μ(*→*) (f :: (*->*) -> (*->*)) (a::*)
  = In(*→*) (f (Mu(*→*) f)) a

data PowlF (r::*->*) (a::*)
  = PCons a (r(a,a)) | PNil

type Powl a = μ(*→*) PowlF a
pcons x xs = In(*→*) (PCons x xs)
pnil       = In(*→*) PNil

mit(*→*) :: (∀ r a.(∀a.r a->x a) -> f r a -> x a)
         -> μ(*→*) f a -> x a
mit(*→*) phi (In(*→*) z) = phi (mit(*→*) phi) z

-- above is Haskell (with some GHC extensions)
-- below is Haskell-ish pseudocode

newtype μ(Nat→*) (f::(Nat->*)->(Nat->*)) {n::Nat}
  = In(Nat→*) (f (μ(Nat→*) f)) {n}

data VecF (a::*) (r::Nat->*) {n::Nat} where
  VCons :: a -> r n -> VecF a r {S n}
  VNil  :: VecF a r {Z}

type Vec a {n::Nat} = μ(Nat→*) (VecF a) {n}
vcons x xs = In(Nat→*) (VCons x xs)
vnil       = In(Nat→*) VNil

mit(Nat→*)::(∀ r n.(∀n.r{n}->x{n})->f r {n}->x{n})
         -> μ(Nat→*) f {n} -> x{n}
mit(Nat→*) phi (In(Nat→*) z) = phi (mit(Nat→*) phi) z
```

Figure 6. 2-level types and their Mendler-style iterators in Haskell

is encoded as:

$$
\begin{array}{l}
\texttt{Lam} \triangleq \lambda C^{\texttt{Nat}\to *}\lambda i^{\texttt{Nat}}.\forall X^{\texttt{Nat}\to *}. \\
\qquad (\forall j^{\texttt{Nat}}.C\{j\} \to X\{j\}) \\
\qquad\quad \to (\forall j^{\texttt{Nat}}.X\{j\} \to X\{j\} \to X\{j\}) \\
\qquad\qquad \to (\forall j^{\texttt{Nat}}.X\{\texttt{S}\,j\} \to X\{j\}) \\
\qquad\qquad\quad \to X\{i\}
\end{array}
$$

For a concrete representation one can consider `Lam Fin` *where*

```
data Fin { i: Nat } where
  FZ : Fin{S i}
  FS : Fin{i} -> Fin{S i}
```

This is encoded as

$$
\begin{array}{l}
\texttt{Fin} \triangleq \lambda i^{\texttt{Nat}}.\forall X^{\texttt{Nat}\to *}. \\
\qquad (\forall j^{\texttt{Nat}}.X\{\texttt{S}\,j\}) \to (\forall j^{\texttt{Nat}}.X\{j\} \to X\{\texttt{S}\,j\}) \\
\qquad\quad \to X\{i\}
\end{array}
$$

notation: $\boldsymbol{\lambda}\mathbb{I}^\kappa.F = \lambda I_1^{\kappa_1}.\cdots.\lambda I_n^{\kappa_n}.F \qquad \forall\mathbb{I}^\kappa.B = \forall I_1^{\kappa_1}.\cdots.\forall I_n^{\kappa_n}.B \qquad F\mathbb{I} = F I_1 \cdots I_n \qquad F \xrightarrow{\kappa} G = \forall\mathbb{I}^\kappa.F\mathbb{I} \to G\mathbb{I}$

where $\kappa = \kappa_1 \to \cdots \to \kappa_n \to *$ and I_i is an index variable (i_i) when κ_i is a type,
$\mathbb{I} = I_1, \ \ldots \ \ldots \ , \ I_n$ a type constructor variable (X_i) otherwise.

$$\mu_\kappa \ : \ (\kappa \to \kappa) \to \kappa \qquad = \lambda F^{\kappa\to\kappa}.\boldsymbol{\lambda}\mathbb{I}^\kappa.\forall X^\kappa.(\forall X_r^\kappa.(X_r \xrightarrow{\kappa} X) \to (F X_r \xrightarrow{\kappa} X)) \to X\mathbb{I}$$

$$\mu_* \ : \ (* \to *) \to * \qquad = \lambda F^{*\to *}. \qquad \forall X^*.(\forall X_r^*.(X_r \to X) \to (F\,X_r \to X)) \to X$$

$$\mu_{*\to *} \ : \ ((* \to *) \to (* \to *)) \to (* \to *)$$

$$= \lambda F^{(*\to *)\to(*\to *)}.\lambda X_1^*.\forall X^{*\to *}.(\forall X_r^{*\to *}.(\forall X_1^*.X_r X_1 \to X X_1) \to (\forall X_1^*.F\,X_r X_1 \to X X_1)) \to X X_1$$

$$\mu_{\mathtt{Nat}\to *} \ : \ ((\mathtt{Nat} \to *) \to (\mathtt{Nat} \to *)) \to (\mathtt{Nat} \to *)$$

$$= \lambda F^{(\mathtt{Nat}\to *)\to(\mathtt{Nat}\to *)}.\lambda i_1^{\mathtt{Nat}}.\forall X^{\mathtt{Nat}\to *}.(\forall X_r^{\mathtt{Nat}\to *}.(\forall i_1^{\mathtt{Nat}}.X_r i_1 \to X i_1) \to (\forall i_1^{\mathtt{Nat}}.F\,X_r i_1 \to X i_1)) \to X i_1$$

$$\mathtt{In}_\kappa \ : \ \forall F^{\kappa\to\kappa}.F(\mu_\kappa F) \xrightarrow{\kappa} \mu_\kappa F \qquad = \lambda x_r.\lambda x_\varphi.x_\varphi\,(\mathtt{mit}_\kappa\ x_\varphi)\,x_r$$

$$\mathtt{mit}_\kappa \ : \ \forall F^{\kappa\to\kappa}.\forall X^\kappa.(\forall X_r^\kappa.(X_r \xrightarrow{\kappa} X) \to (F X_r \xrightarrow{\kappa} X)) \to (\mu_\kappa F \xrightarrow{\kappa} X) \qquad = \lambda x_\varphi.\lambda x_r.x_r\,x_\varphi$$

Figure 7. Embedding of the recursive operators (μ_κ), their data constructors ($\mathtt{In}_\kappa$), and the Mendler-style iterators ($\mathtt{mit}_\kappa$).

4.3 Leibniz index equality

The quantification over type-indexed kinding available in System F_i allows the definition of *Leibniz-equality type* constructors $\mathtt{Eq}_A : A \to A \to *$ on closed types A, defined as follows:

$$\mathtt{Eq}_A \triangleq \lambda i^A.\,\lambda j^A.\,\mathtt{LEq}_A\{i\}\{j\} \times \mathtt{LEq}_A\{j\}\{i\} \ ,$$
$$\text{where } \mathtt{LEq}_A \triangleq \lambda i^A.\,\lambda j^A.\,\forall X^{A\to *}.\,X\{i\} \to X\{j\} \ .$$

For $F_A \in \{\mathtt{Eq}_A, \mathtt{LEq}_A\}$, observe that the following types are inhabited:

(Reflexive) $\forall i^A.\,F_A\{i\}\{i\}$

(Transitive) $\forall i^A.\,\forall j^A.\,\forall k^A.\,F_A\{i\}\{j\} \to F_A\{j\}\{k\} \to F_A\{i\}\{k\}$

(Logical) $\forall i^A.\,\forall j^A.\,F_A\{i\}\{j\} \to \forall f^{A\to B}.\,F_B\{f\,i\}\{f\,j\}$
$\forall f^{A\to B}.\,\forall g^{A\to B}.\,F_{A\to B}\{f\}\{g\} \to \forall i^A.\,F_B\{f\,i\}\{g\,i\}$

Hence Leibniz equality is a congruence; in that, in addition to the above one also has the inhabitation of the type

(Symmetric) $\forall i^A.\,\forall j^A.\,\mathtt{Eq}_A\{i\}\{j\} \to \mathtt{Eq}_A\{j\}\{i\}$

In applications, the types $\mathtt{LEq}_A$ are useful in constraining the term-indexing of datatypes. A general such construction is given by the type constructors $\mathtt{Ran}_{A,B} : (A \to B) \to (A \to *) \to B \to *$. These are defined as

$$\mathtt{Ran}_{A,B} \triangleq \lambda f^{A\to B}.\,\lambda X^{A\to *}.\,\lambda j^B.\forall i^A.\,\mathtt{LEq}_B\{j\}\{f\,i\} \to X\{i\}$$

and are in spirit right Kan extensions, a notion that is being extensively used in programming, e.g. [2, 12]. One of their usefulness comes from the fact that the following type is inhabited by a section

$$\forall Y^{B\to *}.\,\forall X^{A\to *}.\,\forall f^{A\to B}.$$
$$\big(\forall i^A.\,Y\{f\,i\} \to X\{i\}\big) \to \big(\forall j^B.\,Y\{j\} \to (\mathtt{Ran}_{A,B}\{f\}X)\{j\}\big)$$

This allows one to represent functions from input datatypes with constrained indices as plain indexed functions, and vice versa. For instance, by means of the iterators of the previous section one can define a vector tail function of type

$$\forall X^*.\,\forall j^{\mathtt{Nat}}.\,\mathtt{Vec}\,X\,\{j\} \to \big(\mathtt{Ran}_{\mathtt{Nat},\mathtt{Nat}}\,\{\mathtt{S}\}(\mathtt{Vec}\,X)\big)\{j\}$$

and retract it to one of type

$$\forall X^*.\,\forall i^{\mathtt{Nat}}.\,\mathtt{Vec}\,X\,\{\mathtt{S}\,i\} \to \mathtt{Vec}\,X\,\{i\} \ .$$

Analogously, one can use an iterator to define a single-variable capture-avoiding substitution function of type

$$\forall i^{\mathtt{Nat}}.\,(\mathtt{Lam\,Fin})\{i\}$$
$$\to \big(\mathtt{Ran}_{\mathtt{Nat},\mathtt{Nat}}\{\mathtt{S}\}(\lambda j^{\mathtt{Nat}}.\,\mathtt{Lam\,Fin}\{j\} \to \mathtt{Lam\,Fin}\{j\})\big)\{i\}$$

and then retract it to one of type

$$\forall i^{\mathtt{Nat}}.\,(\mathtt{Lam\,Fin})\{\mathtt{S}\,i\} \to (\mathtt{Lam\,Fin})\{i\} \to (\mathtt{Lam\,Fin})\{i\} \ .$$

Type constructors $\mathtt{Lan}_{A,B} : (A \to B) \to (A \to *) \to B \to *$, which are in spirit left Kan extensions, permit the encoding of functions of type $(\forall i^A.\,F\{i\} \to G\{t\,i\})$, for $F : A \to *$, $G : B \to *$, and $t : A \to B$, as functions of type $(\forall j^B.\,(\mathtt{Lan}_{A,B}\{t\}F)\{j\} \to G\{j\})$. Left Kan extensions are dual to right Kan extensions, but to define them as such one needs existential and product types. In formalisms without them, these have to be encoded. This can be done as follows:

$$\mathtt{Lan}_{A,B} \triangleq \lambda f^{A\to B}.\,\lambda X^{A\to *}.\,\lambda j^B.$$
$$\forall Z^*.\,(\forall i^A.\,\mathtt{LEq}_B\{f\,i\}\{j\} \to X\{i\} \to Z) \to Z$$

The type

$$\forall X^{A\to *}.\,\forall Y^{B\to *}.\,\forall f^{A\to B}.$$
$$(\forall i^A.\,X\{i\} \to Y\{f\,i\}) \to (\forall j^B.\,(\mathtt{Lan}_{A,B}\{f\}X)\{j\} \to Y\{j\})$$

is thus inhabited by a section, providing a retractable coercion between the two functional representations.

Left Kan extensions come with a canonical section of type $\forall f^{A\to B}.\,\forall X^{A\to *}.\,\forall i^A.\,X\{i\} \to (\mathtt{Lan}_{A,B}\{f\}X)\{f\,i\}$ that, according to a reindexing function $t : A \to B$, embeds an A-indexed type F (at index s) into the B-indexed type $\mathtt{Lan}_{A,B}\{t\}F$ (at index $t\,s$). For instance, the type constructor $\mathtt{Lan}_{A,A\times A}\{\lambda x.\,\mathtt{pair}\,x\,x\}$ embeds arrays of types into matrices along the diagonal; while the type constructors $\mathtt{Lan}_{A\times A,A}\{\mathtt{fst}\}$ and $\mathtt{Lan}_{A\times A,A}\{\mathtt{snd}\}$ respectively encapsulate matrices of types as arrays by columns and by rows.

5. Metatheory

The expectation is that System F_i has all the nice properties of System F_ω, yet is more expressive because of the addition of term-indexed types.

We show some basic well-formedness properties for the judgments of F_i in §5.1. We prove erasure properties of F_i, which captures the idea that indices are erasable since they are irrelevant for

reduction in §5.2. We show strong normalization, logical consistence, and subject reduction for F_i by reasoning about well-known calculi related to F_i in §5.3.

5.1 Well-formedness properties and substitution lemmas

We want to show that the sorting, kinding, and typing derivations give well-formed results under well-formed contexts. That is, sorting derivations result in well-formed sorts (Proposition 1), kinding derivations result in well-sorted kinds under well-formed type level contexts (Proposition 2), and typing derivations result in well-kinded types under well-formed type and term level contexts (Proposition 3).

Since the definitions of sorting, kinding, and typing rules are mutually recursive, these three properties are considered as one big property (illustrated below) in order to be more rigorous abouts the induction principle used in the proof.

Proposition (The big well-formedness property of F_i, roughly[4])**.**

case $\boxed{\vdash \kappa : \square}$ *(Proposition 1)* $\quad \dfrac{\vdash \kappa : \mathfrak{s}}{\mathfrak{s} = \square}$

case $\boxed{\Delta \vdash F : \kappa}$ *(Proposition 2)* $\quad \dfrac{\vdash \Delta \quad \Delta \vdash F : \kappa}{\vdash \kappa : \square}$

case $\boxed{\Delta; \Gamma \vdash t : A}$ *(Proposition 3)* $\quad \dfrac{\Delta \vdash \Gamma \quad \Delta; \Gamma \vdash t : A}{\Delta \vdash A : *}$

The big well-formedness property has one of the three forms – $\boxed{\vdash \kappa : \square}$ (sorting), $\boxed{\Delta \vdash F : \kappa}$ (kinding), and $\boxed{\Delta; \Gamma \vdash t : A}$ typing. That is, a derivation for a judgment of either sorting, kinding, or typing results in either a well-formed sort (when it is a sorting judgment), a well-sorted kind (when it is a kinding judgment), or a well-kinded type (when it is a typing judgment), under well-formed contexts for the judgment (no context for sorting judgments, Δ for kinding judgments, and $\Delta; \Gamma$ for typing judgments).

We can prove the big well-formedness property of F_i by induction on the derivation of a judgment, which can be any one of the three forms. Here, we illustrate the proof for the three propositions as if they were separate proofs. Because it provides a more intuitive proof sketch, during the proof description, the proof for each proposition references the other properties (which are yet another application of the induction hypothesis of the big well-formedness property). So, when we say "by induction" during the proofs, what we really mean is the induction hypothesis of the big well-formedness property.

Proposition 1 (sorting derivations result in well-formed sorts)**.**

$$\frac{\vdash \kappa : \mathfrak{s}}{\mathfrak{s} = \square}$$

Proof. Obvious since $\square$ is the only sort in F_i. ■

Proposition 2 (kinding derivations under well-formed contexts result in well-sorted kinds)**.**

$$\frac{\vdash \Delta \quad \Delta \vdash F : \kappa}{\vdash \kappa : \square}$$

Proof. By induction on the derivation.

[4] Technically, this is not yet completely rigorous since there are three more forms of judgments in the mutually recursive definition. The *kind equality*, *type considered equality*, and *term equality* rules are part of the mutually recursive definition along with the sorting, kinding, and typing rules. So, the complete description of the big well-formedness property will consist of six cases, which correspond to Proposition 1, Proposition 2, Proposition 3, Lemma 1, Lemma 2, and Lemma 3.

case (Var) Trivial by the second well-formedness rule of Δ.

case $(Conv)$ By induction and Lemma 1.

case (λ) By induction and Proposition 1 we know that $\vdash \kappa : \square$.
By the second well-formedness rule of Δ, we know that $\vdash \Delta, X^\kappa$ since we already know that $\vdash \kappa : \square$ and $\vdash \Delta$ from the property statement.
By induction, we know that $\vdash \kappa' : \square$ since we already know that $\vdash \Delta, X^\kappa$ and that $\Delta, X^\kappa \vdash F : \kappa'$ from induction hypothesis.
By the sorting rule (R), we know that $\vdash \kappa \to \kappa' : \square$ since we already know that $\vdash \kappa : \square$ and $\vdash \kappa' : \square$.

case $(@)$ By induction, easy.

case (λi) By induction we know that $\cdot \vdash A : *$. By the third well-formedness rule of Δ, we know that $\vdash \Delta, i^A$ since we already know that $\cdot \vdash A : *$ and that $\vdash \Delta$ from the property statement.
By induction, we know that $\vdash \kappa : \square$ since we already know that $\vdash \Delta, i^A$ and that $\Delta, i^A \vdash F : \kappa$ from the induction hypothesis.
By the sorting rule (Ri), we know that $\vdash A \to \kappa : \square$ since we already know that $\cdot \vdash A : *$ and $\vdash \kappa : \square$.

case $(@i)$ By induction and Proposition 3, easy.

case $(\to)$ Trivial since $\vdash * : \square$.

case $(\forall)$ Trivial since $\vdash * : \square$.

case $(\forall i)$ Trivial since $\vdash * : \square$. ■

The basic structure of the proof for the following proposition on typing derivations is similar to above. So, we illustrate the proof for most of the cases, which can be done by applying the induction hypothesis, rather bravely. We elaborate more on interesting cases $(\forall E)$ and $(\forall Ei)$ which involve substitutions in the types resulting from the typing judgments.

Proposition 3 (typing derivations under well-formed contexts result in well-kinded types)**.**

$$\frac{\Delta \vdash \Gamma \quad \Delta; \Gamma \vdash t : A}{\Delta \vdash A : *}$$

Proof. By induction on the derivation.

case $(:)$ Trivial by the second well-formedness rule of Γ.

case $(: i)$ Trivial by the third the well-formedness rule of Δ.

case $(=)$ By induction and Lemma 2.

case $(\to I)$ By induction and well-formedness of Γ.

case $(\to E)$ By induction.

case $(\forall I)$ By induction and well-formedness of Δ.

case $(\forall E)$ By induction we know that $\Delta \vdash \forall X^\kappa.B : *$.
By the kinding rule $(\forall)$, which is the only kinding rule able to derive $\Delta \vdash \forall X^\kappa.B : *$, we know that $\Delta, X^\kappa \vdash B : *$.
Then, we use the type substitution lemma (Lemma 4(1)).

case $(\forall Ii)$ By induction and well-formedness of Δ.

case $(\forall Ei)$ By induction we know that $\Delta \vdash \forall i^A.B : *$.
By the kinding rule $(\forall i)$, which is the only kinding rule able to derive $\Delta \vdash \forall i^A.B : *$, we know that $\Delta, i^A \vdash B : *$.
Then, we use the index substitution lemma (Lemma 4(2)). ■

Lemma 1 (kind equality is well-sorted)**.** $\dfrac{\vdash \kappa = \kappa' : \square}{\vdash \kappa : \square \quad \vdash \kappa' : \square}$

Proof. By induction on the derivation of kind equality and using the sorting rules. ■

Lemma 2 (type constructor equality is well-kinded)**.**

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F : \kappa \quad \Delta \vdash F' : \kappa}$$

Proof. By induction on the derivation of type constructor equality and using the kinding rules. Also use the type substitution lemma (Lemma 4(1)) and the index substitution lemma (Lemma 4(2)). ■

Lemma 3 (term equality is well-typed)**.**

$$\frac{\Delta, \Gamma \vdash t = t' : A}{\Delta, \Gamma \vdash t : A \quad \Delta, \Gamma \vdash t' : A}$$

Proof. By induction on the derivation of term equality and using the typing rules. Also use the term substitution lemma (Lemma 4(3)). ■

The proofs for the three lemmas above are straightforward once we have dealt with the interesting cases for the equality rules involving substitution. We can prove those interesting cases by applying the substitution lemmas. The other cases fall into two categories: firstly, the equality rules following the same structure of the sorting, kinding, and typing rules; and secondly, the reflexive rules and the transitive rules. The proof for the equality rules following the same structure of the sorting, kinding, and typing rules can be proved by induction and applying the corresponding sorting, kinding, and typing rules. The proof for the reflexive rules and the transitive rules can be proved simply by induction.

Lemma 4 (substitution)**.**

1. (type substitution) $\dfrac{\Delta, X^{\kappa} \vdash F : \kappa' \qquad \Delta \vdash G : \kappa}{\Delta \vdash F[G/X] : \kappa'}$
2. (index substitution) $\dfrac{\Delta, i^{A} \vdash F : \kappa \qquad \Delta; \cdot \vdash s : A}{\Delta \vdash F[s/i] : \kappa}$
3. (term substitution) $\dfrac{\Delta; \Gamma, x : A \vdash t : B \qquad \Delta; \Gamma \vdash s : A}{\Delta, \Gamma \vdash t[s/x] : B}$

The substitution lemma is fairly standard, comparable to substitution lemmas in other well-known systems such as F_{ω} or ICC.

5.2 Erasure properties

We define a meta-operation of index erasure that projects F_i-types to F_{ω}-types.

Definition 1 (index erasure)**.**

$$\boxed{\kappa^{\circ}} \quad *^{\circ} = * \qquad (\kappa_1 \to \kappa_2)^{\circ} = {\kappa_1}^{\circ} \to {\kappa_2}^{\circ} \qquad (A \to \kappa)^{\circ} = \kappa^{\circ}$$

$$\boxed{F^{\circ}} \quad X^{\circ} = X \qquad (A \to B)^{\circ} = A^{\circ} \to B^{\circ}$$

$$(\lambda X^{\kappa}.F)^{\circ} = \lambda X^{{\kappa}^{\circ}}.F^{\circ} \qquad (\lambda i^{A}.F)^{\circ} = F^{\circ}$$

$$(F\ G)^{\circ} = F^{\circ}\ G^{\circ} \qquad (F\ \{s\})^{\circ} = F^{\circ}$$

$$(\forall X^{\kappa}.B)^{\circ} = \forall X^{{\kappa}^{\circ}}.B^{\circ} \qquad (\forall i^{A}.B)^{\circ} = B^{\circ}$$

$$\boxed{\Delta^{\circ}} \quad \cdot^{\circ} = \cdot \qquad (\Delta, X^{\kappa})^{\circ} = \Delta^{\circ}, X^{{\kappa}^{\circ}} \qquad (\Delta, i^{A})^{\circ} = \Delta^{\circ}$$

$$\boxed{\Gamma^{\circ}} \quad \cdot^{\circ} = \cdot \qquad (\Gamma, x : A)^{\circ} = \Gamma^{\circ}, x : A^{\circ}$$

Example 2. *The meta-operation of index erasure simply discards all indexing information. The effect of this on most datatypes is to project the indexing invariants while retaining the type structure. This is clearly seen for the vector type constructor* `Vec` *whose index erasure is the list type constructor* `List`*, see Figure 5. One can however build pathological examples. For instance, the type* $\mathtt{P}_A \triangleq \forall i^{A}.\forall j^{A}.\mathtt{LEq}_A\{i\}\{j\}$ *has index erasure* $\mathtt{Unit} \triangleq \forall X^{*}.X \to X$*.*

Theorem 1 (index erasure on well-sorted kinds)**.** $\dfrac{\vdash \kappa : \Box}{\vdash \kappa^{\circ} : \Box}$

Proof. By induction on the sorting derivation. ■

Remark 1. For any well-sorted kind κ in F_i, κ° is a kind in F_{ω}.

Theorem 2 (index erasure on well-formed type level contexts)**.**

$$\frac{\vdash \Delta}{\vdash \Delta^{\circ}}$$

Proof. By induction on the derivation for well-formed type level context and using Theorem 1. ■

Remark 2. For any well-formed type level context Δ in F_i, Δ° is a well-formed type level context in F_{ω}.

Theorem 3 (index erasure on kind equality)**.** $\dfrac{\vdash \kappa = \kappa' : \Box}{\vdash \kappa^{\circ} = \kappa'^{\circ} : \Box}$

Proof. By induction on the kind equality judgement. ■

Remark 3. For any well-sorted kind equality $\vdash \kappa = \kappa' : \Box$ in F_i, $\vdash \kappa^{\circ} = \kappa'^{\circ} : \Box$ is a well-sorted kind equality in F_{ω}.

The three theorems above on kinds are rather simple to prove since there is no need to consider mutual recursion in the definition of kinds due to the erasure operation on kinds. Recall that the erasure operation on kinds discards the type (A) appearing in the index arrow type ($A \to \kappa$). So, there is no need to consider the types appearing in kinds and the index terms appearing in those types, after the erasure.

Theorem 4 (index erasure on well-kinded type constructors)**.**

$$\frac{\vdash \Delta \qquad \Delta \vdash F : \kappa}{\Delta^{\circ} \vdash F^{\circ} : \kappa^{\circ}}$$

Proof. By induction on the kinding derivation.

case (Var) Use Theorem 2.
case ($Conv$) By induction and using Theorem 3.
case (λ) By induction and using Theorem 1.
case (@) By induction.
case (λi) We need to show that $\Delta^{\circ} \vdash (\lambda i^{A}.F)^{\circ} : (A \to \kappa)^{\circ}$, which simplifies to $\Delta^{\circ} \vdash F^{\circ} : \kappa^{\circ}$ by Definition 1.
By induction, we know that $(\Delta, i^{A})^{\circ} \vdash F^{\circ} : \kappa^{\circ}$, which simplifies $\Delta^{\circ} \vdash F^{\circ} : \kappa^{\circ}$ by Definition 1.
case (@i) We need to show that $\Delta^{\circ} \vdash (F\ \{s\})^{\circ} : \kappa^{\circ}$, which simplifies to $\Delta^{\circ} \vdash F^{\circ} : \kappa^{\circ}$ by Definition 1.
By induction we know that $\Delta^{\circ} \vdash F^{\circ} : (A \to \kappa)^{\circ}$, which simplifies to $\Delta^{\circ} \vdash F^{\circ} : \kappa^{\circ}$ by Definition 1.
case ($\to$) By induction.
case ($\forall$) We need to show that $\Delta^{\circ} \vdash (\forall X^{\kappa}.B)^{\circ} : *^{\circ}$, which simplifies to $\Delta^{\circ} \vdash \forall X^{{\kappa}^{\circ}}.B^{\circ} : *$ by Definition 1.
Using Theorem 1, we know that $\vdash \kappa^{\circ} : \Box$.
By induction we know that $(\Delta, X^{\kappa})^{\circ} \vdash B^{\circ} : *^{\circ}$, which simplifies to $\Delta^{\circ}, X^{{\kappa}^{\circ}} \vdash B^{\circ} : *$ by Definition 1.
Using the kinding rule ($\forall$), we get exactly what we need to show: $\Delta^{\circ} \vdash \forall X^{{\kappa}^{\circ}}.B^{\circ} : *$.
case ($\forall i$) We need to show that $\Delta^{\circ} \vdash (\forall i^{A}.B)^{\circ} : *^{\circ}$, which simplifies to $\Delta^{\circ} \vdash B^{\circ} : *$ by Definition 1.
By induction we know that $(\Delta, i^{A})^{\circ} \vdash B^{\circ} : *^{\circ}$, which simplifies $\Delta^{\circ} \vdash B^{\circ} : *$ by Definition 1. ■

Theorem 5 (index erasure on type constructor equality)**.**

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta^{\circ} \vdash F^{\circ} = F'^{\circ} : \kappa^{\circ}}$$

Proof. By induction on the derivation of type constructor equality.

Most of the cases are done by applying the induction hypothesis and sometimes using Proposition 2.

The only interesting cases, which are worth elaborating on, are the equality rules involving substitution. There are two such rules.

$$\frac{\Delta, X^{\kappa} \vdash F : \kappa' \qquad \Delta \vdash G : \kappa}{\Delta \vdash (\lambda X^{\kappa}.F)\, G = F[G/X] : \kappa'}$$

We need to show $\Delta^\circ \vdash ((\lambda X^{\kappa}.F)\, G)^\circ = (F[G/X])^\circ : \kappa'^\circ$, which simplifies to $\Delta^\circ \vdash (\lambda X^{\kappa^\circ}.F^\circ)\, G^\circ = (F[G/X])^\circ : \kappa'^\circ$ by Definition 1.

By induction, we know that $(\Delta, X^{\kappa})^\circ \vdash F^\circ : \kappa'^\circ$, which simplifies to $\Delta^\circ, X^{\kappa^\circ} \vdash F^\circ : \kappa'^\circ$ by Definition 1.

Using the kinding rule (λ), we get $\Delta^\circ \vdash \lambda X^{\kappa^\circ}.F^\circ : \kappa^\circ \to \kappa'^\circ$.

Using the kinding rule (@), we get $\Delta^\circ \vdash (\lambda X^{\kappa^\circ}.F^\circ)G^\circ : \kappa'^\circ$.

Using the very equality rule of this case,

we get $\Delta^\circ \vdash (\lambda X^{\kappa^\circ} F^\circ)G^\circ = F^\circ[G^\circ/X] : \kappa'^\circ$.

All we need to check is $(F[G/X])^\circ = F^\circ[G^\circ/X]$, which is easy to see.

$$\frac{\Delta, i^{A} \vdash F : \kappa \qquad \Delta; \cdot \vdash s : A}{\Delta \vdash (\lambda i^{A}.F)\, \{s\} = F[s/i] : \kappa}$$

By induction we know that $\Delta^\circ \vdash F^\circ : \kappa^\circ$.

The erasure of the left hand side of the equality is $((\lambda i^{A}.F)\, \{s\})^\circ = (\lambda i^{A}.F)^\circ = F^\circ$.

All we need to show is $(F[s/i])^\circ = F^\circ$, which is obvious since index variables can only occur in index terms and index terms are always erased. Recall the index erasure over type constructors in Definition 1; in particular, $(\lambda i^{A}.F)^\circ = F^\circ$, $(F\{s\})^\circ = F^\circ$, and $(\forall i^{A}.B)^\circ = B^\circ$. ■

Remark 4. For any well-kinded type constructor equality $\Delta \vdash F = F' : \kappa$ in F_i, $\Delta^\circ \vdash F^\circ = F'^\circ : \kappa^\circ$ is a well-kinded type constructor equality in F_ω.

The proofs for the two theorems above on type constructors need not consider mutual recursion in the definition of type constructors due to the erasure operation. Recall that the erasure operation on type constructors discards the index term (s) appearing in the index application ($F\ \{s\}$). So, there is no need to consider the index terms appearing in the types after the erasure.

Theorem 6 (index erasure on well-formed term level contexts)**.**

$$\frac{\Delta \vdash \Gamma}{\Delta^\circ \vdash \Gamma^\circ}$$

Proof. By induction on Γ.

case ($\Gamma = \cdot$) It trivially holds.

case ($\Gamma = \Gamma', x : A$), we know that $\Delta \vdash \Gamma'$ and $\Delta \vdash A : *$ by the well-formedness rules and that $\Delta^\circ \vdash \Gamma'^\circ$ by induction.
From $\Delta \vdash A : *$, we know that $\Delta^\circ \vdash A^\circ : *$ by Theorem 4.
We know that $\Delta^\circ \vdash \Gamma'^\circ, x : A^\circ$ from $\Delta^\circ \vdash \Gamma'^\circ$ and $\Delta^\circ \vdash A^\circ : *$ by the well-formedness rules.
Since $\Gamma'^\circ, x : A^\circ = (\Gamma', x : A)^\circ = \Gamma^\circ$ by definition, we know that $\Delta^\circ \vdash \Gamma^\circ$.

■

Theorem 7 (index erasure on index-free well-typed terms)**.**

$$\frac{\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash t : A}{\Delta^\circ; \Gamma^\circ \vdash t : A^\circ} \quad (\mathsf{dom}(\Delta) \cap \mathrm{FV}(t) = \emptyset)$$

Proof. By induction on the typing derivation. Interesting cases are the index related rules (: i), ($\forall Ii$), and ($\forall Ei$). Proofs for the other cases are straightforward by induction and applying other erasure theorems corresponding to the judgment forms.

case (:) By Theorem 6, we know that $\Delta^\circ \vdash \Gamma^\circ$ when $\Delta \vdash \Gamma$. By definition of erasure on term-level context, we know that $(x : A^\circ) \in \Gamma^\circ$ when $(x : A) \in \Gamma$.

case (: i) Vacuously true since t does not contain any index variables (i.e., $\mathsf{dom}(\Delta) \cap \mathrm{FV}(t) = \emptyset$).

case ($\to I$) By Theorem 4, we know that $\cdot \vdash A^\circ : *$. By induction, we know that $\Delta^\circ; \Gamma^\circ, x : A^\circ \vdash t^\circ : B^\circ$. Applying the ($\to I$) rule to what we know, we have $\Delta^\circ; \Gamma^\circ \vdash \lambda x.t^\circ : A^\circ \to B^\circ$.

case ($\to E$) Straightforward by induction.

case ($\forall I$) By Theorem 1, we know that $\vdash \kappa^\circ : \Box$. By induction, we know that $\Delta^\circ, X^{\kappa^\circ}; \Gamma^\circ \vdash t : B^\circ$. Applying the ($\forall I$) rule to what we know, we have $\Delta^\circ; \Gamma^\circ \vdash t : \forall X^{\kappa^\circ}.B^\circ$.

case ($\forall E$) By induction, we know that $\Delta^\circ; \Gamma^\circ \vdash t : \forall X^{\kappa^\circ}.B^\circ$. By Theorem 4, we know that $\Delta^\circ \vdash G^\circ : \kappa^\circ$. Applying the ($\forall E$) rule, we have $\Delta^\circ; \Gamma^\circ \vdash t : B^\circ[G^\circ/X]$.

case ($\forall Ii$) By Theorem 4, we know that $\cdot \vdash A^\circ : *$. By induction, we know that $\Delta^\circ; \Gamma^\circ \vdash t : B^\circ$, which is what we want since $(\forall i^{A}.B)^\circ = B^\circ$.

case ($\forall Ei$) By induction, we know that $\Delta^\circ; \Gamma^\circ \vdash t : B^\circ$, which is what we want since $(B[s/i])^\circ = B^\circ$.

case (=) By Theorem 5 and induction. ■

Example 3. *The theorem yields that the pathological type* $\mathtt{P}_A$ *of Example 2 is not inhabited, as it is impossible to have both* $t : \mathtt{P}_A$ *and* $t : (\mathtt{P}_A)^\circ = \mathtt{Unit}$*. It follows as a corollary that the implication of Theorem 7 does not admit a converse.*

In this context for $A = \mathtt{Void}$*, note that even though one has* $i^{\mathtt{Void}}; \cdot \vdash \lambda x.\, i : \forall j^{\mathtt{Void}}.\forall X^{\mathtt{Void}\to *}.\, X\{i\} \to X\{j\}$*, this open term cannot be closed by rule* ($\forall Ii$) *because of its side condition. This is in stark contrast to what is possible in calculi with full type dependency. In System* F_i*, the index variables in type level context* Δ *cannot appear dynamically at term level. Conversely, term variables in the term level context* Γ *cannot be used for instantiation of index polymorphic types (rule* ($\forall Ei$)*).*

Similar considerations to the above show that $\mathtt{LEq}_A$ *is not symmetric, in that the type* (Symmetric) *in §4.3 is not inhabited.*

We introduce an index variable selection meta-operation that selects all the index variable bindings from the type level context.

Definition 2 (index variable selection)**.**

$$\cdot^\bullet = \cdot \qquad (\Delta, X^{A})^\bullet = \Delta^\bullet \qquad (\Delta, i^{A})^\bullet = \Delta^\bullet, i : A$$

Theorem 8 (index erasure on well-formed term level contexts prepended by index variable selection)**.**

$$\frac{\Delta \vdash \Gamma}{\Delta^\circ \vdash (\Delta^\bullet, \Gamma)^\circ}$$

Proof. Straightforward by Theorem 6 and the typing rule (: i). ■

The following result is the appropriate version of Theorem 7 without the side condition therein.

Theorem 9 (index erasure on well-typed terms)**.**

$$\frac{\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash t : A}{\Delta^\circ; (\Delta^\bullet, \Gamma)^\circ \vdash t : A^\circ}$$

Proof. The proof is almost the same as that of Theorem 7, except for the (: i) case. The proof for the rule (: i) case is easy since $(i : A) \in \Delta^\bullet$ when $i^{A} \in \Delta$ by definition of the index variable selection operation. The indices from Δ being prepended to Γ do not affect the proof for the other cases. ■

5.3 Strong normalization and logical consistency

Strong normalization is a corollary of the erasure property since we know that System F_ω is strongly normalizing. Logical consistency is immediate since System F_i is a strict subset of the *restricted implicit calculus* [15], which is in turn a restriction of ICC [16]. Subject reduction is also immediate for the same reason.

6. Related work

System F_i is most closely related to Curry-style System F_ω[1, 2, 9] and the Implicit Calculus of Constructions (ICC) [16]. All terms typable in a Curry-style System F_ω are typable (with the same type) in System F_i and all terms typable in F_i are typable (with the same type[5]) in ICC.

We can derive strong normalization, logical consistency, and subject reduction of System F_i, from both System F_ω and a subset of ICC. In fact, ICC is more than just an extension of System $\mathsf{F}_i$with dependent types and stratified universes. ICC includes η-reduction and the extensionality typing rule. We do not foresee any problems adding η-reduction and the extensionality typing rule to System F_i. Although System F_i accepts fewer terms than ICC, it enjoys simpler erasure properties (Theorem 7 and Theorem 9), which ICC cannot enjoy due to its support for full dependent types. In System F_i, index terms appearing in types (e.g., s in $F\{s\}$) are always erasable. Mishra-Linger and Sheard [17] generalized the ICC framework to one which describes erasure on arbitrary Church-style calculi (EPTS) and Curry-style calculi (IPTS), but they only consider β-equivalence for type conversion.

We mentioned (§3.1) that Curry-style calculi enjoy better reduction properties (e.g. $\beta\eta$-reduction is Church-Rosser) than Church-style calculi. For Church-style terms with $\beta\eta$-reduction, Nederpelt [19] gave a counterexample to the Church-Rosser property. Geuvers [8] proved that $\beta\eta$-reduction is Church-Rosser in functional PTSs, which are special classes of Church-style calculi. Seldin [20] discusses the relationship between the Church-style typing and the Curry-style typing.

In the practical setting of programming language implementations, Yorgey et al. [27], inspired by McBride [14], designed an extension to Haskell, allowing datatypes to be used as kinds. For instance, `Bool` is promoted to a kind (i.e., $\texttt{Bool} : \Box$) and its data constructors `True` and `False` are promoted to types. To support this, they extended System F_C (The Glasgow Haskell Compiler's (GHC) intermediate core language), naming the extension System $F_C^\uparrow$. The key difference between $F_C^\uparrow$ and F_i is the kind syntax, as illustrated below:

$$F_C^\uparrow \textbf{ kinds}: \quad \kappa ::= * \mid \kappa \to \kappa \mid F\vec{\kappa} \mid \mathcal{X} \mid \forall \mathcal{X}.\kappa \mid \cdots$$
$$\mathsf{F}_i \textbf{ kinds}: \quad \kappa ::= * \mid \kappa \to \kappa \mid A \to \kappa$$

In $F_C^\uparrow$, all type constructors (F) are promotable to the kind level and become kinds when fully applied to other kinds ($F\vec{\kappa}$). On the other hand, in F_i, a type can only appear as the domain of an index arrow kind ($A \to \kappa$). This seemingly small difference allows $F_C^\uparrow$ to be a much more expressive language than F_i. The promotion of a type constructor, for instance, $\texttt{List} : * \to *$ to a kind constructor $\texttt{List} : \Box \to \Box$ enables type-level data structures such as $[\texttt{Nat}, \texttt{Bool}, \texttt{Nat} \to \texttt{Bool}] : \texttt{List}\ *$. Type-level data structures motivate type-level computations over promoted data. This is made possible by type families[6]. The promotion of polymorphic types naturally motivates kind polymorphism ($\forall \mathcal{X}.\kappa$), which is known to break strong normalization and cause logical inconsistency [10]. In a functional *programming language*, inconsistency is not an issue. However, when studying term-indexed datatypes in a logically consistent calculi, we need a more conservative approach, as in F_i.

System F_i is the smallest possible extension to F_ω that we could devise that maintains normalization and consistency. An alternative is to restrict a system with full-spectrum dependent types. Swamy et al. [24] developed F^*, a language for secure distributed programming with value dependent types. Terms appearing in dependent types in F^* are restricted to first-order values, similar to the value restriction of ML type inference. Taming dependent types with this restriction, they were able to have a usable programming language and self-certify [22] their compiler by implementing F^* type checker in F^*.

The Literature about type equality constraints in systems supporting GADTs is vast. We list just a few. System F_C [23] is arguably the most influential system, being the core language of GHC. Vytiniotis and Weirich [25] proved parametricity of System R_ω [7] (an extension to Curry-style System F_ω with the type-representation datatype and its primitive recursor), so that one may derive *free theorems* [26] in the presence of type equalities.

7. Conclusion and Future work

System F_i is a strongly-normalizing, logically-consistent, higher-order polymorphic lambda calculus that was designed to support the definition of datatypes indexed by both terms and types. In terms of expressivity, System F_i sits between System F_ω and ICC. We designed System F_i as a tool to reason about programming languages with term-indexed datatypes.

We have applied this tool to the design of the programming language Nax (not yet published). Nax is given semantics in terms of System F_i. In Nax, Mendler style operators are primitive operators with their own typing rules. Nax has been designed to be expressive over the Hindley-Milner subset of System F_i. It supports type inference with minimal typing annotations. We believe this is an advantage made possible because our extensions to F_ω are all static. This would be made much more difficult had we restricted ICC.

Typing annotations in Nax are necessary only on case statements (for non-recursive term-indexed datatypes) and Mendler-style operators (for recursive term-indexed datatypes). Programs involving only type-indexing require no annotations elsewhere. A typing annotation takes a limited form of a large elimination, which is an abstraction over both type- and term-indices to types (e.g., $X, i_1, i_2 \mapsto FX\{i_1 + i_2\}$), which is somewhat similar to the convoy pattern idiom [5] found in Coq scripts to aid type checking dependent case expressions. Future work includes richer form of large elimination, which enables selection of different type constructors for the result type of case statements and Mendler-style operators (e.g., $X, i_1, i_2 \mapsto$ **if** $i_1 < i_2$ **then** $F_1\, X\, \{i_1\}$ **else** $F_2\, X\, \{i_2\}$). Enriching the type annotations in Nax will motivate us to identify the features needed to extend F_i to support a notion of large eliminations.

Acknowledgments

This work was supported by NSF grant 0910500.

References

[1] A. Abel, R. Matthes, and T. Uustalu. Generalized iteration and coiteration for higher-order nested datatypes. In *FoSSaCS*, volume 2620 of *LNCS*, pages 54–69. Springer, 2003.

[2] A. Abel, R. Matthes, and T. Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1-2):3 – 66, 2005.

[3] K. Y. Ahn and T. Sheard. A hierarchy of Mendler-style recursion combinators: Taming inductive datatypes with negative occurrences. In *ICFP '11*, pages 234–246. ACM, 2011.

[5] The $*$ kind in F_ω and F_i corresponds to Set in ICC

[6] A GHC extension to define type-level functions in Haskell.

[4] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[5] A. Chlipala. Certified programming with dependent types. To be published real soon. URL `http://adam.chlipala.net/cpdt/`.

[6] A. Church. A set of postulates for the foundation of logic (2nd paper). *Annals of Mathematics*, 34(4):839–864, Oct. 1933.

[7] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP '98*, pages 301–312. ACM, 1998.

[8] H. Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 453–460, June 1992.

[9] P. Giannini, F. Honsell, and S. R. D. Rocca. Type inference: Some results, some problems. *Fundam. Inform.*, 19(1/2):87–125, 1993.

[10] Girard, J.-Y. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, June 1972.

[11] L.-J. Guillemette. *A Type-Preserving Compiler from System F to Typed Assembly Language*. PhD thesis, Oct. 2010.

[12] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *POPL*, pages 297–308, 2008.

[13] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[14] C. McBride. The Strathclyde Haskell Enhancement. URL `http://personal.cis.strath.ac.uk/conor/pub/she/`.

[15] A. Miquel. A model for impredicative type systems, universes, intersection types and subtyping. In *LICS*, pages 18–29. IEEE Computer Society, 2000.

[16] A. Miquel. The implicit calculus of constructions. In *TLCA*, pages 344–359, 2001.

[17] N. Mishra-Linger and T. Sheard. Erasure and polymorphism in pure type systems. In R. M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2008.

[18] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[19] R. P. Nederpelt. *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types*. PhD thesis, Eindhoven University of Technology, 1973.

[20] J. P. Seldin. On the relation between the Church-style typing and the Curry-style typing. Submitted to Journal of Applied Logic. URL `http://people.uleth.ca/~jonathan.seldin/RCS2.pdf`.

[21] T. Sheard and E. Pasalic. Two-level types and parameterized modules. *J. Funct. Program.*, 14(5):547–587, Sept. 2004. ISSN 0956-7968.

[22] P.-Y. Strub, N. Swamy, C. Fournet, and J. Chen. Self-certification: bootstrapping certified typecheckers in F* with Coq. In *POPL '12*, pages 571–584. ACM, 2012.

[23] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System f with type equality coercions. In *TLDI '07*, TLDI '07, pages 53–66. ACM, 2007.

[24] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP '11*, pages 266–278. ACM, 2011.

[25] D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Program.*, 20(2):175–210, 2010.

[26] P. Wadler. Theorems for free! In *FPCA '89*, pages 347–359. ACM, 1989.

[27] B. A. Yorgey, S. Weirich, J. Cretin, S. L. P. Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *TLDI*, pages 53–66. ACM, 2012.

Revision: 2605 (Wed, 23 Mar 2011)

Security Type Error Diagnosis

Jeroen Weijers
Wilhelm-Schickard Institute of Computer Science,
Universität Tübingen
Sand 13, 72076 Tübingen, Germany
jeroen.weijers@uni-tuebingen.de

Jurriaan Hage
Dept. of Inf. and Comp. Sciences,
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
J.Hage@uu.nl

Stefan Holdermans
Vector Fabrics
Paradijslaan 28, 5611 KN Eindhoven,
The Netherlands
stefan@vectorfabrics.com

Abstract

We consider how to improve error diagnosis for a security analysis defined for a simply-typed call by value lambda calculus extended with lists, pairs and the security specific constructs declassify and protect. We first provide a non-standard type system in which types are annotated with security levels, and discuss its associated constraint-based implementation. We can then define and motivate heuristics that help diagnose inconsistencies among the constraints, and show their effect on a selection of security incorrect programs.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Functional constructs, Type structure

General Terms Languages, Theory

Keywords type-based program analysis, security analysis, error feedback

1. Introduction

We take for granted that a compiler for a strongly-typed language refuses to generate code for a program that is not type correct. Nowadays, we also expect the compiler to provide a reasonable explanation of what is wrong, as long as the languages features we use are not very experimental. This has not always been the case. As the literature study of Heeren [12] documents, the problem of type error diagnosis for the Hindley-Milner type system, and several extensions thereof, has been extensively studied, and we believe that this work has had its effects on modern compilers of, say, Haskell. Thus far, however, most effort in this area has been directed to the intrinsic type systems of functional programming languages, and not to other validation-oriented type based analyses such as security analysis [30].

Consider the following expression (taken from [13]; this is not code anyone would care to write) for which we have provided explicit type and *security* annotations:

$$\textbf{if } (\mathit{True} :: \mathtt{Bool}^{\mathsf{H}}) \textbf{ then } (\mathit{True} :: \mathtt{Bool}^{\mathsf{L}}) \textbf{ else } (\mathit{False} :: \mathtt{Bool}^{\mathsf{L}}) :: \mathtt{Bool}^{\mathsf{L}}$$

In this expression, an annotation H means that the expression to which it is attached, delivers values that are highly confidential. Expressions annotated with L deliver values of low confidentiality. The purpose of security analysis is to verify that data is never leaked to expressions that may also evaluate to less confidential data. Intuitively, in a security correct program no value may inadvertently influence a value of a lesser security level. In the expression above, this is not the case: because the value of the condition decides whether the then or else part must be evaluated, so observing the value of the low confidentiality result reveals information about the highly confidential conditional. Therefore, the expression is not security type correct (although it *is* type correct) and it should be rejected. The problem can be fixed by changing the annotation on the condition to L, or by changing the annotation on the complete expression to H.

King et al [17] observed that information-flow reporting techniques are inadequate to explain security type errors. In their work they assign information-flow blame, and provide traces of the Java programs they analyze to show how values of high confidentiality end up in locations that may only expose values of lower confidentiality. Their work, however, does not transfer easily to a functional setting: their analysis is (largely) context-insensitive (i.e., monovariant), the language they consider is first-order, there is no discussion of parametric polymorphism, and their trace-like explanation does not seem so natural for a higher-order functional language. Moreover, as the authors themself suggest, their work has not been combined with heuristics to further prune the traces they provide (see, e.g., [15] on the CQual tool, and [9] for work on Hindley-Milner).

The paper offers the following contributions:

- We address the problem of security type error diagnostics for a polyvariant lambda-calculus extended with recursion, lists and tuples, and special security specific constructs, **declassify** and **protect**. This language is strongly influenced by FlowCaml [24].
- We are the first to combine the type error slicing approach of Haack and Wells [8] with the heuristic approach of Heeren [12].
- We introduce and motivate a number of heuristics, divided into four essentially different categories, as described in Section 2. We provide a substantial number of example programs that how our approach works, and
- provide a prototype implementation of our work obtainable at `http://www.cs.uu.nl/people/jur/sfunplusplus.zip`

```
File "Login.fml", line 11, characters 0-144:
This expression generates the following
    information flow:
root < everyone
which is not legal.
```

Figure 1. Error message for FlowCaml.

The paper is structured as follows. We describe our approach in Section 2. In Section 3 we define the subject language, specify the security annotated type system and indicate how the type system can be transformed into an inference algorithm. In Section 4 we then describe the heuristics we have devised, and Section 5 gives further examples as a first step in validating our work. In Section 6 we discuss related work, and Section 7 concludes. For reasons of space we omit many details, in particular, example code for the FlowCaml system and SecLib library, and the complete inference algorithm and constraint solver that implement the security type system. These details can be obtained from [33]; substantial parts have also been included in the supplemental material as appendices as a courtesy to the reviewers.

2. Approach

In this section we describe our approach and the heuristics that come into play, described at a high level. The type based security analysis we provide error diagnosis for is polyvariant and includes a rule for subeffecting, but not full subtyping. The rule for subeffecting is somewhat more restrictive than that of full subtyping, but this is not relevant for our work here (and note that the loss precision is to a large extent compensated for by the polyvariance of the analysis). Our source language is a higher-order polymorphic functional language, very much akin to FlowCaml, an implementation based on the Core ML language [23, 24].

In a FlowCaml program the programmer describes the relations that must hold in the lattice of security levels, e.g., $!everyone < !root$. If, during analysis, the analyzer finds that $!root < !everyone$, then an error has occurred, and this is communicated to the programmer. An example error message of this kind is given in Figure 1. The message explains there is an illegal flow, and the location points to the line where the definition where the inconsistency is detected begins. Although correct, the message does not explain how the flow was derived, which subexpressions were responsible, and what the programmer can do to fix the problem. For reasons of space, we omit the example program written in FlowCaml and further discussion (see Appendix A for details).

Essentially, our work combines the approaches of Hage and Heeren [9, 12], and Haack and Wells [8]. Like Haack and Wells, we first compute a (security) type error slice when a security type error has been found. A security type error slice is a program slice (or fragment) that only contains those parts of the program that contribute to the error. The constraints that are needed to construct such a slice together form a minimal unsatisfiable set of constraints: remove any of its elements, and it becomes satisfiable. If a program contains multiple errors, then the next error will be revealed only after the first one has been corrected.

For completeness, we repeat the details on how to compute a minimal unsatisfiable set of constraints from the complete set of constraints. Haack and Wells [8], and Stuckey, Sulzmann and Wazny [28] both present an algorithm for computing such a set. We follow the algorithm presented by Stuckey, Sulzmann and Wazny in Section 7 of [28], as displayed in Figure 2. The algorithm takes an unsatisfiable set of constraints D and constructs a minimal unsatisfiable set of constraints M iteratively. The algorithm consists of two nested while loops: the inner loop adds a constraint from

$minUnsat\ (D) = \mathbf{do}$
  $M = \{\}$
  while satisfiable M {
    $C = M$
    while satisfiable C {
      let $e \in D - C$
      $C = C \cup \{e\}$
    }
    $D = C$
    $M = M \cup \{e\}$
  }
  return M

Figure 2. Computing a minimal unsatisfiable set of constraints

the original set D minus C, to a copy, called C, of the unsatisfiable set collected thus far. When C becomes unsatisfiable in the inner loop, then the last added constraint is known to contribute to the error, and it is then added to the minimal unsatisfiable set M. This process continues until the set M becomes unsatisfiable.

Displaying the security type error slice is a first approximation for the type error, but in some cases there may be strong evidence that a smaller set of locations will do just as well, or that we can suggest a fix for the mistake. In Section 4, we present a number of heuristics that inspect the constraints in the minimal unsatisfiable set to determine whether certain constraints/locations should never be marked as the cause of a security type error or, just the opposite, a particular constraint should be blamed for the mistake. In the latter case, a very specific security type error message can be provided. Because a compiler cannot know what the intentions of the programmer are, we are taking a risk here. This risk can be mitigated by, for example, offering various security error messages and allowing the programmer to scan through these. Our implementation currently offers only one error message. Adding a facility such as we just described is only a matter of engineering.

There are two good reasons to start from a minimal unsatisfiable set of constraints. First, it is impossible to blame a constraint that cannot be responsible for the mistake (which may be considered a "soundness property" for the heuristics). Second, the heuristics need only look at a restricted set of constraints, which we may hope is much smaller than the complete set of constraints.

In the end, we will be left with a set of constraints that will receive the blame for the mistake. When a constraint is generated, meta information about the AST node where it was generated is added to the constraint. The collection of AST nodes associated with the constraints in the minimal unsatisfiable subset together form the program slice. This slice can in itself be presented as an error message, with some explanation on the nature of the error [8].

Because our analysis is polyvariant, simplification/solving of constraints will take place for every definition, i.e., at any point that generalisation is to take place. During this process of simplification (a call to *simplify* in the algorithm, see the definition of generalisation in Section 3), the error diagnosis process we have just sketched will be invoked whenever simplification results in an inconsistency.

The heuristics we have implemented can be divided into four categories:

- generic heuristics that borrow heavily from earlier work and apply just as well in the current setting, e.g., a heuristic that filters out constraints that equate the security level of a let-expression with that of the let-body.
- propagation heuristics that prevent blaming code that only propagates the security levels of their inputs. For example, blaming a

function inc that increments an integer value (and that implicitly maintains the level of confidentiality of its input) cannot be sensibly blamed for an inconsistency. In practice, we expect most functions to be of this kind. Changes in levels of confidentiality are most likely to arise from implicit control-flow (see the example in the introduction), from security specific operations like **protect** and **declassify** and from values explicitly provided with security annotations.

- heuristics derived from the assumption that programmers may be used to dealing with the intrinsic type system, and will be unaware of the subtle differences that arise from the fact that a security type system is in fact a dependency analysis [2]. We systematically derive these heuristics from observable differences between the specification of security typing and the underlying intrinsic type system.
- heuristics that are specific to security analysis, in particular the operations of **declassify** and **protect**.

Although, we have no evidence to support this, we believe that many of the heuristics and our approach can be reused in other settings besides security analysis. For example, the heuristics in the third category are also likely to apply to other dependency analyses.

3. Security type system

The language we use is based upon the Fun language (see Chapter 3 of [20]), a simply typed, call-by-value lambda calculus. We have extended Fun with a few special purpose security program constructs as well as some additional features to make the analysis and examples more interesting. We call this extended language sFun++.

In this paper we employ the following syntactic categories

n	$\in$	**Nat**	*natural numbers* ,
b	$\in$	**Bool**	*booleans* ,
e	$\in$	**Exp**	*expressions* ,
f, x	$\in$	**Var**	*variables* ,
$u, \oplus$	$\in$	**Op'**, **Op**	*unary and binary operators* ,
p	$\in$	**Prog**	*program* ,
d	$\in$	**Decl**	*declarations* ,
s	$\in$	**Sec**	*security levels*

Most categories should speak for themselves. We note that the category **Sec** ranges over security levels, which we assume to form a lattice [4], meaning that given a finite non-empty set S of security levels, there is a unique lowest security level that is at least as secure as each element of S. The join operator of this lattice is, as usual, denoted by $\sqcup$.

The abstract syntax of our language is defined as:

$$
\begin{array}{lcl}
p & ::= & d^* \\
d & ::= & f = e_1 \\
e & ::= & n \mid b \mid x \mid \textbf{fn}\ x \Rightarrow e_0 \mid \textbf{fun}\ f\ x \Rightarrow e_0 \\
 & \mid & e_0\ e_1 \mid \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
 & \mid & \textbf{let}\ x = e_0\ \textbf{in}\ e_1 \mid e_1 \oplus e_2 \mid u\ e_1 \\
 & \mid & \textbf{Cons}\ e_1\ e_2 \mid \textbf{Nil} \mid (e_1, e_2) \\
 & \mid & \textbf{fst}\ e_1 \mid \textbf{snd}\ e_1 \mid \textbf{null}\ e_1 \mid \textbf{hd}\ e_1 \mid \textbf{tl}\ e_1 \\
 & \mid & \textbf{declassify}\ e_0\ s \mid \textbf{protect}\ e_0\ s
\end{array}
$$

A program p is a list of declarations, and each declaration binds an expression to an identifier. As in [20], **fn** $x \Rightarrow e_0$ defines a non-recursive function and **fun** $f\ x \Rightarrow e_0$ a recursive one. In the latter, the identifier f refers to the recursively defined function. Function application is left associative. Local definitions **let** $x = e_0$ **in** e_1 are non-recursive. Top level declarations are syntactic sugar for a nested let. This means that a declaration can only use functions that are declared earlier in the program. For data types we have pairs (e_1, e_2), and lists are built from **Cons** and **Nil** as usual. Pairs are destructed by **fst** and **snd**, and **hd** and **tl** destruct lists. Finally, we can test for the empty list with the **null** predicate.

The two security constructs are **protect** and **declassify**. The expression **protect** $e_0\ s$ increases the level of protection (security) of an expression e_0 to level s. It is important to note that this construct can only increase the security level of e_0, so level s has to be at least as secure as the level at which e_0 was previously protected. The expression **declassify** $e_0\ s$ does exactly the opposite: it decreases the security level of e_0 to level s. The presence of this construct implies that our analysis is not sound with respect to confidentiality. However, without some form of declassification it will be hard to write useful programs. For example, we cannot write a valid program that informs an unauthorised user that he or she entered an invalid password (assuming the password information is confidential).

3.1 The sFun++ type language

As usual, we specify the security type system as an annotated type system ([19], and Chapter 5 of [20]). Our security analysis is polyvariant, which means that we can quantify over annotation variables. The relations between annotation variables, and restrictions on them are expressed as constraints. These constraints may be added to types, in the style of qualified types [16].

We introduce the following new syntactic categories:

α	$\in$	**TyVar**	*type variables*
β	$\in$	**AnnVar**	*annotation variables*
π	$\in$	**Constr**	*constraints*
l	$\in$	**Levels**	*security levels*
φ	$\in$	**Ann**	*security annotations*
τ	$\in$	**Ty**	*annotated types*
ρ	$\in$	**Qualified Types**	*qualified types*
σ	$\in$	**TyScheme**	*annotated type schemes*
C	$\in$	**Constraints**	*constraint set*
Γ	$\in$	**TyEnv**	*type environments*

The sets of annotation variables, **AnnVar**, and type variables, **TyVar** are assumed to be mutually disjoint. An annotation is either some security level l (taken from any given security lattice), or an annotation variable β. Constraints relate two security levels, where we take $\varphi_1 \sqsubseteq \varphi_2$ to mean that φ_2 is at least as secure as φ_1.

$$
\begin{array}{lcl}
\varphi & ::= & l \mid \beta \\
\pi & ::= & \varphi_1 \sqsubseteq \varphi_2
\end{array}
$$

We then define a three-layer type language:

$$
\begin{array}{lcl}
\tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{List}\ \tau^{\varphi} \\
 & \mid & (\tau_1{}^{\varphi}, \tau_2{}^{\varphi}) \mid \tau_1{}^{\varphi} \to \tau_2{}^{\varphi} \mid \alpha \\
\rho & ::= & \pi \Rightarrow \rho \mid \tau \\
\sigma & ::= & \forall \alpha.\, \sigma \mid \forall \beta.\, \sigma \mid \exists \beta.\, \sigma \mid \rho
\end{array}
$$

The first layer, τ, consists of annotated types for the primitive types, lists, pairs and function types. We introduce type variables in order to be able to construct type schemes. Qualified types ρ consist of a type and a sequence of constraints that further restrict the type. Finally, type schemes allow us to quantify universally over type and annotation variables, and existentially only over annotation variables. The latter facility is used to deal with annotation type variables that may be constrained in some way, but that are not exposed as part of the type (see also [7]). Note that in contrast to FlowCaml [24] we do have annotations on pairs. These only exists for reasons of uniformity with other data types.

A type environment Γ is a mapping from variables x to a pair consisting of a type scheme σ and a top-level annotation for x.

$$
\begin{array}{lcl}
\Gamma & ::= & \emptyset \mid \Gamma\,[x \mapsto (\sigma, \varphi)] \\
C & ::= & \emptyset \mid \{\pi_1, \ldots, \pi_n\} \cup C
\end{array}
$$

The pair associated with a variable x in an environment Γ is written $\Gamma\ (x)$; it returns a pair associated with the rightmost occurrence of x. Note that top level annotations are not present in type schemes, but are stored separately in the type environment. As a result, we can not quantify over these annotations, and therefore declarations will never be polyvariant in their top level annotation. Although one might think that this causes a loss of expressivity, the presence of a rule for sub-effecting will counter this loss. Constraint sets C will be used to store a constraint environment in our typing judgment.

3.2 The security type system

In Figure 3 we give the non syntax directed type rules for our security analysis. The main judgment is $\Gamma, C \vdash e : \sigma^{\varphi}$, which reads "under the type environment Γ and constraint environment C, the expression e can have type σ and is protected at level φ". The constraint environment C contains the relations between security levels that define the security lattice proper, as well as the constraints that should hold for e.

We note that some of the rules in Figure 3 may specify a family of rules. For example, in the rule [*t-true*], φ ranges over the elements of the chosen lattice, and thereby provides a single rule for each such element. In the case of [*t-list*], we have a rule for each combination of choices for φ_1 and φ_2. Also note that the rule *t-int* specifies an infinite family of rules: for every combination of an integer n and an element of the security lattice φ. We shall now discuss the more interesting rules one by one.

As seen in rules [*t-Nil*] and [*t-Cons*], lists have separate security annotations for the elements and the structure of the list. Moreover, these annotations are independent: we can, say, protect the length of a list at a higher level than the actual contents, and vice versa. Obviously, for the Cons case, the annotation on the new element φ_1 should correspond to the annotation on the elements of the list, and the annotation on the argument list φ propagates to the result list. For rules [*t-null*], [*t-hd*] and [*t-tl*] we note that the functions **null**, **hd** and **tl** all reveal information about the structure of the list, and **hd** additionally reveals information about the contents of the list; hence the least upper bound in the consequent of [*t-hd*]. As mentioned before, for reasons of consistency pairs also have a top level annotation, although we learn nothing from knowing the structure of a pair that the type hasn't already conveyed (see [*t-Pair*], [*t-fst*] and [*t-snd*]).

Function application ([*t-app*]) requires the annotation of the provided argument to be equal to the annotation of the expected argument. The annotation of the application result is the least upper bound of the security level of the result, and of the result of applying the function. The reason for the former, is that applying a function may reveal information about that function. Functions declared at top level are assigned the lowest security level $\bot$, so then the annotation of the application only depends on the annotation on the result of the body; the same reasoning applies to operators in the rules [*t-Bin-op*] and [*t-Un-op*].

For the rule [*t-if*], recall from the Introduction that the security annotation on the condition should also propagate to the security level of the result of the conditional. Otherwise, the outcome may leak secure information accessed during the evaluation of the condition.

The influence of **protect** and **declassify** on the security levels are expressed in the rules [*t-protect*] and [*t-declass*], respectively. The expression **protect** $e_0\ \varphi_0$ is protected at level φ_0, under the condition that e_0 is at most as secure as φ_0. Declassification does exactly the opposite, lowering the level of security.

As evidenced by many of our rules, we typically insist that different subexpressions have exactly the same annotated type. In the rule [*t-if*], for example, the condition, the then part and the else part need to have exactly the same security level. This is by itself too restrictive, making reasonable programs unanalysable. Therefore, a rule for subeffecting, [*t-sub*], is introduced that may increase the level of protection for an expression.

The rules for annotation generalisation and instantiation, [*t-ann-gen*] and [*t-ann-ins*], are analogous to the standard rules [*t-gen*] and [*t-ins*]. The rule [*t-gen*] uses the function $ftv(x)$ to compute the free type variables in x, and [*t-ann-gen*] uses a similar function *fav* to compute the free annotation variables. The functions are straightforward, and we omit the details.

Qualification ([*t-qual*]) allows us to move a constraint from the constraint set into the (qualified) type, and resolution ([*t-res*]) allows us to do the opposite. In Figure 4 rules for reasoning with constraints are provided. The rule [*c-in*] states that if π is in C then π holds. Transitivity is provided through the rule of [*c-trans*], and reflexivity through [*c-reflex*]. The rules [*c-bot*] and [*c-top*] state that any φ is above $\bot$ and below $\top$.

Our type system specification follows those described in [2, 24]. We are confident therefore that — omitting the rule for declassification —, the security type system satisfies a non-interference result. Intuitively, noninterference imples that replacing a expression of some confidentiality with any other (of the same level of confidentiality), does not change the values of any expression of lower confidentiality. Since these properties are well-known, and our focus in this paper is on deriving heuristics from the security type system, we forego the definition of semantics that we need to formally specify the non-interference result.

3.3 Towards an algorithm

In this section we follow Chapter 5 of [20]. In order to arrive at an algorithm, we first turn the type system of Figure 3 into a type system that is completely syntax-directed.

As usual, we replace the rules [*t-var*], [*t-ins*] and [*t-ann-ins*] with a single rule for variables that immediately instantiates all quantified type and annotation variables (with a fresh variable of the right kind), thereby resulting in a type (and not a type scheme). Similarly, generalisation is merged into the rule [*t-let*], in that it takes the monotype found for the let-definition, generalises over all free type and annotation variables, and attaches the resulting type scheme to the let-defined identifier in the environment so that it may be used inside the let-body. We give more details on the function gen that performs the generalisation later in this section.

The syntax-directed type system can then be transformed into an inference algorithm in the style of algorithm W [3]. Here we follow the standard approach of dealing with annotations on types: the traversal of the abstract syntax tree computes the underlying types of the expressions, and introduces fresh annotation variables wherever necessary. Annotations in types are restricted to annotation variables, so that unification of annotated types is much simplified. At this time, the security annotations themselves are not yet computed, but, instead, concrete security information is expressed by constraints between annotations, which are collected during the traversal. These constraints can then later be solved by a worklist algorithm, yielding the concrete security annotations for the expressions in the program. For example, for the rule [*t-app*] we introduce a fresh annotation variable, β say, to represent the security level of the applcation $e_1\ e_2$, and express its security level by constraints $\varphi \sqsubseteq \beta$ and $\varphi_1 \sqsubseteq \beta$. In this way, we can express the entire analysis in terms of constraints of the form $\_ \sqsubseteq \_$.

However, we must make one further adjustment to this process, because the process we have just decribed only works for monovariant analysis.

Let-generalisation is performed on the annotated type of a let-defined identifier before it is stored inside an annotated type environment and passed to the body of the let. At this time, we have not

Typing judgements $\boxed{\Gamma, C \vdash e : \sigma^{\varphi}}$

$$\frac{}{\Gamma, C \vdash n : \mathtt{Int}^{\varphi}}\ [t\text{-}int] \qquad \frac{}{\Gamma, C \vdash \mathbf{true} : \mathtt{Bool}^{\varphi}}\ [t\text{-}true] \qquad \frac{}{\Gamma, C \vdash \mathbf{false} : \mathtt{Bool}^{\varphi}}\ [t\text{-}false]$$

$$\frac{}{\Gamma, C \vdash \mathbf{Nil} : (\mathtt{List}\ \tau^{\varphi_2})^{\varphi_1}}\ [t\text{-}Nil] \qquad \frac{\Gamma, C \vdash e_1 : \tau^{\varphi_1} \quad \Gamma, C \vdash e_2 : (\mathtt{List}\ \tau^{\varphi_1})^{\varphi}}{\Gamma, C \vdash \mathbf{Cons}\ e_1\ e_2 : (\mathtt{List}\ \tau^{\varphi_1})^{\varphi}}\ [t\text{-}Cons]$$

$$\frac{\Gamma, C \vdash e_1 : {\tau_1}^{\varphi_1} \quad \Gamma, C \vdash e_2 : {\tau_2}^{\varphi_2}}{\Gamma, C \vdash (e_1, e_2) : ({\tau_1}^{\varphi_1}, {\tau_2}^{\varphi_2})^{\varphi}}\ [t\text{-}Pair] \qquad \frac{\Gamma\,(x) = (\sigma, \varphi)}{\Gamma, C \vdash x : \sigma^{\varphi}}\ [t\text{-}var]$$

$$\frac{\Gamma\,[x \mapsto (\tau_x, \varphi_x)], C \vdash e_0 : {\tau_0}^{\varphi_0}}{\Gamma, C \vdash \mathbf{fn}\ x \Rightarrow e_0 : {\tau_x}^{\varphi_x} \rightarrow^{\varphi} {\tau_0}^{\varphi_0}}\ [t\text{-}fn] \qquad \frac{\Gamma\,[f \mapsto ({\tau_x}^{\varphi_x} \rightarrow {\tau_0}^{\varphi_0}, \varphi)]\,[x \mapsto (\tau_x, \varphi_x)], C \vdash e_0 : {\tau_0}^{\varphi_0}}{\Gamma, C \vdash \mathbf{fun}\ f\ x \Rightarrow e_0 : {\tau_x}^{\varphi_x} \rightarrow^{\varphi} {\tau_0}^{\varphi_0}}\ [t\text{-}fun]$$

$$\frac{\Gamma, C \vdash e_1 : {\tau_2}^{\varphi_2} \rightarrow^{\varphi} {\tau_0}^{\varphi_0} \quad \Gamma, C \vdash e_2 : {\tau_2}^{\varphi_2}}{\Gamma, C \vdash e_1\ e_2 : {\tau_0}^{\varphi \sqcup \varphi_0}}\ [t\text{-}app] \qquad \frac{\Gamma, C \vdash e_0 : \mathtt{Bool}^{\varphi} \quad \Gamma, C \vdash e_1 : \tau^{\varphi} \quad \Gamma, C \vdash e_2 : \tau^{\varphi}}{\Gamma, C \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau^{\varphi}}\ [t\text{-}if]$$

$$\frac{\Gamma, C \vdash e_1 : \sigma^{\varphi} \quad \Gamma\,[x \mapsto (\sigma, \varphi)], C \vdash e_2 : {\tau_2}^{\varphi_2}}{\Gamma, C \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : {\tau_2}^{\varphi_2}}\ [t\text{-}let]$$

$$\frac{\Gamma, C \vdash e_1 : {\tau^1_{\oplus}}^{\varphi} \quad \Gamma, C \vdash e_2 : {\tau^2_{\oplus}}^{\varphi}}{\Gamma, C \vdash e_1\ \oplus\ e_2 : {\tau_{\oplus}}^{\varphi}}\ [t\text{-}Bin\text{-}op] \qquad \frac{\Gamma, C \vdash e_1 : {\tau^1_{\oplus}}^{\varphi}}{\Gamma, C \vdash u\ e_1 : {\tau_{\oplus}}^{\varphi}}\ [t\text{-}Un\text{-}op]$$

$$\frac{\Gamma, C \vdash e_1 : ({\tau_1}^{\varphi_1}, {\tau_2}^{\varphi_2})^{\varphi}}{\Gamma, C \vdash \mathbf{fst}\ e_1 : {\tau_1}^{\varphi \sqcup \varphi_1}}\ [t\text{-}fst] \qquad \frac{\Gamma, C \vdash e_1 : ({\tau_1}^{\varphi_1}, {\tau_2}^{\varphi_2})^{\varphi}}{\Gamma, C \vdash \mathbf{snd}\ e_1 : {\tau_2}^{\varphi \sqcup \varphi_2}}\ [t\text{-}snd]$$

$$\frac{\Gamma, C \vdash e_1 : (\mathtt{List}\ \tau^{\varphi_1})^{\varphi}}{\Gamma, C \vdash \mathbf{null}\ e_1 : \mathtt{Bool}^{\varphi}}\ [t\text{-}null] \qquad \frac{\Gamma, C \vdash e_1 : (\mathtt{List}\ \tau^{\varphi_1})^{\varphi}}{\Gamma, C \vdash \mathbf{hd}\ e_1 : \tau^{\varphi_1 \sqcup \varphi}}\ [t\text{-}hd] \qquad \frac{\Gamma, C \vdash e_1 : (\mathtt{List}\ \tau^{\varphi_1})^{\varphi}}{\Gamma, C \vdash \mathbf{tl}\ e_1 : (\mathtt{List}\ \tau^{\varphi_1})^{\varphi}}\ [t\text{-}tl]$$

$$\frac{\Gamma, C \vdash e : \tau^{\varphi} \quad C \vdash \varphi_0 \sqsubseteq \varphi}{\Gamma, C \vdash \mathbf{declassify}\ e\ \varphi_0 : \tau^{\varphi_0}}\ [t\text{-}declass] \qquad \frac{\Gamma, C \vdash e : \tau^{\varphi} \quad C \vdash \varphi \sqsubseteq \varphi_0}{\Gamma, C \vdash \mathbf{protect}\ e\ \varphi_0 : \tau^{\varphi_0}}\ [t\text{-}protect]$$

$$\frac{\Gamma, C \vdash e : \tau^{\varphi_1} \quad C \vdash \varphi_1 \sqsubseteq \varphi}{\Gamma, C \vdash e : \tau^{\varphi}}\ [t\text{-}sub]$$

$$\frac{\Gamma, C \vdash e : \sigma^{\varphi} \quad \beta \notin \mathit{fav}(\Gamma) \cup \mathit{fav}(C) \cup \mathit{fav}(\varphi)}{\Gamma, C \vdash e : (\forall \beta.\, \sigma)^{\varphi}}\ [t\text{-}ann\text{-}gen] \qquad \frac{\Gamma, C \vdash e : (\forall \beta.\, \sigma)^{\varphi}}{\Gamma, C \vdash e : ([\beta \mapsto \varphi_1]\ \sigma)^{\varphi}}\ [t\text{-}ann\text{-}ins]$$

$$\frac{\Gamma, C \vdash e : \sigma^{\varphi} \quad \alpha \notin \mathit{ftv}(\Gamma)}{\Gamma, C \vdash e : (\forall \alpha.\, \sigma)^{\varphi}}\ [t\text{-}gen] \qquad \frac{\Gamma, C \vdash e : (\forall \alpha.\, \sigma)^{\varphi}}{\Gamma, C \vdash e : ([\alpha \mapsto \tau]\ \sigma)^{\varphi}}\ [t\text{-}ins]$$

$$\frac{\Gamma, C \cup \{\pi\} \vdash e : \rho^{\varphi}}{\Gamma, C \vdash e : (\pi \Rightarrow \rho)^{\varphi}}\ [t\text{-}qual] \qquad \frac{\Gamma, C \vdash e : (\pi \Rightarrow \rho)^{\varphi} \quad C \vdash \pi}{\Gamma, C \vdash e : \rho^{\varphi}}\ [t\text{-}res]$$

Figure 3. Non-syntax directed rules for security analysis

Typing judgements $\boxed{C \vdash \pi}$

$$\frac{\pi \in C}{C \vdash \pi}\ [c\text{-}in] \qquad \frac{C \vdash \varphi_1 \sqsubseteq \varphi_2 \quad C \vdash \varphi_2 \sqsubseteq \varphi_3}{C \vdash \varphi_1 \sqsubseteq \varphi_3}\ [c\text{-}trans] \qquad \frac{}{C \vdash \varphi \sqsubseteq \varphi}\ [c\text{-}reflex]$$

$$\frac{}{C \vdash \bot \sqsubseteq \varphi}\ [c\text{-}bot] \qquad \frac{}{C \vdash \varphi \sqsubseteq \top}\ [c\text{-}top]$$

Figure 4. Rules for constraints

only an annotated type for the identifier, but also a set of constraints C that constrains the annotation variables within the annotated type (and maybe some others as well). In order to establish over which annotation variables we should universally or existentially quantify, we must first simplify, or solve, C; this explains the use of *simplify* in the definition of *gen* given in Figure 5. The task of *simplify* is to decide whether the constraint set is still consistent, and, if this is the case, remove trivially satisfied constraints, and return a partition (C', C'') of the remaining constraints. The latter contains constraints that involve annotation variables that are free in the environment Γ, while C' contains the remaining constraints. We quantify universally over the type and annotation variables that occur free in τ, and quantify existentially over the remaining free annotation variables (from C'). The constraints from C' can be stored in the type scheme, because they only involve annotation variables that we just quantified over, while the constraints in C'' are returned for further propagation.

$$
\begin{array}{l}
\mathit{gen}\ \Gamma\ \varphi\ \tau\ C = \\
\quad (\forall \alpha_1 \ldots \alpha_n . \forall \beta_1 \ldots \beta_m . \exists \beta_{m+1} \ldots \beta_p . C' \Rightarrow \tau, C'')\ \textbf{where} \\
\qquad (C', C'') = \mathit{simplify}\ C \\
\qquad \{\alpha_1, \ldots, \alpha_n\} = \mathit{ftv}(\tau) - \mathit{ftv}(\Gamma) \\
\qquad \{\beta_1, \ldots, \beta_m\} = \mathit{fav}(\tau) - \mathit{fav}(\Gamma) - \mathit{fav}(\varphi) \\
\qquad \{\beta_{m+1}, \ldots, \beta_p\} = \mathit{fav}(C') - \mathit{fav}(\tau) - \mathit{fav}(\Gamma) - \mathit{fav}(\varphi)
\end{array}
$$

Figure 5. The generalisation function *gen*

Note that the non-syntax-directed rule [*t-sub*] can be handled by generating fresh annotation variables in various places, and relating these by constraints. For example, in the case for [*t-if*] we generate a fresh annotation variable for the annotated type for the conditional, say β, and relate the annotation variable on the then part, say β_1, by the constraint $\beta_1 \sqsubseteq \beta$; the else part and the conditional can be treated similarly.

For reasons of space, we do not provide further details of the syntax-directed type system, the algorithm and the solver in this paper, but refer instead to Appendix C.

4. Heuristics

This section discusses the various heuristics we have developed, organized into four different categories: generic heuristics, propagation heuristics, dependency analysis specific heuristics, and security specific heuristics. We motivate and describe the heuristics themselves, and define the order in which they are applied, and why. In Section 5 we provide additional examples. We note that whatever the heuristics do, they will *never* remove all constraints from the current set. This would imply that no constraint can be blamed for the inconsistency.

We note that having had to refashion some of our code to fit the columns of the paper, the location information may not be correct in all cases.

4.1 Generic heuristics

The heuristics in this section are generally applicable heuristics that have also been employed in other work on heuristics-based type error diagnosis.

Majority heuristic

Johnson and Walz introduced the idea to look at the amount of evidence for a constraint to the source of an inconsistency [14]. We use this idea to point to a possible mistake in a value that is involved in a security error. The majority heuristic retrieves all constraints from an expression that is used as an argument but is considered to be too secure. Then it computes for each security level the number

$$
\begin{array}{l}
\mathit{one} = \textbf{protect}\ 1\ \mathit{Low} \\
\mathit{two} = \textbf{protect}\ 2\ \mathit{Low} \\
\mathit{three} = \textbf{protect}\ 3\ \mathit{Low} \\
\mathit{four} = \textbf{protect}\ 4\ \mathit{Low} \\
\mathit{five} = \textbf{protect}\ 5\ \mathit{High} \\
\mathit{fifteen} = \mathit{print}\ (\mathit{one} + \mathit{two} + \mathit{three} + \mathit{four} + \mathit{five})
\end{array}
$$

Figure 6. Example program: majority of Low values in faulty subexpression

```
Error in application:
"(print ((((one + two) + three) + four) +
    five))" at: (line 6, column 12)
Expected an argument protected at at most
    level: Low
The argument is protected at level: High
Because of the following subexpression(s):
"five" at: (line 6, column 46)
```

Figure 7. Error generated by the majority heuristic in Figure 6

of constraints that imply that the expression should at least have that security level. If the amount of constraints that testify that the expression should have a lower security level is substantially larger than the amount of constraints demanding a higher security level, then the subexpressions where the latter were generated might be the actual cause of the inconsistency. As a result, a mention of those subexpressions will be added to the type error message, stating that they caused the security level to be so high. Note, that we do *not* propose that these expressions are at fault.

In Figure 6, the first five lines declare some values, where the first four are protected at level *Low* and the last value is protected at level *High*. The declaration *fifteen* on the sixth line computes the sum of the five values and passes the result to the *print* function. The latter expects a value that is protected at level *Low*, but the sum of the five values is protected at level *High*. The only value that causes the sum to be protected at level *High* is *five*, all four other values are protected at the level *Low*. The heuristic blames the application of *print*. As there is very little evidence stating that the sum should be protected at level *High* the heuristic also explains what caused the expression to be protected at this level. The programmer can now decide whether the use of *five* in this place was incorrect or whether the use of *print* was at fault.

The least trusted constraint

All constraints are assigned a certain amount of trust based on the AST-node they are generated at. We believe that there is good a reason to have more trust in certain programming constructs than others, because some constructs are more often the cause of an inconsistency or are less intuitive. Constraints that result from instantiating the type of a program variable receive a higher trust value than constraints that were generated locally. The constraints that are generated at these sites belong to the declaration of that particular variable and were found to be consistent when generalising the type of that program variable. Constraints that are generated at application sites receive the least amount of trust, because this construct is considered most likely to introduce inconsistencies. Later in this section, in Figure 16, we discuss an example of a program where the least trusted constraint heuristic accurately blames the application. In Figure 18 the error that is generated is given.

$$
\begin{array}{l}
secureVal :: \texttt{Int}^{High} \\
incr :: \forall\beta_1.\,\forall\beta_2.\,\beta_1 \sqsubseteq \beta_2 \Rightarrow \texttt{Int}^{\beta_1} \rightarrow \texttt{Int}^{\beta_2} \\
id :: \forall\alpha.\,\forall\beta_1.\,\forall\beta_2.\,\beta_1 \sqsubseteq \beta_2 \Rightarrow \alpha^{\beta_1} \rightarrow \alpha^{\beta_2} \\
print :: \forall\alpha.\,\alpha^{Low} \rightarrow \alpha^{Low} \\
print\ (incr\ (incr\ (id\ secureVal)))
\end{array}
$$

Figure 8. Propagating security levels through $incr$ and id

$$
\begin{array}{l}
log = \textbf{fn}\ x \Rightarrow \textbf{protect}\ x\ Medium \\
hVal = \textbf{protect}\ True\ High \\
mVal = \textbf{protect}\ 1\ Medium \\
lVal = \textbf{protect}\ 2\ Low \\
error = log\ (\textbf{if}\ hVal\ \textbf{then}\ lVal\ \textbf{else}\ mVal)
\end{array}
$$

Figure 9. Secure information leaks through conditional

Irrefutable constraints

At some nodes of the abstract syntax tree we generate a constraint for reasons of uniformity with the rest of the type system. We know that such constraints can never be wrong, and therefore should never be blamed. The irrefutable constraints heuristic removes these constraints from the current set of constraints. This is achieved by setting the trust for such constraints to infinity. Our type system, for example, generates a constraint at let bindings stating that the let binding is at least as protected as the body of the let binding. This constraint is, for obvious reasons, always true, and should never be blamed. This heuristic was introduced by Heeren in Section 8.3 of [12].

4.2 The propagation heuristic

Many of the generated constraints only propagate security levels. Consider for example the program in Figure 8. The highly secure $secureVal$ is passed through the identity function once and then twice in succession $incr$ before being passed to the print function. Both functions, $incr$ and id, are polyvariant and propagate the security annotation from their argument to their result. An algorithm will generate explicit constraints to, step-by-step, propagate the security level of $secureVal$ to the $print$ function. Since the $print$ function expects a value of low confidentiality, the program is inconsistent. When we want to assign blame, it makes no sense to blame the application of functions like id and $incr$, because they do not affect the security levels of their argument. What we want then is that the constraints responsible for the propagation of security levels are never blamed for an inconsistency. Therefore we have devised a heuristic that will remove such constraints from the current constraint set. Of course, it may well be possible to replace a function like id by a function that does change the security properties (like **declassify** in this particular case). But since there is no way that we can decide which function should be replaced and with what it should be replaced, this can only serve to confuse the programmer. We will thus have to accept that the sequence of calls to security agnostic functions is correct, which is attained by deleting all propagation constraints from the minimal unsatisfiable constraint set.

Note that the usefulness of this heuristic stems from the polyvariance of the analysis, and is independent of the fact that the underlying language is polymorphic or monomorphic.

4.3 Heuristics for dependency analyses

Security analysis is an instance of a dependency analysis [2], which makes some of the type rules that govern security annotations slightly and subtly different from those of the intrinsic type system that the programmer will most likely be used to. In this section we describe a few heuristics that are constructed with this in mind. For example, the heuristic for the conditional considers that a programmer may believe that the security level of the condition does not contribute to that of the whole conditional, although the type rule in Figure 3 says otherwise. The heuristic tries to discover whether such a misunderstanding explains an inconsistency among the constraints.

We have systematically compared the constraints on security annotations and the constraints on the underlying types in each of the rules of Figure 3. For most of the discrepancies we have found, we have implemented a corresponding heuristic. We shall consider all of these below, but in the interest of conciseness, we only provide details and examples for the heuristic for the conditional.

The first discrepancy is that the security level of an application may be influenced by the security level on the *function itself*. Specifically, in the rule [*t-app*], the φ on the arrow of the type of e_1 contributes to the security level of the application $e_1\ e_2$. In the underlying type system, only the result type of the function type contributes to the type of $e_1\ e_2$. We note that functions are usually created at the lowest security level, but it is still possible to increase the security level by protecting it at a higher level.

The second discrepancy can be found in the rule [*t-hd*]. In the underlying type system, only the element type of the list determines the type of the result of **hd**, but since successfully applying **hd** to a list also provides some information about the structure of its argument list (it is not empty), the security level φ attached to the structure of the list also contributes to the security level of the result of **hd**. We note that in the case of **fst** and **snd** the same reasoning applies, but that the annotations on pairs are there for reasons of uniformity of presentation only. However, it is possible to explicitly increase the security level on a pair by using protect, which will then indeed influence the security level on the result of **fst** and **snd**. It is easy to add such heuristics to our prototype, but currently these have not been implemented.

The final and arguably most striking discrepancy arises for the conditional statement. As explained in the introduction, the outcome of a conditional may reveal information about the value of the condition, and therefore if the condition is highly secure, the result of the conditional will be highly secure, even if the then and else parts themselves are not as secure. This fact can be gleaned from the rule [*t-if*], in which the annotation on e_0 contributes to the security annotation on the conditional, but that only the type τ of the then and else parts determine the type of the conditional.

The heuristic determines whether the security level of the condition is the reason that the whole expression is protected at a level higher than expected. If so it will report this to the programmer and will also explain why the conditional expression has a high security level. In Figure 9 we present a program where secure information is leaked through the conditional. The error presented in Figure 10 is generated by our heuristic. This is an example of a program that uses a lattice consisting of values Low, Medium and High, with the expected relations $Low \sqsubseteq Medium$ and $Medium`sub\ High$. The message describes why the conditional is protected at level $High$, and that this is inconsistent with the security level expected by the function log.

4.4 Security specific heuristics

The sFun++ language has two constructs to explicitly change the security level, **declassify** and **protect**. It is not unlikely that a programmer, at first, will confuse the two. For example, he may try to declassify an expression e by using the **protect** statement,

```
The conditional: hVal of the if statement:
  if hVal then lVal else mVal     at (l 9, c 14)
uses a value: hVal protected at level: High.
This causes the whole if expression:
  if hVal then lVal else mVal     at (l 9, c 14)
to be protected at level: High.
Instead a value protected at level: Medium was
    expected by: log.
```

Figure 10. Error generated by the if-heuristic for the program in Figure 9

$$secureValue = \textbf{protect}\ True\ High$$
$$printSecure = printValue\ (\textbf{protect}\ secureValue\ Low)$$

Figure 11. Example program: misuse of protect statement

```
You try to protect the expression: "protect
    secureValue Low" at (line 3, column 27)
    to level: Low
But the expression you are protecting is
    protected at level: High.
Try declassify to assign the expression the
    level: Low
```

Figure 12. Error generated by sibling heuristic for the program in Figure 11

and provide a security level lower than the level inferred for e, or vice versa. In this section we describe in more detail the situation that a **protect** may need to be replaced by **declassify**. The inverse situation follows dually, and we will not discuss it further. Both have been implemented in the prototype.

In Figure 11 we present an expression that contains a mistake. The value $secureValue$ has type $\texttt{Bool}^{High}$ and the function $printValue$ prints a value that is protected at level Low. The programmer of the program tried to declassify $secureValue$ by protecting it at level Low. Our heuristic will generate the error message presented in Figure 12. The error suggests to replace **protect** with **declassify**, and this indeed results in a security correct program.

It may seem that this heuristic can always be applied whenever we have an inconsistency involving **protect**, but that is not the case: it is essential that the level explicitly provided is *lower* than the inferred level, because that provides the additional hint to the system that **declassify** was intended.

The heuristic requires that there are currently only two constraints in the constraint set, each containing one non-variable annotation. It will then consider the nodes where the constraints originate from, and if this includes a **protect** node, and the explicitly provided level is lower than the inferred security level of the argument to **protect**, then the heuristic will be applicable and generate a suitable error message.

We note that these heuristics are instances of the sibling heuristic introduced by Heeren et al [11].

4.5 How heuristics are applied

In our prototype heuristics are applied in sequence, and in a particular order. One can easily come up with various common-sense reasons why a given heuristic should be tried after another. For example, it makes sense to first filter out irrefutable constraints, and to keep the less-focused heuristics to later. As a rule, we prefer to try the heuristics that pinpoint a particular often-made mistake early on. In general, however, the "best" order very much depends on programmer preference. This is why we made it easy to change the order in which heuristics are considered in our prototype (to be precise, the identifier $heuristics$ defined in module $Heuristics\ /\ Heuristics.hs$). For the examples in this paper, we have used the following ordering that corresponds to the ordering in our downloadable prototype:

1. remove irrefutable constraints,
2. select constraints with heuristics for dependency analyses (if, head, application) (we omitted heuristics for fst and snd, because they are not likely to be useful)
3. remove propagation constraints,
4. security specific heuristics,
5. select constraint based on majority heuristic,
6. select least trusted constraint,
7. pick among the remainder, the constraint that is "earliest" in the program (first come, first blamed)

The motivation for this particular order is as follows: we first delete all irrefutable constraints, because we know that blaming such a constraint will seem very silly. It makes sense to remove propagation constraints at this point, because blaming any of these will not make sense either. In our implementation, however, it is much easier to delete such constraints *after* we know that the dependency-analysis based heuristics are known not to apply. Having removed the propagation constraints, we move on to the last of the specific heuristics, that specifically targets **declassify** or **protect** invocations in the program. The remaining heuristics are more generic, starting with the heuristic that targets specific constraints to blame, followed by weaker heuristics that filter out constraints that are less likely to be to blame.

On the occasion that after applying heuristics 1 through 6 we have not yet found a single cause of the error, it is still possible to present a program slice. Since some of the heuristics will have filtered out various constraints, such a slice is typically much smaller than the original minimal unsatisfiable set of constraints. The program presented in Figure 6 would, for example, result in the slice presented in Figure 13, which indeed only shows those point of the program that contribute to the inconsistency. The explanation of the slice, on the first line, is determined by the kind of AST-node that forms the root of the slice. The endpoints refer to the expected and the provided security levels. Thus when we are not able to find the cause of the inconsistency presenting the program slice may still provide a useful error message. Our prototype implementation cannot present program slices, but it can list all program points that contribute to an inconsistency. We believe that it is possible to construct a program slice from this information. However, because we cannot easily display a type error slice, we resort to a catch-all heuristic (no. 7) that picks from the remaining constraint the one that occurs earliest in the program, and bases the error report on that constraint.

```
Sort of error: use of secure value as less
    secure argument, endpoints Low vs. High
print ((..) + five)
```

Figure 13. Program slice error from the program in Figure 6

5. Further examples

Although we much prefer to follow the approach of Lerner et al [18] to perform our validation, security facilities have not found

their way yet to mainstream functional languages. An exception may be the Jif system [17], but that is a Java based system, a context in which we have no polyvariance and no higher-order functions (see Section 6 for a more detailed discussion). We therefore resort to discussing a number of examples, and variations thereof. Our implementation (see the introduction) provides additional example programs.

In Figure 14 we present a security type correct login program written in sFun++ (similar to the programs presented in Sections A and B). In the program user names and passwords are simple integers, as our language does not support characters and strings. A password file is an (unprotected) list of pairs, consisting of an unprotected user name and a protected password. The function *findUser* retrieves the user name from the list. Because we do not have a proper maybe type, we return either a singleton list when the user is found, or the empty list when the requested user name is not known. The function *login* receives a user name and a password as arguments; it then looks up the user record in the password file. If a user record is found it compares the password inside with the given password. If no user was found it returns *False*. The result of the password comparison is protected at security level *High*, the print function expects a value that is protected no higher than *Low*. It is therefore declassified before it is printed.

In Figure 15 we present a variation of the login function where declassification is forgotten. This means that a highly secure boolean value is passed to the *print* function. The corresponding error message is given in Figure 17. In this case, it is the least trusted constraint heuristic that correctly blames the application.

A mistake that is easily made is that the programmer tries to declassify information through the **protect** statement. In Figure 16 the login function uses **protect** to declassify information. The error provided by our prototype is shown in Figure 18. This time one of the security specific heuristics discovers the mistake, and generates an error message that suggests to replace **protect** by **declassify**.

We now continue with an example of an invocation of **hd** that leads to an inconsistency in Figure 19. In this case, the *log* function expects a value of confidentiality *Low*, but the expression **hd** *zl* makes one of the possible arguments to *log* a value of confidentiality *High*, which can be traced back to the fact that the structure of the list has *High* confidentiality and this is inherited by the value returned by **hd**; that the contents of the list has confidentiality *Low* is not important. The error message that results is can be found in Figure 20. Strikingly, the error messages changes substantially when we change the confidentiality level of the subexpression 1 in the definition of *zl* to *Medium*. This is because we now have two sources for the fact that **hd** *zl* is not of level *Low*. so now the application of *log* to its argument is blamed. Fortunately, the message does still explain that the too high confidentiality arises from *zl*. Note that the applications of *id* are never blamed for anything, and that how *id* is exactly implemented has no bearing on what is derived for it. This is, of course, what we expect.

Another example is the program in Figure 22. Here we have implemented a function and protected the function (and not its return value) at level *High*; the default for functions is *Low*. Because of the rule for application, a return value of this function will be the join of the confidentiality of the body and the function itself, which will in this case always be *High*. This leads to an inconsistency, because *log* expects a value with *Low* confidentiality. The error report is shown in Figure 23.

A final example is the program in Figure 24, where we provide a nested if-statement in which the if in the else part leads to an inconsistency, because the function *fakeId* returns a value of *High* confidentiality. Note that the message in Figure 25 indeed picks out this particular subexpression to blame, but also refers to where the mistake shows up: in the call to *log*.

```
passwordFile =
  Cons (1, protect 31415 High)
    (Cons (2, protect 27182 High) Nil)
findUser =
  fun f user ⇒
    fn l ⇒ if (null l) then Nil
           else
             let r = hd l
             in if ((fst r) ≡ user)
                then Cons r Nil
                else f user (tl l)
login =
  fn u ⇒
    fn p ⇒ print (declassify
      (let userRecord = (findUser u passwordFile)
        in (if (null userRecord)
           then False
           else ((snd (hd userRecord)) ≡ p))) Low)
```

Figure 14. An sFun++ login program

```
login =
  fn u ⇒
    fn p ⇒ print
      (let userRecord = (findUser u passwordFile)
        in (if (null userRecord)
           then False
           else ((snd (hd userRecord)) ≡ p)))
```

Figure 15. The login function without declassification

```
login =
  fn u ⇒
    fn p ⇒ print (protect
      (let userRecord = (findUser u passwordFile)
        in (if (null userRecord)
           then False
           else ((snd (hd userRecord)) ≡ p))) Low)
```

Figure 16. The login function with protection

```
Error in application:
"(print let userRecord = ((findUser u)
    passwordFile) in if null userRecord then
    False else (snd head userRecord == p))" at: (
    line 12, column 25)
The function: "print"
Expected an argument protected at at most level:
    Low
But the argument: "let userRecord ... == p)"
Is protected at a higher level.
```

Figure 17. sFun++ error given for program in Figure 15

```
You try to protect the expression: "let
    userRecord = ((findUser u) passwordFile)
    in if null userRecord then False else (
    snd head userRecord == p)" at: "(line 12,
     column 32) to level: Low
But the expression you are protecting is
    protected at level: High
Try declassify to assign the expression to
    the level: Low
```

Figure 18. sFun++ error given for program in Figure 16

$$
\begin{aligned}
&log = \textbf{fn}\ x \Rightarrow \textbf{protect}\ x\ Low\\
&boolVal = \textbf{protect}\ True\ Low\\
&lVal = \textbf{protect}\ 2\ Low\\
&zl = \textbf{Cons}\ (\textbf{protect}\ 1\ Low)\ (\textbf{protect Nil}\ High)\\
&id = \textbf{fn}\ x \Rightarrow \textbf{let}\ y = x\ \textbf{in}\ y\\
&main = log\ (\textbf{if}\ id\ (id\ boolVal)\ \textbf{then}\ id\ lVal\\
&\quad \textbf{else hd}\ zl)
\end{aligned}
$$

Figure 19. Another example

```
The structure of list: zl at (l 11, c 40)
applied to head in the expression:
  head zl at (l 11, c 37)
is protected at level: High.
This causes the result of the head operation to
    be protected at level: High.
Instead a value protected at level: Low that was
    expected by: log ....
```

Figure 20. sFun++ error given for program in Figure 19

```
Error in application:
(log if (id (id boolVal)) then (id lVal) else
    head zl) at (l 10, c 8)
An argument protected at at most level: Low is
    expected by: log ...
The argument provided:
  if (id (id boolVal)) then (id lVal) else head
     zl at (l 10, c 13)
is protected at level: High
Because of the following sub-expressions: zl at (
    l 11, c 40)
```

Figure 21. sFun++ error given for program in Figure 19

$$
\begin{aligned}
&log = \textbf{fn}\ x \Rightarrow \textbf{protect}\ x\ Low\\
&fLow = \textbf{fn}\ x \Rightarrow \textbf{protect}\ x\ Low\\
&fHigh = \textbf{protect}\ fLow\ High\\
&main = log\ (fLow\ 2 + fHigh\ 3)
\end{aligned}
$$

Figure 22. An example to show the effect of highly confidential functions

```
The function: fHigh at (l 6, c 22)
in the application:
  (fHigh 3) at (l 6, c 22)
is protected at level: High.
This causes the result of the application to be
    protected at level: High.
Instead a value protected at level: Low that was
    expected by: log ....
```

Figure 23. sFun++ error given for the program in Figure 22

$$
\begin{aligned}
&log = \textbf{fn}\ x \Rightarrow \textbf{protect}\ x\ Low\\
&lVal = \textbf{protect}\ 2\ Low\\
&fakeId = \textbf{fn}\ x \Rightarrow \textbf{let}\ y = x\ \textbf{in}\ (\textbf{protect}\ y\ High)\\
&main = log\ (\\
&\quad \textbf{if}\ True\ \textbf{then}\\
&\quad\quad \textbf{if}\ False\ \textbf{then}\ lVal + 2\ \textbf{else}\ 10\\
&\quad \textbf{else if}\ fakeId\ False\ \textbf{then}\ lVal\ \textbf{else}\ lVal + 1)
\end{aligned}
$$

Figure 24. An example with a nested conditional

```
The  conditional: (fakeId False) at (l 8, c 12)
    of the if statement:
if (fakeId False) then lVal else (lVal) + (1) at
    (l 8, c 9)
uses a value: '(fakeId False) at (l 8, c 12)'
    protected at level: High.
This causes the whole if expression:
if (fakeId False) then lVal else (lVal) + (1) at
    (l 8, c 9)
to be protected at level: High.
Instead a value protected at level: Low was
    expected by: log ....
```

Figure 25. sFun++ error given for program in Figure 24

6. Related work

Sabelfeld and Myers provide a comprehensive overview of the field of security analysis [27], in particular the part of the field that derives from the work of Volpano, Smith and others [30, 32, 31]. Although the survey also considers other forms of security hazards, our interest in this paper lies with the issue of program confidentiality: to prevent secure information from leaking (partially) through a non-secure output. In early work on confidentiality all data was labelled with a security level and checked dynamically; we follow the static approach that aims to reject programs at compile time.

Heintze, Nevin and Riecke present a small statically typed, lambda calculus based, language for security analysis called the Slam calculus [13]. The call by value language employs two kinds of security annotations, one for direct readers and one for indirect readers. A variable that has a High secure direct annotation and a Low secure indirect annotation can be used by someone with low permissions to make decisions. In the Slam calculus all values are explicitly annotated with both direct and indirect reader permissions. The type system they present features sub-typing, making the analysis more accurate. There is also a construct **protect** available that allows the programmer to increase the security level of a computed value. An analogue of **declassify** is not available.

In [23, 24] a security analysis for a lightweight version of ML (called Core ML) is presented. This work forms the theoretical basis for the FlowCaml implementation discussed in this paper. The language from the paper also forms the basis for the language used in this paper, although we have omitted some advanced features such as exceptions and references. Core ML does not have the **protect** construct; it is not needed in the present of a subtyping rule. Instead, a higher than necessary security level for a value can be set explicitly by the programmer; if, then, the inferred security level is higher than the explicit annotation, the program will be rejected. The Core ML specification is polyvariant, and the underlying language is polymorphic, as it is in this paper. This allows the programmer to write security agnostic functions.

Security analysis has been shown to be an instance of dependency analysis [2]. This work was later generalized to a polymorphic setting [1]. According to the authors, the three advantages for defining analyses as instances of their Dependency Core Calculus (DCC) are that it allows one to find out in which sense instances of dependency analyses differ, the mapping from dependency analysis to the instance can be used to verify that the expected dependencies do indeed arise, and the encoding of dependency analyses into DCC yield simple proofs of non-interference for these analyses. A fourth advantage may be that heuristics that address the peculiarities of dependency-like analyses may also be transferable to other instances of DCC. Since the other instances of DCC discussed in [2] are optimising analyses, this may not seem to be of any use. However, when programmer provided explicit annotated type signatures may be provided, inconsistencies may again arise.

A lot of work has been done on providing feedback for type errors for polymorphic, higher-order functional languages. The PhD thesis of Heeren [12] contains a comprehensive overview of the field up to 2004. Our work in this paper is mainly based on [8], [28] and [12]. In [12], a framework for type inferencing and type error reporting is presented. Here the Hindley-Milner type system is specified in a constraint-based manner (see also [22, 21]). The way we employ heuristics, and even some of the heuristics themselves are described in [9].

Whereas the previous work intends to provide as precise type errors as possible, [8] take a different approach. When a type error occurs, a program slice is presented to the user. The slice contains all positions in the program that may contribute to the error; positions outside the slice are known not to contribute to the error. Their work is based on an algorithm for finding a minimal, inconsistent constraint set, and based on the associations between constraints and parts of the program, a slice can then be displayed. Recent work shows that the method scales to a full size language [25].

In this paper we combine the two approaches: when we find an inconsistency, we first restrict ourselves to the constraints originating from the associated program slice. Then we apply our heuristics to these constraint to see if they can come up with a more specific error message for the error at hand. Our algorithm for computing the slice is taken from [29].

A third approach is that of the Chameleon Type Debugger [28, 29]. Instead of displaying a type error message, it assists the programmer by starting an interactive type debugging session. During this session the programmer will be asked to supply type information, in order to find out where the source of the inconsistency lies. A distinct advantage of this system is that it becomes much easier to discover mistakes in definitions that were found to be type consistent, but that, as judged from their use later in the program, were found to be at odds with the intentions of the programmer.

The intentions of the work in this paper are most closely related to that of [5] and [17]. Both papers investigate security type error diagnosis, and in contrast to our work, both do so in an imperative setting. In particular, the work of Deng and Smith [5] works for a simple imperative language extended with arrays, and develops a tailormade inference algorithm that additionally provides a recursive trace of the security levels of all the variables involved in the inconsistency. This is precise, but also verbose. The work by King et al. [17] is much more mature, and has been implemented in the Jif information-flow compiler. Their work applies to Java, and by employing an algorithm similar to the algorithm to compute the minimal unsatisfiable subset and using the fact that the analysis is implemented as a dataflow analysis, the system can provide an execution trace leading up to the inconsistency, but restricted to the security aspect only. The analysis they provide is context-insensitive, does not deal with higher-order functions, and also does not seem to be able to deal with parametric polymorphism. A limited form of context-sensitivity is attained by essentially duplicating the analyses of pieces of code for use in a secure and a non-secure; moreover the choice to use either one of these versions has to be made explicitly by the programmer. This decreases the usability of the system and it still remains to be seen how well this extends to larger lattices. Our analysis is context sensitive/polyvariant, and therefore does not suffer from these issues. Note also that one of our heuristics in particular addresses the diagnostic issues arising from being context-sensitive (Section 4.2).

Moreover, there is an ad-hoc extension in order to deal with implicit control-flow, for which they need to introduce additional security variables (which amounts to analysing an SSA form of the program). As a result, the type error diagnosis is not always easy to map back onto the original program. Our work can handle implicit flows, in fact explicit flows and implicit flows cannot be distinguished in a higher-order setting. Moreover, some of our heuristics in particular address issues arising from implicit control-flow (Section 4.3). The authors suggest adding heuristics to their work as future work. This is exactly one of the major contributions of this paper.

A totally different approach to security typing, is to employ an embedded domain-specific language, SecLib [26]. The library uses type classes and monads to enforce security. All security levels and flows are checked by the Haskell type system, illegal flows are reported as regular type errors. All functions have explicit types, as the library heavily relies on type classes for which the programmer has to restrict the instances available. Furthermore the library requires some language extensions to be enabled (all of them are related to type classes). The compiler cannot infer the correct type for all expressions, so we have to help it by providing explicit types. Although beneficial from the viewpoint of language engineering, the fact that we embed the security types into normal types, implies that type errors and security errors cannot easily be distinguished: normal type errors will also reveal security type annotations, and vice versa. Also, one may expect that security errors will only be reported for type correct programs. The work in this paper presupposes working directly on the security language, and not through an embedding in a general purpose language. For that, we suggest the approach of Heeren and Hage [11]. For that to work it needs to be scaled up to full Haskell, which is far from trivial.

7. Conclusion and future work

In this paper we combine a heuristics-based approach to type error diagnosis with a type error slice approach, in order to control and improve security type error diagnosis. These heuristics that work directly on the constraint set that relates the security annotations on expressions and identifiers. Even though the constraints are generated during type inferencing, type and security errors are strictly separated, in that security error messages are only provided if the program is type correct.

In our approach, we (straightforwardly) compute a minimal unsatisfiable set of constraints, that allows us to determine the locations in a program that contribute to an error. Since these slices can be large, we prefer to generate more specific messages by applying a number of heuristics to this minimal unsatisfiable subset. In Section 4 we presented a number of such heuristics, divided into four categories: generic, propagation, dependency analysis specific and security analysis specific heuristics.

There are many clear directions for future work: extension to the source language, in particular explicitly provided security type signatures, and extending the set of heuristics (particularly for newly introduced language constructs).

A different kind of extension is to consider solving constraints beyond a single binding group. For example, when considering constraints for a module as a whole, our majority heuristic can be generalized so that it may suggest to change a definition in order to make the majority of its uses consistent. It should be noted that such an extension is not entirely without danger: somebody may change the properties of a definition on purpose only to find out which parts of the program are affected.

Although we believe we have been able to argue that our heuristics make sense, and we have tried to provide sensible examples of the quality of our messages, a full scale experimental validation still needs to be executed. A problem is the absence of a benchmark collection of security incorrect programs, also because security typing has not made its way yet into the mainstream of (functional) programming. Although our use of the minimal unsatisfiable subset as the baseline for our heuristics, gives a certain guarantee of "soundness", this still does not mean that our messages generally reflect

the diagnosis that an expert may come up with. Finally, in this paper we have committed ourselves to a particular order in which the heuristics are applied. Variations in this order are possible. In our implementation these are very easy to make, but we have not yet investigated the effects on the quality of error messages.

Acknowledgments

This work was supported by the Netherlands Organisation for Scientific Research through its project on "Scriptable Compilers" (612.063.406) and carried out while the third author was employed at Utrecht University. The authors would like to thank Sean Leather and anonymous referees for their helpful comments.

References

[1] M. Abadi. Access control in a core calculus of dependency. *Electron. Notes Theor. Comput. Sci.*, 172:5–31, 2007.

[2] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM.

[4] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1990.

[5] Z. Deng and G. Smith. Type inference and informative error reporting for secure information flow. In *Proceedings of the 44th annual Southeast regional conference*, ACM-SE 44, pages 543–548, New York, NY, USA, 2006. ACM.

[6] GHC Team. *The Glasgow Haskell Compiler*. `http://www.haskell.org/ghc`.

[7] K. Glynn, P. J. Stuckey, and M. Sulzmann. Effective strictness analysis with horn constraints. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 73–92, London, UK, 2001. Springer-Verlag.

[8] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.

[9] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Implementation of Functional Languages – IFL 2006*, volume 4449, pages 199 – 216, Heidelberg, 2007. Springer Verlag.

[10] J. Hage, S. Holdermans, and A. Middelkoop. A generic usage analysis with subeffect qualifiers. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 235–246, New York, NY, USA, 2007. ACM.

[11] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13. ACM Press, 2003.

[12] B. J. Heeren. Top quality type error messages. September 2005.

[13] N. Heintze and J. G. Riecke. The slam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, New York, NY, USA, 1998. ACM.

[14] G. F. Johnson and J. A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *POPL '86: Proceedings of the 13th ACM symposium on Principles of programming languages*, pages 44–57, New York, 1986. ACM.

[15] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, pages 9–9. USENIX Association, 2004.

[16] M. P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1994.

[17] Dave King, Trent Jaeger, Somesh Jha, and Sanjit A. Seshia. Effective blame for information-flow violations. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 250–260, New York, NY, USA, 2008. ACM.

[18] B.S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *ACM SIGPLAN Notices*, volume 42, pages 425–434. ACM, 2007.

[19] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM.

[20] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, 1999.

[21] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[22] F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 389 – 489. MIT Press, 2005.

[23] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, New York, NY, USA, 2002. ACM.

[24] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

[25] V. Rahli, J. B. Wells, and F. Kamareddine. A constraint system for a SML type error slicer. Technical Report HW-MACS-TR-0079, Herriot Watt University, Edinburgh, Scotland, Aug 2010.

[26] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 13–24, New York, NY, USA, 2008. ACM.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[28] P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 72–83, New York, NY, USA, 2003. ACM.

[29] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 80–91, New York, NY, USA, 2004. ACM.

[30] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, 1996.

[31] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–276, New York, NY, USA, 2000. ACM.

[32] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.

[33] J. Weijers. Feedback-oriented security analysis, 2010. `http://www.cs.uu.nl/people/jur/jweijers-msc.pdf`.

```
flow ! everyone < !root
type ('α,'β) pwdEntry =
  {
    userName : 'α string;
    password : 'β string
  }
val pwdList : ((!everyone, !root) pwdEntry, !everyone) list
val checkPwd :
  'α string → (!everyone, !root) pwdEntry → 'α bool
```

Figure 26. Password.fmli

```
type pwdEntry =
  {
    userName : string;
    password : string
  }
let pwdList =
  let jimmy : pwdEntry =
      {userName = "Jimmy"; password = "1234"} in
  ...
  [jimmy; ...]
let checkPwd p uRec =
  if p = uRec.password then true else false
```

Figure 27. Password.ml

```
flow ! everyone < !root
val login : 'α string → 'α string → 'α bool
  with ! everyone < 'α
```

Figure 28. Login.fmli

A. FlowCaml example

We now discuss the implementation that can be found in Figures 26, 27, 28 and 29. Figures 26 and 27 provide the types respectively implementations of the password list information *pwdList* and the login function *checkPwd*. We can conclude from the type of *pwdList* that the *userName* in each password entry is accessible to *everyone*, but that access to the password is restricted to *root*. The function *checkPwd* takes a password, and considers whether it corresponds to the password in the password entry that is its second argument. In the way that this function is used by the *login* function in Figure 29, the first argument will be the password provided by the user (which may have security level *everyone*), but the second argument will contain a password from the *pwdList* that is at level *root*. Security analysis would demand that the output of *checkPwd* have security level *root*, so in order to be able to convey the output of the comparison (a boolean), the result of the comparison must be declassified. This is done within FlowCaml by writing the *checkPwd* function in plain OCaml. Note that the code provided in Figure 29 may look like ordinary OCaml code, but it is however checked not to violate any of the security types provided. This a consequence of the presence of the flow statement at the top of the source file.

Consider now the situation that we had instead provided the type signature

```
'α string → (!everyone, !everyone) pwdEntry → 'α bool
```

```
flow ! everyone < !root
let getUserRecord name =
  let rec lookUp l ls =
    match ls with
      [] → None
    | u :: us → if u.Password.userName = l
                then Some u
                else lookUp l us
  in
    lookUp name Password.pwdList
let login user pwd =
  let userRecord = getUserRecord user
    in match userRecord with
      None → false
    | Some u → Password.checkPwd pwd u
```

Figure 29. Login.fml

```
File "Login.fml", line 11, characters 0-144:
This expression generates the following
    information flow:
root < everyone
which is not legal.
```

Figure 30. Error message for FlowCaml.

for *checkPwd*. Then one would expect the compiler to complain about passing an insecure *pwdEntry* into a function that expects otherwise in the definition of *login* in *Login.fml*.

Instead the compiler returns message in Figure 30 telling us that something invalid was derived.

The message explains there is some illegal flow, and the location points to the line where the *login* function begins. The flow that is found indeed exists within the login function and is illegal, but it is not explained how the flow was derived, which subexpression was responsible, and what the programmer can do to fix the problem. In this particular case it might not be very hard to find the cause of the problem, but in more complicated programs error messages like these are not very helpful.

B. SecLib example

The situation is completely difference for SecLib [26]. For our example we defined a lattice with three security levels (although only one is used explicitly), and we only used the non IO security monad. The library uses type classes and monads to enforce security. All security levels and flows are checked by the Haskell type system, illegal flows are reported as regular type errors.

In the code fragment in Figure 31 the user name lookup and password verification functions are presented. Any declassification is done in a different library where we have access to a special function called *reveal* that removes all security information from a value.

All functions have explicit types, as the library heavily relies on type classes for which the programmer has to restrict the instances available. Furthermore the library requires some language extensions to be enabled (all of them are related to type classes). The compiler cannot infer the correct type for all expressions, so we have to help it by providing explicit types (see the *verifyPassword* function for example in Figure 31). Having to write explicit types at so many places makes writing a program much harder and regular type errors and security errors become harder to read. For example

```
{-# LANGUAGE FlexibleContexts #-}
module Spwd
(
  getSpwdName,
  verifyPassword
)
where
import SecLibTypes
import Sec
import SpwdTypes
import DeclCombinators (hatch, showResult, combine)
import Data.Maybe (fromJust)
verifyPassword :: Less α R ⇒
                  Sec R Spwd → Sec α Pwd → Sec α Bool
verifyPassword record provided =
  let known = fromJust $ ((hatch (λ(_, p) → p) record)
              :: Maybe (Sec R Pwd))
  in fromJust $ showResult $ ((combine (≡) known provided)
              :: (Sec R Bool))
getSpwdName :: Sec α Usr → (Maybe (Sec R Spwd))
getSpwdName nam = findUser nam passWordFile
findUser ::  Sec α Usr → [(Usr, Sec R Spwd)] →
             Maybe (Sec R Spwd)
findUser usr ((u, r) : xs) =
  let match = public $ fromJust $ showResult (matchUser usr u)
  in case match of
     True → Just r
     False → findUser usr xs
findUser _ [] = Nothing
matchUser :: Sec α Usr → Usr → Sec α Bool
matchUser usr u = fromJust $ (hatch ((≡) u) usr)
passWordFile :: [(Usr, Sec R Spwd)]
passWordFile = [("Jimmy", sec ("Jimmy", "1234"))
               ,("Steven", sec ("Steven", "5678"))]
```

Figure 31. Spwd.hs

```
verifyPassword :: Less α R ⇒
                  Sec R Spwd → Sec α Pwd → Sec α Bool
verifyPassword record provided =
  let known = fromJust $ ((hatch (λ(_, p) → p) record)
              :: Maybe (Sec R Pwd))
  in combine (≡) known provided
```

Figure 32. Wrong version of *verifyPassword*

consider the version of *verifyPassword* in Figure 32, where the type signature of *combine* is given as

$$combine :: (Less\ s\ s'', Less\ s'\ s'') \Rightarrow (\alpha \to \beta \to \gamma) \to \mathbf{Sec}\ s\ \alpha \to \mathbf{Sec}\ s'\ \beta \to \mathbf{Sec}\ s''\ \gamma$$

This function is correct when only the types are considered, but inconsistencies arise because instead of the expected **Sec** α `Bool`, the function returns **Sec** R `Bool`. GHC [6] provides the error found in Figure 33. This error provides quite a lot of information, and a large fragment of code. It accurately blames the expression where *combine* is used, but explains the illegal flow of R below α in terms of type classes and not in terms of secure information flow.

Moreover, when we make a mistake that ordinarily results in a normal type error, we get a message that also contains (irrelevant) information about the security annotations. In short, the lack of separation between types and annotations also show up when programmers make mistakes. In this particular program the type errors never really get complicated because of the simple nature of the program. In Figure 34 the result of a small type error is presented. It is obvious that the problem lies in the inferred *Maybe*, but the difference in the security type variable is incidental and not problematic.

```
Spwd.hs:31:19:
    Couldn't match expected type 'Sec a Bool'
                against inferred type 'Maybe (Sec
                    s' Bool)'
    In the expression: (hatch ((==) u) usr)
    In the definition of 'matchUser':
        matchUser usr u = (hatch ((==) u) usr
                          )
```

Figure 34. Error that GHC gives when *fromJust* is forgotten in *matchUser*.

$$(\theta_2.\theta_1)\ (\tau) \equiv \theta_2\ (\theta_1\ (\tau))$$

Figure 35. Substitution composition

Note that the issues that arise here are not restricted to security analysis. Indeed, one of the two major problems of embedding DSLs into general programming languages is that although the programming constructs made available to programmers become domain-specific, the error feedback typically does not follow suit. There has been research to address this problem, but only for Haskell 98 [11].

C. Inference Algorithm

In this chapter an algorithmic version of the inference system from Section 3 is presented. The inferencer is based on Algorithm $\mathcal{W}$ [3]. We will start off by presenting substitutions (Section C.1). We will discuss our unification algorithm in Section C.2. In Section C.3 we present our inference algorithm.

C.1 Substitutions

During type inferencing a variable is introduced whenever a type or annotation cannot be determined, as soon as it can be determined a substitution, represented by θ, is generated that replaces the type or annotation variable by the actual type or annotation.

A substitution is a pair of finite mappings from type variables to types, and annotation variables to annotations. The sets of type and annotation variables are assumed to be disjoint. Substitutions are idempotent, this means that $\theta\ (\theta\ \tau)$ is equal to $\theta\ \tau$. Two substitutions can be combined by substitution composition. Application of a composed substitution is defined in Figure 35.

The empty substitution is represented by Id. Type variables that are not in the domain of the substitution are not changed when applied to a substitution.

Substitutions are applied recursively to a type τ as presented in Figure 36. Application of substitutions on annotation is presented in Figure 37.

Substitutions can be applied to type schemes, and qualified types as well. We define substitutions on type schemes in Figure 38. We introduce a new syntactic category all variables:

$$\zeta \in \mathbf{TyVar} \cup \mathbf{AnnVar} \qquad \textit{all variables}$$

```
Spwd.hs:17:13:
    Could not deduce (Less R a) from the context (Less a R)
      arising from a use of 'combine' at Spwd.hs:17:13-39
    Possible fix:
      add (Less R a) to the context of
        the type signature for 'verifyPassword'
      or add an instance declaration for (Less R a)
    In the expression: combine (==) known provided
    In the expression:
        let
          known = fromJust
                $ ((hatch (\ (_, p) -> p) record)
                  :: Maybe (Sec R Pwd))
        in combine (==) known provided
    In the definition of 'verifyPassword':
        verifyPassword record provided
                       = let
                           known = fromJust
                                 $ ((hatch (\ (_, p) -> ...) record)
                                   :: Maybe (Sec R Pwd))
                         in combine (==) known provided
```

Figure 33. The error that GHC gives when the version of *verifyPassword* from Figure 32 is used.

$$
\begin{array}{ll}
\theta\ \mathtt{Int} & = \mathtt{Int} \\
\theta\ \mathtt{Bool} & = \mathtt{Bool} \\
\theta\ (\tau_1{}^{\varphi_1}, \tau_2{}^{\varphi_2}) & = ((\theta\ \tau_1)^{(\theta\ \varphi_1)}, (\theta\ \tau_2)^{(\theta\ \varphi_2)}) \\
\theta\ (\tau_1{}^{\varphi_1} \to \tau_2{}^{\varphi_2}) & = (\theta\ \tau_1)^{(\theta\ \varphi_1)} \to (\theta\ \tau_2)^{(\theta\ \varphi_2)} \\
\theta\ (\mathtt{List}\tau^{\varphi}) & = \mathtt{List}(\theta\ \tau)^{(\theta\ \varphi)} \\
\theta\ \alpha & = \theta(\alpha)
\end{array}
$$

Figure 36. Substitution application on types

$$
\begin{array}{l}
\theta\ \beta = \theta(\beta) \\
\theta\ l\ = l
\end{array}
$$

Figure 37. Substitution application on annotations

$$
\begin{array}{ll}
\theta\ (\forall\alpha.\sigma) & = \forall\alpha.((\theta/\alpha)\ \sigma) \\
\theta\ (\forall\beta.\sigma) & = \forall\beta.((\theta/\beta)\ \sigma) \\
\theta\ (\exists\beta.\sigma) & = \exists\beta.((\theta/\beta)\ \sigma) \\
\multicolumn{2}{l}{\quad \textbf{where}} \\
\multicolumn{2}{l}{\quad (\theta/\zeta)\ \zeta_1 = \zeta_1 \quad \textbf{if}\ \zeta \equiv \zeta_1} \\
\multicolumn{2}{l}{\qquad\qquad \theta\ \zeta_1\ \textbf{otherwise}}
\end{array}
$$

Figure 38. Substitution application on type schemes

$$\theta\ (C \Rightarrow \tau) = (\theta\ C \Rightarrow \theta\ \tau)$$

Figure 39. Substitution application on qualified types

Bound variables can not be substituted, therefore we introduce a new sort of substitution θ/ζ that prevents the bound variable ζ from being substituted by θ. In Figure 39 we present substitutions on qualified types.

Substitutions can also be applied to type environments (Figure 40). A type environment is defined as a list of mappings from a program variable to a pair of a type and annotation. A substitution updates each of the mappings in the environment, so that all variables map to substituted types and annotations. In Figure 41 we define substitutions on constraints and constraint environments.

$$
\begin{array}{l}
\theta\ \Gamma\ [x \mapsto (\tau, \varphi)] = (\theta\ \Gamma)\ [x \mapsto (\theta\ \tau, \theta\ \varphi)] \\
\theta\ [\,] = [\,]
\end{array}
$$

Figure 40. Substitution application on type environments

$$
\begin{array}{l}
\theta\ C = \{\theta\ \pi \mid \pi\ \in\ C\} \\
\theta\ (\varphi_1 \sqsubseteq \varphi_2) = (\theta\ \varphi_1 \sqsubseteq \theta\ \varphi_2)
\end{array}
$$

Figure 41. Substitution application on constraints and constraint environments

C.2 Unification

During type inferencing two types are often required to be equal. To ensure that two types are equal we use a unification algorithm $\mathcal{U}$. The unification algorithm is defined both for annotated types and annotations, the result of unification is a substitution. In Figure 42 we present unification over types. Unification of two `Int`s or `Bool`s results in the empty substitution Id. Unification over container types (lists, pairs and functions) requires both containers to be equal, and that unification of its element types and annotations succeed. The resulting substitution of a unification is always propagated to the remaining elements before they are unified. The end result of the unification is the composition of the two results. Pairs and functions are unified similarly. Unification of any type with a type variable results in a substitution of that variable with the given type. The type is required not to contain the type variable (unless the type is the same type variable), if the variable does occur in the type unification fails. Unification also fails whenever two different top level type constructors are unified.

In Figure 43 the unification algorithm for security annotations is listed. Only two annotation variables can be unified. Our inferencing algorithm will always assign an annotation variable to an expression, the relation to a security level is expressed by a constraint. Annotations for top level bindings are always security levels. Our var rule introduces a fresh variable and generates a constraint that

$$
\begin{array}{lllll}
\mathit{unifyTy} :: \mathbf{Type} \rightarrow & & \mathbf{Type} \rightarrow \mathbf{Substitution} & & \\
\mathit{unifyTy} & \texttt{Int} & \texttt{Int} & = \mathit{Id} & \\
\mathit{unifyTy} & \texttt{Bool} & \texttt{Bool} & = \mathit{Id} & \\
\mathit{unifyTy} & \texttt{List}\tau_1{}^{\varphi_1} & \texttt{List}\tau_2{}^{\varphi_2} & = & \\
\quad \textbf{let} & \theta_1 = \mathit{unify}\ \tau_1\ \tau_2 & & & \\
& \theta_2 = \mathit{unify}\ (\theta_1\ \varphi_1)\ (\theta_1\ \varphi_2) & & & \\
\quad \textbf{in} & \theta_2.\theta_1 & & & \\
\mathit{unifyTy} & (\tau_{11}{}^{\varphi_{11}}, \tau_{12}{}^{\varphi_{12}})\ (\tau_{21}{}^{\varphi_{21}}, \tau_{22}{}^{\varphi_{22}}) = & & & \\
\quad \textbf{let} & \theta_1 = \mathit{unifyTy}\ \tau_{11}\ \tau_{21} & & & \\
& \theta_2 = \mathit{unifyAnn}\ (\theta_1\ \varphi_{11})\ (\theta_1\ \varphi_{21}) & & & \\
& \theta_3 = \mathit{unifyTy}\ (\theta_2.\theta_1\ \tau_{12})\ (\mathit{apply}\ (\theta_2.\theta_1)\ \tau_{22}) & & & \\
& \theta_4 = \mathit{unifyAnn}\ (\theta_3.\theta_2.\theta_1\ \varphi_{12})\ (\theta_3.\theta_2.\theta_1\ \varphi_{22}) & & & \\
\quad \textbf{in} & \theta_4.\theta_3.\theta_2.\theta_1 & & & \\
\mathit{unifyTy} & \tau_{11}{}^{\varphi_{11}} \rightarrow \tau_{12}{}^{\varphi_{12}}\ \tau_{21}{}^{\varphi_{21}} \rightarrow \tau_{22}{}^{\varphi_{22}} = & & & \\
\quad \textbf{let} & \theta_1 = \mathit{unifyTy}\ \tau_{11}\ \tau_{21} & & & \\
& \theta_2 = \mathit{unifyAnn}\ (\theta_1\ \varphi_{11})\ (\theta_1\ \varphi_{21}) & & & \\
& \theta_3 = \mathit{unifyTy}\ (\theta_2.\theta_1\ \tau_{12})\ (\theta_2.\theta_1\ \tau_{22}) & & & \\
& \theta_4 = \mathit{unifyAnn}\ (\theta_3.\theta_2.\theta_1\ \varphi_{12})\ (\theta_3.\theta_2.\theta_1\ \varphi_{22}) & & & \\
\quad \textbf{in} & \theta_4.\theta_3.\theta_2.\theta_1 & & & \\
\mathit{unifyTy} & \tau & \alpha & = \textbf{if}\ (\tau \equiv \alpha\ \vee\ \alpha \notin (\mathit{ftv}\ \tau)) & \\
& & & \quad \textbf{then}\ [\alpha \mapsto \tau] & \\
& & & \quad \textbf{else}\ \mathit{fail} & \\
\mathit{unifyTy} & \alpha & \tau & = \textbf{if}\ (\tau \equiv \alpha\ \vee\ \alpha \notin (\mathit{ftv}\ \tau)) & \\
& & & \quad \textbf{then}\ [\alpha \mapsto \tau] & \\
& & & \quad \textbf{else}\ \mathit{fail} & \\
\mathit{unifyTy} & \_ & \_ & = \mathit{fail} &
\end{array}
$$

Figure 42. Type Unification

$$
\begin{array}{llll}
\mathit{unifyAnn} :: \mathbf{Annotation} \rightarrow \mathbf{Annotation} \rightarrow \mathbf{Substitution} & & & \\
\mathit{unifyAnn} & \beta_1 & \beta_2 & = [\beta_1 \mapsto \beta_2] \\
\mathit{unifyAnn} & \_ & \_ & = \mathit{fail}
\end{array}
$$

Figure 43. Annotation Unification

restricts the variable to be at least as secure as the security level in the type environment. This ensures that the invariant of always assigning any expression an annotation variable as annotation.

C.3 Security Analysis Inference Algorithm

With the inference system presented in Section 3 as a formal specification we define our inference algorithm $\mathcal{W}_{sec}$. The algorithm infers principal type-schemes, security annotations and constraints on security constraints. The algorithm is based on algorithm $\mathcal{W}$ [3]. The algorithm is defined by structural induction on e and has type:

$$\mathcal{W}_{sec} :: (\mathbf{TyEnv}, \mathbf{Exp}) \rightarrow (\mathbf{Ty}, \mathbf{Ann}, \mathbf{Subst}, \mathbf{Constraints})$$

The algorithm takes a pair of a type environment Γ and an expression e as an argument. The type environment Γ maps program variables to a pair consisting of a type-scheme and an annotation. The result of the algorithm is a quadruple containing a type τ, an annotation φ, a substitution θ and a set of constraints C. Sub-effecting is handled through the constraints and verified by our constraint solver described in Section C.4. Our algorithm is listed in the Figures 44, 45 and 46. In Section 3 we specified the rules of our algorithm; in this section we discuss how we made an algorithm from the specification. Most alternatives follow straightforwardly from the inference rules or from other alternatives. We will therefore only discuss those rules that are special.

The alternatives that type natural numbers and booleans return a type `Int` and `Bool` respectively, a fresh annotation variable, the empty substitution and an empty constraint set. An annotation variable is generated because there is no information that insists on a specific security level at this point. The alternative that types the empty list, returns the type $\texttt{List}\ \alpha^{\beta_1}$. The element type cannot be determined at this point hence the introduction of the type variable. The annotation variable is generated because there is no information at this point that insists on a specific security level.

The alternatives for function abstraction and recursive function abstraction are a bit more involved. The type of the function argument is not known at this point, we therefore type the function body with a type environment extended with the variable associated with a fresh type and annotation variable. During type inference of the body information about the actual type may be discovered, this then results in a substitution. The returned substitution is applied to the domain part of the function type. The returned annotation is a fresh variable as it cannot be determined yet at what level the function should be protected. In recursive functions the type of the recursive call also needs to have a type. The argument of the recursive call is of the same type as the function's argument. The result type and annotation are fresh variables. After inferring the type and annotation of the function body these variables are unified with the type and annotation of the function body.

The alternative for typing variables straightforwardly follows the original inference rule. The type scheme and annotation of the variable is retrieved from in the environment. The type scheme is instantiated as described in Section 3. The resulting qualified type is then divided into a type and a constraint environment. To maintain the invariant of never assigning a security level to an expression we introduce a fresh annotation variable and restrict it to be at least as secure as the security level from the type environment.

The function application alternative is a bit more involved. The type of e_2 has to match the type expected by the function. This type can however not easily be extracted from the inferred type of e_1 and the result type of the whole application can also not easily be extracted from the type of e_1. The types cannot be easily extracted from the type of e_1 because at this point it is not yet known whether the inferred type τ_1 is indeed a function type. We introduce fresh variables for the result type and annotation as well as for the argument annotation (sub-effecting). These variables are combined in a function type together with the inferred type of e_2. This type is then unified with the inferred type of e_1. If unification is successful the type of e_2 matches the expected argument type, and the substitution that is returned by the unification maps the introduced type and annotation variables to the values that were inferred for e_2.

The alternative for let-bindings deserves a bit more attention as this is the place where generalisation takes place. First the type of e_1 is inferred. The result is generalised, as explained in Figure refinfSecInstGen, resulting in a type scheme and a set of constraints. The returned constraints contain annotation variables that occur unbound in the type environment and therefore are passed on. The type scheme and annotation variable are added to the type environment under which e_2 will be typed. The resulting type of the whole let binding is the type of e_2. Its annotation is the fresh variable β which is constrained to be at least as secure as φ_2.

The alternatives for **declassify** $e\ \varphi$ and **protect** $e\ \varphi$ are relatively simple, but crucial for this particular analysis. In both cases, the type of the whole expression is the inferred type for e, and its security level β is at least φ. The security level inferred for e, φ_1, has to be at least as secure as φ in the case of declassify, and φ has to be at least as secure as φ_1 in the case of protect. These requirements are enforced by constraints.

C.4 Constraint Solving

The inference system presented in the previous section generates constraints. Our generalisation algorithm presented in Figure refin-

$\mathcal{W}_{sec}\ (\Gamma, e) = (\tau, \varphi, \theta, C)$

$\mathcal{W}_{sec}\ (\Gamma, n) =$ **let** β *be fresh* **in** $(\texttt{Int}, \beta, \quad id, \emptyset)$
$\mathcal{W}_{sec}\ (\Gamma, \mathit{True}) =$ **let** β *be fresh* **in** $(\texttt{Bool}, \beta, id, \emptyset)$
$\mathcal{W}_{sec}\ (\Gamma, \mathit{False}) =$ **let** β *be fresh* **in** $(\texttt{Bool}, \beta, id, \emptyset)$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{Nil}) =$ **let** α, β, β_1 *be fresh* **in** $(\texttt{List}\ \alpha^{\beta_1}, \beta, id, \emptyset)$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{Cons}\ e_1\ e_2) =$
  let β, β_1 *be fresh*
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ (\theta_1\ \Gamma, e_2)$
    $\theta_3 = \mathcal{U}\ (\tau_2, \theta_2\ (\texttt{List}\ {\tau_1}^{\varphi_1}))$
    $\theta = \theta_3.\theta_2.\theta_1$
  in $(\theta\ (\texttt{List}\ {\tau_1}^{\beta_1}), \beta, \theta, \theta\ (C_1 \cup C_2 \cup \{\varphi_1 \sqsubseteq \beta_1, \varphi_2 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, (e_1, e_2)) =$
  let β *be fresh*
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ (\theta_1\ \Gamma, e_2)$
    $\theta = \theta_2.\theta_1$
  in $(\theta\ ({\tau_1}^{\varphi_1}, {\tau_2}^{\varphi_2}), \beta, \theta, \theta\ (C_1 \cup C_2))$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{fn}\ x \Rightarrow e_0) =$
  let α_x, β_x, β *be fresh*
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma\ [x \mapsto (\alpha_x, \beta_x)], e_0)$
  in $(\theta_1\ ({\alpha_x}^{\beta_x} \to {\tau_1}^{\varphi_1}), \beta, \theta_1, C_1)$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{fun}\ f\ x \Rightarrow e_0) =$
  let $\alpha_x, \alpha_r, \beta_x, \beta_r, \beta$ *be fresh*
    $(\tau_0, \varphi_0, \theta_0, C_0) =$
      $\mathcal{W}_{sec}\ (\Gamma\ [f \mapsto ({\alpha_x}^{\beta_x} \to {\alpha_r}^{\beta_r}, \beta)]\ [x \mapsto (\alpha_x, \beta_x)], e_0)$
    $\theta_1 = \mathcal{U}\ (\tau_0, \theta_0\ \alpha_r)$
    $\theta_2 = \mathcal{U}\ (\theta_1\ \varphi_0, (\theta_1.\theta_0)\ \beta_r)$
    $\theta = \theta_2.\theta_1.\theta_0$
  in $(\theta\ ({\alpha_x}^{\beta_x} \to {\tau_0}^{\varphi_0}), (\theta_2.\theta_1)\ \beta, \theta, C_0)$
$\mathcal{W}_{sec}\ (\Gamma, x) =$
  let β_1 *be fresh*
    $(\sigma, \varphi) = \Gamma\ (x)$
    $C' \Rightarrow \tau = \mathit{inst}\ (\sigma)$
  in $(\tau, \beta_1, id, C' \cup \{\varphi \sqsubseteq \beta_1\})$
$\mathcal{W}_{sec}\ (\Gamma, e_1\ e_2) =$
  let $\alpha_r\ \beta_r\ \beta_x\ \beta$ *be fresh*
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ (\theta_1\ \Gamma, e_2)$
    $\theta_3 = \mathcal{U}\ (\theta_2\ \tau_1, {\tau_2}^{\beta_x} \to {\alpha_r}^{\beta_r})$
    $\theta = \theta_3.\theta_2.\theta_1$
    $C = C_1 \cup C_2 \cup \{\varphi_2 \sqsubseteq \beta_x, \varphi_1 \sqsubseteq \beta, \beta_r \sqsubseteq \beta\}$
  in $(\theta_3\ \alpha_r, \beta, \theta, \theta\ C)$

Figure 44. Type and Security Analysis Inference Algorithm

$\mathcal{W}_{sec}\ (\Gamma, e) = (\tau, \varphi, \theta, C)$

$\mathcal{W}_{sec}\ (\Gamma, \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) =$
  let β *be fresh*
    $(\tau_0, \varphi_0, \theta_0, C_0) = \mathcal{W}_{sec}\ (\Gamma, e_0)$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\theta_0\ \Gamma, e_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ ((\theta_1.\theta_0)\ \Gamma, e_2)$
    $\theta_3 = \mathcal{U}\ ((\theta_2.\theta_1)\ \tau_0, \texttt{Bool})$
    $\theta_4 = \mathcal{U}\ (\theta_3\ \tau_2, (\theta_3.\theta_2)\ \tau_1)$
    $\theta = \theta_4.\theta_3.\theta_2.\theta_1.\theta_0$
    $C = C_0 \cup C_1 \cup C_2 \cup \{\varphi_0 \sqsubseteq \beta, \varphi_1 \sqsubseteq \beta, \varphi_2 \sqsubseteq \beta\}$
  in $(\theta\ \tau_2, \beta, \theta, \theta\ C)$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{let}\ x = e_1\ \textbf{in}\ e_2) =$
  let β *be fresh*
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $(\sigma, C') = \mathit{gen}\ (\theta_1\ \Gamma, \varphi_1, \tau_1, C_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ (\theta_1\ \Gamma\ [x \mapsto (\sigma, \varphi_1)], e_2)$
  in $(\tau_2, \beta, \theta_2.\theta_1, \theta_2\ (C' \cup C_2 \cup \{\varphi_2 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, e_1 \oplus e_2) =$
  let β *be fresh*
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ (\theta_1\ \Gamma, e_2)$
    $\tau^1_\oplus \to \tau^2_\oplus \to \tau_\oplus = \Gamma_\oplus\ (\oplus)$
    $\theta_3 = \mathcal{U}\ (\theta_2\ \tau_1, \tau^1_\oplus)$
    $\theta_4 = \mathcal{U}\ (\theta_3\ \tau_2, \tau^2_\oplus)$
    $\theta = \theta_4.\theta_3.\theta_2.\theta_1$
  in $(\tau_\oplus, \beta, \theta, \theta\ (C_1 \cup C_2 \cup \{\varphi_1 \sqsubseteq \beta, \varphi_2 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, u\ e_1) =$
  let β *be fresh*
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $\tau^1_u \to \tau_u = \Gamma_\oplus\ (u)$
    $\theta_2 = \mathcal{U}\ (\tau_1, \tau^1_u)$
  in $(\tau_u, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{fst}\ e_1) =$
  let $\alpha_1, \beta_1, \alpha_2, \beta_2, \beta$ *be fresh*
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $\theta_2 = \mathcal{U}\ (\tau_1, ({\alpha_1}^{\beta_1}, {\alpha_2}^{\beta_2}))$
  in $(\theta_2\ \alpha_1, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta, \beta_1 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{snd}\ e_1) =$
  let $\alpha_1, \beta_1, \alpha_2, \beta_2, \beta$ *be fresh*
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $\theta_2 = \mathcal{U}\ (\tau_1, ({\alpha_1}^{\beta_1}, {\alpha_2}^{\beta_2}))$
  in $(\theta_2\ \alpha_2, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta, \beta_2 \sqsubseteq \beta\}))$

Figure 45. Type and Security Analysis Inference Algorithm (Continued)

fSecInstGen uses a function *simplify* that simplifies, partitions, and verifies satisfiability of a set of constraints. In this section we will discuss the workings of this function. The simplify function is defined in Figure 47. The function first determines the set of variables that cannot be instantiated. It then computes the range of allowed security levels for each annotation variable. After that it verifies whether the constraint set is satisfiable. The satisfiability check results in a substitution that replaces the variables that may be instantiated with the optimal (lowest possible) security level. Finally the constraint set is partitioned. The constraints are partitioned so that those that restrict the types security annotations can be added to the type as qualifiers. We discuss all these steps below in more detail.

A constraint set is simplified by instantiating as many annotation variables as possible. Instantiation of a variable can limit the use of the expression where it was generated, by instantiating it at a security level higher than strictly necessary, or because it further restricts another annotation variable. Both situations are highly undesirable. Therefore only annotation variables that will not restrict the type that is being generalised will be instantiated, or annotation variables that occur free in the type environment (these represent types that will be generalised over at some other time). We call the annotation variables that occur free in the type τ that is being generalised over and the annotation variables that occur free in the type environment Γ the active variables, i.e. $\mathit{fav}\ (\tau) \cup \mathit{fav}\ (\Gamma)$. This set of active variables is, however, not the complete set of variables that may not be instantiated. Other variables that do not occur in the type, but are the result of the inference algorithm can be restricted by active variables. In the constraint $\beta_1 \sqsubseteq \beta_2$, for example, the variable β_2 is restricted to be at least as secure as β_1. If β_1 is an active variable it will not be instantiated, and thus it is not clear

$$\boxed{\mathcal{W}_{sec}\ (\Gamma, e) = (\tau, \varphi, \theta, C)}$$

$$\begin{array}{l}
\mathcal{W}_{sec}\ (\Gamma, \textbf{null}\ e_1) = \\
\quad \textbf{let}\ \alpha_1, \beta_1, \beta\ \textit{be fresh} \\
\qquad (\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1) \\
\qquad \theta_2 = \mathcal{U}\ (\tau_1, \texttt{List}\ {\alpha_1}^{\beta_1}) \\
\quad \textbf{in}\ (\texttt{Bool}, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta\})) \\
\mathcal{W}_{sec}\ (\Gamma, \textbf{hd}\ e_1) = \\
\quad \textbf{let}\ \alpha_1, \beta_1, \beta\ \textit{be fresh} \\
\qquad (\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1) \\
\qquad \theta_2 = \mathcal{U}\ (\tau_1, \texttt{List}\ {\alpha_1}^{\beta_1}) \\
\quad \textbf{in}\ (\theta_2\ \alpha_1, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta, \beta_1 \sqsubseteq \beta\})) \\
\mathcal{W}_{sec}\ (\Gamma, \textbf{tl}\ e_1) = \\
\quad \textbf{let}\ \alpha_1, \beta_1, \beta\ \textit{be fresh} \\
\qquad (\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1) \\
\qquad \theta_2 = \mathcal{U}\ (\tau_1, \texttt{List}\ {\alpha_1}^{\beta_1}) \\
\quad \textbf{in}\ (\theta_2\ \tau_1, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta\})) \\
\mathcal{W}_{sec}\ (\Gamma, \textbf{declassify}\ e\ \varphi) = \\
\quad \textbf{let}\ \beta\ \textit{be fresh} \\
\qquad (\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e) \\
\quad \textbf{in}\ (\tau_1, \beta, \theta_1, C_1 \cup \{\varphi \sqsubseteq \varphi_1, \varphi \sqsubseteq \beta\}) \\
\mathcal{W}_{sec}\ (\Gamma, \textbf{protect}\ e\ \varphi) = \\
\quad \textbf{let}\ \beta\ \textit{be fresh} \\
\qquad (\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e) \\
\quad \textbf{in}\ (\tau_1, \beta, \theta_1, C_1 \cup \{\varphi_1 \sqsubseteq \varphi, \varphi \sqsubseteq \beta\})
\end{array}$$

Figure 46. Type and Security Analysis Inference Algorithm (Continued)

$$\begin{array}{l}
\mathit{simplify}(\Gamma, \varphi, \tau, C) = \textbf{do} \\
\quad \mathit{active} = \mathit{pseudoActive}\ (\mathit{fav}\ (\Gamma) \cup \mathit{fav}\ (\textbf{Ty}), C) \\
\quad \mathit{ranges} = \textbf{range}\ C \\
\quad \theta = \text{satisfiable}\ (\mathit{active}, \mathit{ranges}) \\
\quad c' = \theta\ C \\
\quad \textbf{return}\ \mathit{partition}(\Gamma, c')
\end{array}$$

Figure 47. Simplification algorithm

$$\begin{array}{l}
\mathit{pseudoActive}\ (\mathit{active}, C) = \textbf{do} \\
\quad \mathit{vars} = \{\, u \mid (l \sqsubseteq u) \in C \wedge l \in \mathit{active} \,\} \cup \mathit{active} \\
\quad \textbf{if}\ ((\mathit{vars} \cap \mathit{active}) \neq \emptyset)\ \textbf{then} \\
\qquad \mathit{pseudoActive}\ (\mathit{vars}, C) \\
\quad \textbf{else} \\
\qquad \textbf{return}\ \mathit{vars}
\end{array}$$

Figure 48. Determine set of pseudo active variables

what the lowest possible security level for β_2 is. The lowest possible security level will restrict the use of an expression the least, we therefore choose to delay instantiation of β_2 until β_1 is known. Variables such as β_2 are known as *pseudo active variables*. In Figure 48 we present a function that computes the set of all pseudo active variables, given the set of all active variables and a set of constraints.

All variables that are not (pseudo-) active can be instantiated, if there exists a non empty range of security levels for every variable. If, for any variable, it is not possible to choose a value such that all constraints are satisfied the program contains an error. In that case the compiler will explain where the problem arises and what caused it. Error explanation is discussed in Section 4. The range of allowed values for each variable is computed by a work list algorithm. We present the algorithm in Figure 49. The algorithm is based on the work list algorithm presented in Chapter 3 of [20]. The algorithm first initialises the work list $worklist$, the result of the algorithm $analysis$, and two edge arrays ub and lb. We use two instead of one edge arrays as we compute both a lower bound and an upper bound for each variable. The array ub is used to propagate new values for the upper bound downward. The set $ub\ [\beta]$ contains all constraints where β is the upper bound. The lb array does exactly the opposite. The result for all variables that occur in the constraint set C is initialised with the range $(\bot, \top)$. After initialisation the algorithm will compute the actual ranges by taking constraints from the work list and update the bounds of variables accordingly. A variable β that had its lower bound changed will add all constraints in the set $lb\ [\beta]$ to the work list. A variable β that had its upper bound changed adds all constraints from the set $ub\ [\beta]$ to the work list. In the algorithm we use the following notational conventions: $\lfloor analysis\ [\beta] \rfloor$ denotes the current lower bound of the range associated with β, and $\lceil analysis\ [\beta] \rceil$ returns the upper bound of the range associated with β. The least upper bound operator is written as $\sqcup$ and greatest lower bound operator as $\sqcap$.

The outcome of the work list algorithm is a set of mappings from an annotation variable to a range of security levels. The function satisfiable (presented in Figure 50) processes the results of the work list algorithm. If all variables have a non empty range it will result in a substitution that replaces all non (pseudo-) active variable by the lower bound of the range. If any variable has an empty range an error has been found and the simplification and inference process stop.

Partitioning the constraint set is the final step of the simplification process. The partition of the constraint set consists of two sets. The first set contains constraints that contain at least one variable that is active in Γ. This set of constraints is propagated further by the inference algorithm. It contains information about annotation variables that are later generalised over. The other set contains all other constraints. These constraints will become predicates in the type that is generalised over. We present the partition algorithm in Figure 51.

In our security type system we employ qualifiers for storing constraints in types. The theory of qualified types was developed by [16]. Another more well-known example of qualified types are the type classes of Haskell, where qualifiers can be used to impose restrictions on let-polymorphic types, e.g., the equality operator has the type $\forall\alpha.Eq\ \alpha \Rightarrow \alpha \to \alpha \to \textbf{bool}$, i.e., it is only well-typed for those types α that are instances of the Eq type class. In this paper, the predicates we store in types are only meant to relate the security annotations of the various parts that make up a type. In [10], the same technique was used to exploit qualified types to model polyvariance, but then in the context of usage analysis.

```
range(C) = do
  worklist  = { }
  analysis  = [ ]
  ub        = [ ]
  lb        = [ ]
  for all π in C do worklist := worklist ∪ {π}
  for all β in fav(C) do
    analysis = analysis[β ↦ (⊥, ⊤)]
    ub       = ub[β ↦ { }]
    lb       = lb[β ↦ { }]
  for all π@(β1, β2) in C do
    ub[β2] = ub[β2] ∪ {π}
    lb[β1] = lb[β1] ∪ {π}
  while worklist ≠ { } do
    let C1 ∪ {π} = worklist
    worklist = C1
    case C of
      (β1 ⊑ β2) ⇒
        if (analysis[β1] ≠ analysis[β1] ⊓ ⌈analysis [β2]⌉) then
          worklist = worklist ∪ ub[β1]
          analysis[β1] = analysis[β1] ⊓ ⌈analysis [β2]⌉
        if (analysis[β2] ≠ analysis[β2] ⊔ ⌊analysis [β1]⌋) then
          worklist = worklist ∪ lb[β2]
          analysis[β2] = analysis[β2] ⊔ ⌊analysis [β1]⌋
      (φ ⊑ β)  ⇒
        if (analysis[β] ≠ analysis[β] ⊔ φ) then
          worklist = worklist ∪ lb[β]
          analysis[β] = analysis[β] ⊔ φ
      (β ⊑ φ)  ⇒
        if (analysis[β] ≠ analysis[β] ⊓ φ) then
          worklist = worklist ∪ ub[β]
          analysis[β] = analysis[β] ⊓ φ
  return analysis
```

Figure 49. Worklist algorithm for range computation

```
satisfiable(active, analysis) = do
  substitution = Id
  for all [β ↦ (l, u)] in analysis do
    if (l ⊑ u) then
      if (β ∉ active ∨ l ≡ u) then
        substitution = [β ↦ l].substitution
      else skip
    else error
  return substitution
```

Figure 50. Checking satisfiability

```
partition (Γ, C) = do
  active = fav(Γ)
  prop   = {(l ⊑ u) ∈ C | (l ∈ active ∨ u ∈ active)}
  qual   = {π ∈ C | π ∉ prop}
  return (prop, qual)
```

Figure 51. Partitioning of constraints

A Type- and Control-Flow Analysis for System F

Matthew Fluet

Computer Science Department
Rochester Institute of Technology, Rochester NY 14623
mtf@cs.rit.edu

Abstract. We present a monovariant flow analysis for System F (with recursion). The flow analysis yields both *control-flow* information, approximating the λ- and Λ-expressions that may be bound to variables, and *type-flow* information, approximating the type expressions that may instantiate type variables. Moreover, the two flows are mutually beneficial: the control flow determines which Λ-expressions may be applied to which type expressions (and, hence, which type expressions may instantiate which type variables), while the type flow filters the λ- and Λ-expressions that may be bound to variables (by keeping only those expressions with a static type that is compatible with the static type of the variable with respect to the type flow). As is typical for a monovariant control-flow analysis, control-flow information is expressed as an abstract environment mapping variables to sets of (syntactic) λ- and Λ-expressions that appear in the program under analysis. Similarly, type-flow information is expressed as an abstract environment mapping type variables to sets of (syntactic) types that appear in the program under analysis. Compatibility of static types (with free type variables) with respect to a type flow is decided by interpreting the abstract environment as productions for a *regular-tree grammar* and asking if the languages generated by taking the types in question as starting terms have a non-empty intersection.

1 Introduction

Control-flow analysis is an important enabling technology for the compilation and optimization of functional languages. Because functional languages have first-class functions, the control flow of a functional program is not syntactically apparent: in an application expression, the function can itself be the result of a computation and may not be available until run time. Indeed, during the execution of a program, many different functions may be applied at the same (source-program) application expression. A control-flow analysis [21,44,43,32,27] approximates, at compile time, the flow of first-class functions in a program: which first-class functions might be bound to a given variable or returned by a given expression at run time. This approximate control-flow information can be used to enable optimizations of a functional language.

Control-flow analyses are typically formulated for *dynamically-* or *simply-*typed functional languages.[1] However, most statically-typed functional languages have rich type systems that include polymorphic types. Indeed, System F [13,38], the polymorphic lambda calculus, and extensions thereof are commonly used as typed intermediate languages in compilers for functional languages [47,34,45]. Typed intermediate languages provide a number of benefits. First, explicit type information can support type-dependent optimizations, such as using a specialized representation for known types rather than a universal representation. Second, explicit type information can support validation of optimizations, by detecting when an optimization transforms a well-typed program to an ill-typed program. Since optimizations are performed on a typed intermediate language and optimizations are enabled by control-flow analyses, it is natural to seek a control-flow analysis that is formulated for System F.

While one could naïvely adopt an existing control-flow analysis that is formulated for a dynamically- or simply-typed functional language and ignore the System F features of type abstraction and type application, such an approach fails to take advantage of the static information provided by a well-typed program. Intuitively, a control-flow analysis for System F should exploit the well-typedness of the program under analysis in order to obtain more precise control-flow information. For instance, if a control-flow analysis asserts that a variable x might be bound to a function of type $\mathsf{int} \to \mathsf{int}$, a function of type $\mathsf{bool} \to \mathsf{bool}$, and a function of type $\mathsf{string} \to \mathsf{string}$ (and no other functions), but the static type of x is $\mathsf{int} \to \mathsf{int}$, then the type soundness of the language guarantees that x will only be bound to functions of type $\mathsf{int} \to \mathsf{int}$ at run time and the control-flow result may be soundly refined to assert that x might be bound to only the function of type $\mathsf{int} \to \mathsf{int}$. However, if the static type of x is $\alpha \to \alpha$ (where the type variable α is bound by a type abstraction in the program under analysis), then it is unclear whether or not the control-flow result may be soundly refined, because the type variable α may be soundly instantiated at any type.

Given additional information that asserts that the type variable α might be instantiated at the type int and the type bool (and no other types), then the control-flow result may be soundly refined to assert that x might be bound to only the function of type $\mathsf{int} \to \mathsf{int}$ and the function of type $\mathsf{bool} \to \mathsf{bool}$. This additional information may be obtained by a *type-flow analysis* that approximates, at compile time, the flow of types in a program: which types might instantiate a given type variable at run time. As demonstrated by the example above, this approximate type-flow information can be used to increase the precision of a control-flow analysis. Furthermore, this approximate type-flow information can be used to enable type-dependent optimization, such as guiding the specialization of a polymorphic function that is used at a small number of distinct types or

[1] Although there are many *type-based* [33] control-flow analyses, whereby the analyses themselves are expressed as a sophisticated type systems (e.g., type-and-effect systems [9,31,19], type systems with polymorphic types [36], type systems with union/intersection types [49,29]), the language under analysis is typically a simply-typed language.

eliminating type operations in a language with intensional polymorphism [14]. Just as a control-flow analysis yields useful information because, for a given program, it is unlikely that a given variable is bound to *every* function during execution, a type-flow analysis yields useful information because it is unlikely a given type variable is instantiated at that *every* type during execution.

Although type-flow information and control-flow information might be obtained by independent analyses, the two kinds of information can be mutually beneficial, particularly for the higher-rank impredicative polymorphism of System F. Control-flow analysis supports type-flow analysis by yielding information about the type abstractions which may be applied at type applications and, hence, about the types at which type variables may be instantiated. The type-flow information soundly refines the control-flow information by rejecting flows that are incompatible with the static typing of the program under analysis; because the static typing may be expressed in terms of syntactic types with free type variables, the type-flow information is used to determine the compatibility of types. When the type-flow information refines the control-flow information by rejecting the flow of a type abstraction, the type-flow information itself may be refined because the type abstraction may be applied at fewer type applications, and, hence, there may be fewer types at which the type variable may be instantiated.[2]

In a combined type- and control-flow analysis, the type-flow information soundly refines the control-flow information by determining when types are compatible. In the presence of recursion and higher-rank impredicative polymorphism, the type-flow information must approximate complex relationships between type variables and types and the compatibility or incompatibility of types with respect to the type-flow information may not be obvious. Indeed, during the execution of a program that is well-typed in System F (with recursion), a type variable may be instantiated at an infinite number of types. In order to obtain a computable analysis, the type-flow information must use a finite representation that approximates the (potentially infinite) set of ground types that may instantiate a type variable and the compatibility of types with respect to the type-flow information must be a decidable property.

Most control-flow analyses approximate the (potentially infinite) set of first-class functions that might be bound to a variable at run time by a (necessarily finite) set of function expressions (possibly with free variables) that appear in the program under analysis. Similarly, a type-flow analysis may approximate the (potentially infinite) set of ground types that may instantiate a type variable at run time by a (necessarily finite) set of type expressions (possibly with free type variables) that appear in the program under analysis. For instance, if a type-flow analysis asserts that a type variable α might be instantiated at the type expression $\mathsf{int} \to \mathsf{int}$ and the type expression $\mathsf{int} \to \alpha$ (and no other type expressions), then, by interpreting the type-flow information as productions for a

[2] In practice, though, a flow analysis is computed by adding information that is consistent with existing information (i.e., ascending a lattice) rather than removing information that is inconsistent with existing information (i.e., descending a lattice).

regular-tree grammar [12,2,7], the type-flow analysis may be seen to be asserting that the type variable α might be instantiated at the infinite set of ground types: $\{\mathsf{int} \to \mathsf{int},\ \mathsf{int} \to \mathsf{int} \to \mathsf{int},\ \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int},\ \ldots\}$. Furthermore, if the type-flow analysis asserts that a type variable β might be instantiated at the type expression $\mathsf{int} \to \mathsf{bool}$ and the type expression $\mathsf{int} \to \beta$ (and no other type expressions) and a control-flow analysis asserts that a variable x might be bound to a function of type $\mathsf{int} \to \mathsf{int}$, a function of type $\mathsf{bool} \to \mathsf{int}$, a function of type $\mathsf{string} \to \mathsf{int}$, a function of type $\mathsf{int} \to \alpha$, and a function of type $\mathsf{int} \to \beta$ (and no other functions), but the static type of x is α, then the control-flow result may be soundly refined to assert that x might be bound to only the function of type $\mathsf{int} \to \mathsf{int}$ and the function of type $\mathsf{int} \to \alpha$, because the types of these two functions are compatible with the type α (with respect to the type-flow information), while the types of the other three functions are incompatible with the type α.

Two types are compatible (with respect to the type-flow information) if there exists a ground type that is a member of the sets of ground types at which the types might be instantiated; conversely, two types are incompatible if there does not exist a ground type that is a member of the sets of ground types at which the types might be instantiated. The type soundness of the language guarantees that a variable will only be bound to a well-typed closed function of a ground type at run time; hence, if there is no ground type at which both the static type of a variable and the static type of a function might be instantiated, then that variable will never be bound to that function at run time. For example, the types $\mathsf{int} \to \alpha$ and α are compatible because the type $\mathsf{int} \to \alpha$, interpreted as a starting term for the regular-tree grammar corresponding to the type-flow information, represents the infinite set of ground types $\{\mathsf{int} \to \mathsf{int} \to \mathsf{int}, \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \ldots\}$, which has a non-empty intersection with the infinite set of ground types that might instantiate the type variable α (given above). Similarly, the types $\mathsf{int} \to \beta$ and α are incompatible because the type $\mathsf{int} \to \beta$ represents the infinite set of ground types $\{\mathsf{int} \to \mathsf{int} \to \mathsf{bool}, \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{bool}, \ldots\}$, which has an empty intersection with the infinite set of ground types that might instantiate the type variable α. Since regular-tree grammars are closed under intersection and the emptiness of a regular-tree grammar is decidable, the compatibility of types with respect to the type-flow information is a decidable property.

Overview

We present a monovariant[3] type- and control-flow analysis for System F extended with recursive functions. Our flow analysis is a variation on 0CFA, the classic monovariant control-flow analysis [32]. For a given program, the flow analysis computes an abstract environment that maps variables to (finite) sets of λ- and Λ-expressions that appear in the program and maps type variables to (finite) sets of type expressions that appear in the program.

[3] i.e., context insensitive

Soundness of the analysis is proven with respect to an operational semantics for System F given in the style of the administrative-normal-form (ANF) environment- and continuation-based C_aEK abstract machine [10], where the (concrete) environment component of the abstract machine maps variables to closures (pairs of λ- or Λ-expressions and an environment, which captures the free variables and type variables of the λ- or Λ-expression) and maps type variables to *type closures* (pairs of type expressions and an environment, which captures the free type variables of the type expression). A sound flow analysis computes an abstract environment that approximates every concrete environment that arises during evaluation. To simplify the soundness proof, the machine transition rules are instrumented with explicit type-equality conditions; these conditions are necessarily satisfied by a well-typed program, but are the concrete analogue of the type-compatibility conditions used in the flow analysis. We present both the run-time type-equality and the analysis-time type-compatibility predicates as judgements; this yields a declarative specification of type compatibility, for which the regular-tree-grammar interpretation gives an algorithmic implementation.

Our formulation of the type- and control-flow analysis as a refinement of the syntax-directed constraint-based formulation of 0CFA establishes that the combined type- and control-flow analysis can be more precise than 0CFA. Although not as precise as a type-directed polyvariant[4] control-flow analysis [20], our monovariant type- and control-flow analysis nonetheless rejects some similar classes of spurious flows and furthermore has the benefits of handling full (i.e., impredicative) System F and terminating for all well-typed programs.

2 Language and Semantics

Our source language is a variant of System F, extended with recursive functions and presented in (a restriction of) administrative normal form (ANF). The static semantics of the language is entirely standard, but given for completeness. The operational semantics of the language is presented as an abstract machine, but deviates in some ways from a straightforward adaptation of the environment- and continuation-based C_aEK abstract machine to an ANF variant of System F.

2.1 Syntax

The syntax of our ANF variant of System F is given in Figure 1.

Types include function types, type variables, and universal types; in the universal type $\forall\alpha.\ \tau$, the type variable α is bound in the type τ. Type equality is syntactic identity (up to α-conversion of bound type variables).

Expressions include variables, `let`-bindings of values, `let`-bindings of non-tail function applications, and `let`-bindings of non-tail type applications; in the `let`-binding expressions $\texttt{let}\ x{:}\tau_x\ \texttt{=}\ \cdots\ \texttt{in}\ e$, the variable x is bound in e. Values include recursive functions and recursive type abstractions; in the recursive

[4] i.e., context sensitive

$$
\begin{array}{lrcl}
\textit{Types} & \textit{Type} \ni \tau & ::= & \tau \to \tau \mid \alpha \mid \forall\alpha.\ \tau \\
\textit{Type variables} & \textit{TyVar} \ni \alpha, \beta & & \\
\\
\textit{Expressions} & \textit{Exp} \ni e & ::= & x \mid \texttt{let}\ x{:}\tau\ \texttt{=}\ v\ \texttt{in}\ e \mid \\
 & & & \quad \texttt{let}\ x{:}\tau\ \texttt{=}\ x\ x\ \texttt{in}\ e \mid \\
 & & & \quad \texttt{let}\ x{:}\tau\ \texttt{=}\ x\ \texttt{[}\tau\texttt{]}\ \texttt{in}\ e \\
\textit{Values} & \textit{Value} \ni v & ::= & \mu x{:}\tau.\lambda x{:}\tau.e \mid \mu x{:}\tau.\Lambda\alpha.e \\
\textit{Variables} & \textit{Var} \ni x, y, z, f, g & & \\
\\
\textit{Programs} & \textit{Prog} \ni P & ::= & e
\end{array}
$$

Fig. 1. Syntax of ANF System F

function $\mu f{:}\tau_f.\lambda x{:}\tau_x.e_b$, the variables f and x are bound in the expression e_b and in the recursive type abstraction $\mu f{:}\tau_f.\Lambda\alpha.e_b$, the variable f and the type variable α are bound in the expression e_b. Programs are (closed, well-typed) expressions.

The language is Church-style, in which every bound variable is annotated with its type. In contrast to some presentations of ANF-like languages [39,8,10] but in concert with some others [46,48,5], we restrict the constituents of function applications and type applications to variables, rather than allowing a larger class of "trivial" expressions, and we restrict function applications and type applications to non-tail calls, rather than allowing tail calls. Neither of these restrictions is essential for the forthcoming type- and control-flow analysis; we adopt them simply to reduce the number of inference rules and helper functions in the static semantics, operational semantics, and flow analysis.

We let TyVar_P be the (finite) set of Λ-bound type variables, Var_P be the (finite) set of `let`-, μ-, and λ-bound variables, Type_P be the (finite) set of types, and Value_P be the (finite) set of values that occur in a given program P; distinguishing syntactically identical sub-terms can be defined formally using paths or unique labellings.

Finally, we define a function $\mathsf{last}(\cdot)$ on expressions, which returns the variable that yields the expression's value:

$$
\begin{aligned}
\mathsf{last}(\cdot) &:: \mathit{Exp} \to \mathit{Var} \\
\mathsf{last}(x) &= x \\
\mathsf{last}(\texttt{let}\ x{:}\tau_x\ \texttt{=}\ v\ \texttt{in}\ e) &= \mathsf{last}(e) \\
\mathsf{last}(\texttt{let}\ x{:}\tau_x\ \texttt{=}\ x_f\ x_a\ \texttt{in}\ e) &= \mathsf{last}(e) \\
\mathsf{last}(\texttt{let}\ x{:}\tau_x\ \texttt{=}\ x_f\ \texttt{[}\tau_a\texttt{]}\ \texttt{in}\ e) &= \mathsf{last}(e)
\end{aligned}
$$

2.2 Static Semantics

The standard static semantics for System F, adapted to our ANF variant, are given in Figure 2. A type-variable context Δ records free type variables and

$$\begin{array}{rr} \textit{Type-variable contexts} & \mathit{TCtxt} \ni \Delta ::= \emptyset \mid \Delta, \alpha{:}\star \\ \textit{Variable contexts} & \mathit{Ctxt} \ni \Gamma ::= \emptyset \mid \Gamma, x{:}\tau \end{array}$$

$\boxed{\Delta \vdash \tau}$

$$\frac{\Delta \vdash \tau_a \qquad \Delta \vdash \tau_b}{\Delta \vdash \tau_a \to \tau_b} \qquad \frac{\Delta(\alpha) = \star}{\Delta \vdash \alpha} \qquad \frac{\Delta, \alpha{:}\star \vdash \tau_b}{\Delta \vdash \forall \alpha.\ \tau_b}$$

$\boxed{\Delta; \Gamma \vdash v : \tau}$

$$\frac{\Delta \vdash \tau_f \qquad \Delta \vdash \tau_z \qquad \Delta; \Gamma, f{:}\tau_f, z{:}\tau_z \vdash e_b : \tau_b \qquad \tau_f = \tau_a \to \tau_b \qquad \tau_z = \tau_a}{\Delta; \Gamma \vdash \mu f{:}\tau_f\,.\,\lambda z{:}\tau_z\,.\,e_b : \tau_a \to \tau_b} \qquad \frac{\Delta \vdash \tau_f \qquad \Delta, \beta{:}\star; \Gamma, f{:}\tau_f \vdash e_b : \tau_b \qquad \tau_f = \forall \beta.\ \tau_b}{\Delta; \Gamma \vdash \mu f{:}\tau_f\,.\,\Lambda \beta\,.\,e_b : \forall \beta.\ \tau_b}$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{\Gamma(x) = \tau_x}{\Delta; \Gamma \vdash x : \tau_x} \qquad \frac{\Delta; \Gamma \vdash v : \tau_v \qquad \Delta \vdash \tau_x \qquad \tau_x = \tau_v \qquad \Delta; \Gamma, x{:}\tau_x \vdash e : \tau}{\Delta; \Gamma \vdash \texttt{let}\ x{:}\tau_x\ \texttt{=}\ v\ \texttt{in}\ e : \tau}$$

$$\frac{\Gamma(x_f) = \tau_a \to \tau_b \qquad \Gamma(x_a) = \tau_a \qquad \Delta \vdash \tau_x \qquad \tau_x = \tau_b \qquad \Delta; \Gamma, x{:}\tau_x \vdash e : \tau}{\Delta; \Gamma \vdash \texttt{let}\ x{:}\tau_x\ \texttt{=}\ x_f\ x_a\ \texttt{in}\ e : \tau}$$

$$\frac{\Gamma(x_f) = \forall \beta.\ \tau_b \qquad \Delta \vdash \tau_a \qquad \Delta \vdash \tau_x \qquad \tau_x = [\beta \rightarrowtail \tau_a](\tau_b) \qquad \Delta; \Gamma, x{:}\tau_x \vdash e : \tau}{\Gamma \vdash \texttt{let}\ x{:}\tau_x\ \texttt{=}\ x_f\ \texttt{[}\tau_a\texttt{]}\ \texttt{in}\ e : \tau}$$

$\boxed{\vdash P : \tau}$

$$\frac{\emptyset; \emptyset \vdash e : \tau}{\vdash e : \tau}$$

Fig. 2. Static Semantics of ANF System F

$$
\begin{array}{ll}
\textit{Run-time types} & \textit{RType} \ni \pi ::= \langle \tau; \phi \rangle \\
\textit{Run-time type environments} & \textit{RTEnv} \ni \phi ::= \emptyset \mid \phi, \alpha \mapsto \pi \\
\\
\textit{Run-time values} & \textit{RValue} \ni w ::= \langle v; \phi; \rho \rangle \\
\textit{Run-time value environments} & \textit{REnv} \ni \rho ::= \emptyset \mid \rho, x \mapsto w \\
\\
\textit{Continuations} & \textit{Kont} \ni \kappa ::= \bullet \mid \langle x; \tau; e; \phi; \rho \rangle :: \kappa \\
\textit{States} & \textit{State} \ni \varsigma ::= \langle e; \phi; \rho; \kappa \rangle
\end{array}
$$

$\boxed{\varsigma \longrightarrow \varsigma}$

$$
\frac{\rho_r(x_r) = w_r \qquad \vdash w_r :\equiv \langle \tau_z; \phi \rangle}{\langle x_r; \phi_r; \rho_r; \langle z; \tau_z; e; \phi; \rho \rangle :: \kappa \rangle \longrightarrow \langle e; \phi; \rho, z \mapsto w_r; \kappa \rangle}
\qquad
\frac{w = \langle v; \phi; \rho \rangle}{\langle \texttt{let}\ x : \tau_x\ \texttt{=}\ v\ \texttt{in}\ e; \phi; \rho; \kappa \rangle \longrightarrow \langle e; \phi; \rho, x \mapsto w; \kappa \rangle}
$$

$$
\frac{\begin{array}{c}\rho(x_f) = w_f \qquad w_f = \langle \mu f : \tau_f \,.\, \lambda z : \tau_z \,.\, e_b; \phi_f; \rho_f \rangle \\ \rho(x_a) = w_a \qquad \vdash w_a :\equiv \langle \tau_z; \phi_f \rangle\end{array}}{\begin{array}{l}\langle \texttt{let}\ x : \tau_x\ \texttt{=}\ x_f\ x_a\ \texttt{in}\ e; \phi; \rho; \kappa \rangle \\ \quad \longrightarrow \langle e_b; \phi_f; \rho_f, f \mapsto w_f, z \mapsto w_a; \langle x; \tau_x; e; \phi; \rho \rangle :: \kappa \rangle\end{array}}
$$

$$
\frac{\begin{array}{c}\rho(x_f) = w_f \qquad w_f = \langle \mu f : \tau_f \,.\, \Lambda \beta \,.\, e_b; \phi_f; \rho_f \rangle \\ \pi_a = \langle \tau_a; \phi \rangle\end{array}}{\begin{array}{l}\langle \texttt{let}\ x : \tau_x\ \texttt{=}\ x_f\ \texttt{[}\tau_a\texttt{]}\ \texttt{in}\ e; \phi; \rho; \kappa \rangle \\ \quad \longrightarrow \langle e_b; \phi_f, \beta \mapsto \pi_a; \rho_f, f \mapsto w_f; \langle x; \tau_x; e; \phi; \rho \rangle :: \kappa \rangle\end{array}}
$$

Fig. 3. Operational Semantics of ANF System F

the judgement $\Delta \vdash \tau$ asserts that all of the free type variables of type τ are recorded in Δ. A variable context Γ records free variables and their types and the judgements $\Delta; \Gamma \vdash v : \tau$ and $\Delta; \Gamma \vdash e : \tau$ assert that value v and expression e have type τ in Δ and Γ; in the rule for type applications, we write $[\beta \rightarrowtail \tau_a]\tau_b$ for the capture-avoiding substitution of τ_a for free occurrences of β in τ_b. The judgement $\vdash P : \tau$ asserts that program P is closed and has type τ.

2.3 Operational Semantics

The operational semantics for our ANF-variant of System F is presented as an adaptation of the environment- and continuation-based C_aEK machine [10] and is given in Figure 3.

A machine state ς has four components: a control expression, a run-time type environment, a run-time value environment, and a continuation.

A run-time type environment ϕ is a map from type variables to run-time types and a run-time value environment ρ is a map from variables to run-time values. A run-time type π is a "type closure": a pair of a (possibly open) type and

a run-time type environment; the run-time type environment captures the free type variables of the type. A run-time value w is a "function closure" or a "type-abstraction closure": a triple of a (possibly open) value (a recursive function or a recursive type abstraction), a run-time type environment, and a run-time value environment; the run-time type environment captures the free type variables of the value and the run-time value environment captures the free variables of the value.

A continuation κ is a stack of frames, each of the form $\langle x; \tau_x; \phi; \rho; e \rangle$, where x is the variable receiving the result w of a non-tail function application or non-tail type application, τ_x is the (static, syntactic) type of x, and e is the expression to be evaluated in the environments ϕ and ρ extended with x bound to w to yield the result of the frame.

Ignoring the shaded terms, the machine transition rules are straightforward. The first rule returns a result to the top-most frame of the continuation when the control expression has been reduced to a variable. The second rule creates function closures and type-abstraction closures. The third and fourth rules extract the expression body, run-time type environment, and run-time value environment from the applied function closure or type-abstraction closure, extend the closure's run-time value environment with f bound to the applied function closure or type-abstraction closure (making the recursive function or recursive type-abstraction available to the expression body), extend the closure's run-time type environment with the run-time value argument (in the case of a function application) or extend the closure's run-time type environment with the run-time type argument (in the case of a type application), and push a frame onto the continuation to receive the result of the function application or type application. Note that the machine transitions are syntax directed and deterministic.

We now consider the shaded terms in the first and third rules and the judgements and rules in Figure 4. In essence, the shaded terms perform a kind of run-time type checking at the point where a run-time value environment is extended with a non-local run-time value. In the first rule, the result w_r must have a run-time type equal to $\langle \tau_z; \phi \rangle$, the run-time type of the variable receiving the result. In the third rule, the argument w_a must have a run-time type equal to $\langle \tau_z; \phi_f \rangle$, the run-time type of the variable receiving the argument.

The rules for the judgement $\vdash w :\equiv \pi$ simply form the run-time type of the recursive function or recursive type abstraction from the (static, syntactic) type of the μ-bound variable and the run-time type environment of the closure; note that there is no type checking of the value using the typing judgements from the static semantics. The judgement $\Delta \vdash \pi_1 \equiv \pi_2$ asserts that the run-time types π_1 and π_2 are equal under Δ. The first and second rules expand a Λ-bound type variable according to the appropriate run-time type environment. The third rule asserts that two function types are equal when their argument types are equal and their result types are equal. The fifth rule asserts that two universal types (sharing the same $\forall$-bound type variable via α conversion) are equal when their range types are equal. The fourth rule asserts that two $\forall$-bound type variables are equal when they are the same type variable. Note that the first and second

$\boxed{\Delta \vdash \pi \equiv \pi}$

$$\frac{\phi_1(\alpha_1) = \pi_1' \qquad \emptyset \vdash \pi_1' \equiv \pi_2}{\Delta \vdash \langle \alpha_1; \phi_1 \rangle \equiv \pi_2} \qquad \frac{\phi_2(\alpha_2) = \pi_2' \qquad \emptyset \vdash \pi_1 \equiv \pi_2'}{\Delta \vdash \pi_1 \equiv \langle \alpha_2; \phi_2 \rangle}$$

$$\frac{\Delta \vdash \langle \tau_{z1}; \phi_1 \rangle \equiv \langle \tau_{z2}; \phi_2 \rangle \qquad \Delta \vdash \langle \tau_{b1}; \phi_1 \rangle \equiv \langle \tau_{b2}; \phi_2 \rangle}{\Delta \vdash \langle \tau_{z1} \to \tau_{b1}; \phi_1 \rangle \equiv \langle \tau_{z2} \to \tau_{b2}; \phi_2 \rangle}$$

$$\frac{\Delta(\alpha_1) = \star \qquad \Delta(\alpha_2) = \star \qquad \alpha_1 = \alpha_2}{\Delta \vdash \langle \alpha_1; \phi_1 \rangle \equiv \langle \alpha_2; \phi_2 \rangle} \qquad \frac{\Delta, \alpha{:}\star \vdash \langle \tau_{b1}; \phi_1 \rangle \equiv \langle \tau_{b2}; \phi_2 \rangle}{\Delta \vdash \langle \forall \alpha.\ \tau_{b1}; \phi_1 \rangle \equiv \langle \forall \alpha.\ \tau_{b2}; \phi_2 \rangle}$$

$\boxed{\vdash w :\equiv \pi}$

$$\frac{\emptyset \vdash \langle \tau_f; \phi \rangle \equiv \pi}{\vdash \langle \mu f{:}\tau_f\,.\,\lambda z{:}\tau_z\,.\,e_b; \phi; \rho \rangle :\equiv \pi} \qquad \frac{\emptyset \vdash \langle \tau_f; \phi \rangle \equiv \pi}{\vdash \langle \mu f{:}\tau_f\,.\,\Lambda \beta\,.\,e_b; \phi; \rho \rangle :\equiv \pi}$$

Fig. 4. Run-time Type Equality

rules assert the equality of the expanded type under the empty type-variable context; this is because the expanded run-time type type is not in the scope of the $\forall$-bound type variables appearing in Δ.

We emphasize that the shaded terms and the judgements in Figure 4 are instrumentation that simplifies the proof of flow soundness in Section 3.1. The presence of these terms does not change the evaluation of well-typed programs, a fact that is established in Section 2.4.

2.4 Type Soundness

Theorem 1 (Type Soundness).
If $\vdash e : \tau$ *and* $\langle e; \emptyset; \emptyset; \bullet \rangle \longrightarrow^* \varsigma'$, *then either* $\varsigma' = \langle x'; \phi'; \rho'; \bullet \rangle$ *or* $\varsigma' \longrightarrow \varsigma''$.

A syntactic proof [50], using Progress and Preservation theorems, is given in a companion technical report [11]. In addition to the judgements of Figure 2, we introduce judgements that assert the "well-typedness" of run-time types ($\vdash \pi \Rrightarrow \tau$), run-time type environments ($\vdash \phi : \theta$), run-time values ($\vdash w : \tau$), run-time value environments ($\vdash \rho : \Gamma$), continuations ($\vdash \tau \rhd \kappa : \tau$), and states ($\vdash \varsigma : \tau$). Of note is the judgement $\vdash \phi : \theta$ that asserts that the run-time type environment ϕ corresponds to a substitution θ; the domains of ϕ and θ are equal, but whereas ϕ maps a type variable to a run-time type (a pair of a (possibly open) type and a run-time type environment), θ maps a type variable to a closed type obtained by (recursively) expanding the (possibly open) type by the run-time type environment.

Preservation is entirely straightforward, but Progress requires showing that the shaded terms of Figure 2 can always be satisfied with well-typed configurations. A key lemma is the following, which establishes that two run-time types may be judged equal if their induced expansions are syntactically equal:

Lemma 1 (Syntactic Type Equality of Expansions implies Run-Time Type Equality).
If $\vdash \phi_1 : \theta_1$, $\mathsf{dom}(\phi_1) \cap \mathsf{dom}(\Delta) = \emptyset$, $\Delta \vdash \theta_1(\tau_1)$,
$\vdash \phi_2 : \theta_2$, $\mathsf{dom}(\phi_2) \cap \mathsf{dom}(\Delta) = \emptyset$, $\Delta \vdash \theta_2(\tau_2)$,
and $\theta_1(\tau_1) = \theta_2(\tau_2)$,
then $\Delta \vdash \langle \tau_1; \phi_1 \rangle \equiv \langle \tau_2; \phi_2 \rangle$.

An immediate corollary to Type Soundness is that, for well-typed programs, the operational semantics of Figure 3 with the shaded terms is equivalent to the operational semantics without the shaded terms.

3 Type- And Control-Flow Analysis

Our type- and control-flow analysis is presented as an adaptation of the syntax-directed 0CFA, the classic monovariant control-flow analysis [32, Section 3.3], and is given in Figure 5.

The result of our type- and control-flow analysis is a pair of abstract environments. An abstract type environment $\hat{\phi}$ is a map from type variables to sets of abstract types, where an abstract type is simply a (possibly open) type.[5] An abstract value environment $\hat{\rho}$ is a map from variables to sets of abstract values, where an abstract value is simply a (possibly open) recursive function or recursive type abstraction.[6] Abstract type and value environments form complete lattices.

The judgements $\hat{\phi}; \hat{\rho} \vDash P$, $\hat{\phi}; \hat{\rho} \vDash e$, and $\hat{\phi}; \hat{\rho} \vDash v$ assert that a pair of abstract environments is an acceptable type- and control-flow analysis of the program P, expression e, and value v, respectively. An acceptable type- and control-flow analysis is one that soundly approximates the run-time behavior of the program, in a sense made precise in Section 3.1; intuitively, acceptable abstract type and value environments must describe every run-time type and value environment that arises during evaluation of the program.

Ignoring the shaded terms, the constraints asserted by the rules are standard for a monovariant control-flow analysis. The rules for values assert that the value itself is included in the set of abstract values mapped from the μ-bound variable f (corresponding to the $f \mapsto w_f$ binding in the operational semantics at makes

[5] We introduce abstract types in preparation for future extensions of the analysis; for example, we may wish to introduce a $\top$ abstract type to represent an unknown type coming from outside the scope of the analysis.

[6] Again, we introduce abstract values in preparation for future extensions of the analysis; for example, we may wish to introduce a $\top$ abstract value to represent an unknown value coming from outside the scope of the analysis or we may wish to introduce $[m, n]$ abstract values to incorporate an interval/range data-flow analysis [6,15].

$$
\begin{array}{lrl}
\textit{Abstract types} & \textit{AType} \ni \hat{\pi} & ::= \tau \\
\textit{Sets of abstract types} & \mathcal{P}(\textit{AType}) \ni \hat{\Pi} & \\
\textit{Abstract values} & \textit{AValue} \ni \hat{w} & ::= v \\
\textit{Sets of abstract values} & \mathcal{P}(\textit{AValue}) \ni \hat{W} &
\end{array}
$$

$$
\begin{array}{lrcl}
\textit{Abstract type environments} & \textit{ATEnv} \ni \hat{\phi} & \in & \textit{TyVar} \to \mathcal{P}(\textit{AType}) \\
\textit{Abstract value environments} & \textit{AEnv} \ni \hat{\rho} & \in & \textit{Var} \to \mathcal{P}(\textit{AValue})
\end{array}
$$

$$
\begin{array}{ll}
\hat{\phi}_1 \sqsubseteq \hat{\phi}_2 \stackrel{\text{def}}{=} \forall \alpha \in \textit{TyVar}.\hat{\phi}_1(\alpha) \subseteq \hat{\phi}_2(\alpha) & \hat{\rho}_1 \sqsubseteq \hat{\rho}_2 \stackrel{\text{def}}{=} \forall x \in \textit{Var}.\hat{\rho}_1(x) \subseteq \hat{\rho}_2(x) \\
\left(\bigsqcup_{i\in I} \hat{\phi}_i\right)(\alpha) \stackrel{\text{def}}{=} \bigcup_{i\in I} \hat{\phi}_i(\alpha) & \left(\bigsqcup_{i\in I} \hat{\rho}_i\right)(x) \stackrel{\text{def}}{=} \bigcup_{i\in I} \hat{\rho}_i(x) \\
\left(\bigsqcap_{i\in I} \hat{\phi}_i\right)(\alpha) \stackrel{\text{def}}{=} \bigcap_{i\in I} \hat{\phi}_i(\alpha) & \left(\bigsqcap_{i\in I} \hat{\rho}_i\right)(x) \stackrel{\text{def}}{=} \bigcap_{i\in I} \hat{\rho}_i(x) \\
\hat{\phi}_\bot(\alpha) \stackrel{\text{def}}{=} \{\} & \hat{\rho}_\bot(x) \stackrel{\text{def}}{=} \{\} \\
\hat{\phi}_\top(\alpha) \stackrel{\text{def}}{=} \textit{AType} & \hat{\rho}_\top(x) \stackrel{\text{def}}{=} \textit{AValue}
\end{array}
$$

$\boxed{\hat{\phi};\hat{\rho} \vDash v}$

$$
\frac{\hat{\rho}(f) \supseteq \{\mu f\!:\!\tau_f\,.\,\lambda z\!:\!\tau_z\,.\,e_b\} \qquad \hat{\phi};\hat{\rho} \vDash e_b}{\hat{\phi};\hat{\rho} \vDash \mu f\!:\!\tau_f\,.\,\lambda z\!:\!\tau_z\,.\,e_b}
\qquad
\frac{\hat{\rho}(f) \supseteq \{\mu f\!:\!\tau_f\,.\,\Lambda\beta\,.\,e_b\} \qquad \hat{\phi};\hat{\rho} \vDash e_b}{\hat{\phi};\hat{\rho} \vDash \mu f\!:\!\tau_f\,.\,\Lambda\beta\,.\,e_b}
$$

$\boxed{\hat{\phi};\hat{\rho} \vDash e}$

$$
\frac{}{\hat{\phi};\hat{\rho} \vDash x}
\qquad
\frac{\hat{\phi};\hat{\rho} \vDash v \qquad \hat{\rho}(x) \supseteq \{v\} \qquad \hat{\phi};\hat{\rho} \vDash e}{\hat{\phi};\hat{\rho} \vDash \texttt{let}\ x\!:\!\tau_x\ \texttt{=}\ v\ \texttt{in}\ e}
$$

$$
\frac{\displaystyle\bigwedge_{\mu f\!:\!\tau_f\,.\,\lambda z\!:\!\tau_z\,.\,e_b \in \hat{\rho}(x_f)} \left(\begin{array}{l} \hat{\rho}(z) \supseteq \{\hat{w}_a \in \hat{\rho}(x_a) \mid \vdash \hat{\phi} \twoheadrightarrow \hat{w}_a :\approx \tau_z\} \\ \wedge\ \hat{\rho}(x) \supseteq \{\hat{w}_b \in \hat{\rho}(\mathsf{last}(e_b)) \mid \vdash \hat{\phi} \twoheadrightarrow \hat{w}_b :\approx \tau_x\} \end{array}\right) \qquad \hat{\phi};\hat{\rho} \vDash e}{\hat{\phi};\hat{\rho} \vDash \texttt{let}\ x\!:\!\tau_x\ \texttt{=}\ x_f\ x_a\ \texttt{in}\ e}
$$

$$
\frac{\displaystyle\bigwedge_{\mu f\!:\!\tau_f\,.\,\Lambda\beta\,.\,e_b \in \hat{\rho}(x_f)} \left(\begin{array}{l} \hat{\phi}(\beta) \supseteq \{\tau_a\} \\ \wedge\ \hat{\rho}(x) \supseteq \{\hat{w}_b \in \hat{\rho}(\mathsf{last}(e_b)) \mid \vdash \hat{\phi} \twoheadrightarrow \hat{w}_b :\approx \tau_x\} \end{array}\right) \qquad \hat{\phi};\hat{\rho} \vDash e}{\hat{\phi};\hat{\rho} \vDash \texttt{let}\ x\!:\!\tau_x\ \texttt{=}\ x_f\ \texttt{[}\tau_a\texttt{]}\ \texttt{in}\ e}
$$

$\boxed{\hat{\phi};\hat{\rho} \vDash P}$

$$
\frac{\hat{\phi};\hat{\rho} \vDash e}{\hat{\phi};\hat{\rho} \vDash e}
$$

Fig. 5. Type- and Control-Flow Analysis of ANF System F

$\boxed{\Delta \vdash \hat{\phi} \twoheadrightarrow \hat{\pi} \approx\!\!> \hat{\pi}}$

$$\frac{\hat{\pi}_1' \in \hat{\phi}(\alpha_1) \qquad \emptyset \vdash \hat{\phi} \twoheadrightarrow \hat{\pi}_1' \approx\!\!> \hat{\pi}_2}{\Delta \vdash \hat{\phi} \twoheadrightarrow \alpha_1 \approx\!\!> \hat{\pi}_2} \qquad \frac{\hat{\pi}_2' \in \hat{\phi}(\alpha_2) \qquad \emptyset \vdash \hat{\phi} \twoheadrightarrow \hat{\pi}_1 \approx\!\!> \hat{\pi}_2'}{\Delta \vdash \hat{\phi} \twoheadrightarrow \hat{\pi}_1 \approx\!\!> \alpha_2}$$

$$\frac{\Delta \vdash \hat{\phi} \twoheadrightarrow \tau_{z1} \approx\!\!> \tau_{z2} \qquad \Delta \vdash \hat{\phi} \twoheadrightarrow \tau_{b1} \approx\!\!> \tau_{b2}}{\Delta \vdash \hat{\phi} \twoheadrightarrow \tau_{z1} \to \tau_{b1} \approx\!\!> \tau_{z2} \to \tau_{b2}}$$

$$\frac{\Delta(\alpha_1) = \star \qquad \Delta(\alpha_2) = \star \qquad \alpha_1 = \alpha_2}{\Delta \vdash \hat{\phi} \twoheadrightarrow \alpha_1 \approx\!\!> \alpha_2} \qquad \frac{\Delta, \alpha{:}\star \vdash \hat{\phi} \twoheadrightarrow \tau_{b1} \approx\!\!> \tau_{b2}}{\Delta \vdash \hat{\phi} \twoheadrightarrow \forall\alpha.\ \tau_{b1} \approx\!\!> \forall\alpha.\ \tau_{b2}}$$

$\boxed{\vdash \hat{\phi} \twoheadrightarrow \hat{w} :\!\approx\!\!> \hat{\pi}}$

$$\frac{\emptyset \vdash \hat{\phi} \twoheadrightarrow \tau_f \approx\!\!> \hat{\pi}}{\vdash \hat{\phi} \twoheadrightarrow \mu f{:}\tau_f\,.\,\lambda z{:}\tau_z\,.\,e_b :\!\approx\!\!> \hat{\pi}} \qquad \frac{\emptyset \vdash \hat{\phi} \twoheadrightarrow \tau_f \approx\!\!> \hat{\pi}}{\vdash \hat{\phi} \twoheadrightarrow \mu f{:}\tau_f\,.\,\Lambda\beta\,.\,e_b :\!\approx\!\!> \hat{\pi}}$$

Fig. 6. Analysis-time Type Compatibility

the recursive function or recursive type-application available to the expression body) and that the abstract environments are acceptable for the expression body. Each of the rules for `let`-binding expressions `let` $x{:}\tau_x$ `=` $\cdots$ `in` e assert that the abstract environments are acceptable for the expression e. The rule for `let`-bindings of values asserts that the abstract environments are acceptable for the value v and that the value v is included in the set of abstract values mapped from the variable x. The rule for `let`-bindings of non-tail function applications assert that, for all recursive functions in the set of abstract values mapped from the variable x_f, the abstract values mapped from the actual argument x_a flow to the formal argument z and the abstract values from the function result $\mathsf{last}(e_b)$ flow to the receiving variable x. Similarly, the rule for `let`-bindings of non-tail type applications assert that, for all recursive type abstractions in the set of abstract values mapped from the variable x_f, the actual type argument τ_a flows to the formal type argument β and the abstract values from the function result $\mathsf{last}(e_b)$ flow to the receiving variable x.

We now consider the shaded terms in the third and fourth rules of the $\hat{\phi}; \hat{\rho} \vDash e$ judgement and the judgements and rules in Figure 6. In essence, the shaded terms perform a kind of analysis-time type checking at the point where there is a non-local flow of abstract values. In the third rule, each abstract argument $\hat{w}_a$ that flows to the formal argument z must have an abstract type compatible with τ_z, the static type of formal argument; this is the abstract analogue of the run-time type equality condition in the function application rule of the operational semantics. Similarly, in the third and fourth rules, each abstract result $\hat{w}_b$ that

flow to the receiving variable x must have an abstract type compatible with τ_x, the static type of the receiving variable; this is the abstract analogue of the run-time type equality condition in the return rule of the operational semantics.

The rules for the judgement $\vdash \hat{\phi} \Rrightarrow \hat{w} :\approx \pi$ simply form the abstract type of the recursive function or recursive type abstraction from the (static, syntactic) type of the μ-bound variable. The judgement $\Delta \vdash \hat{\phi} \Rrightarrow \hat{\pi}_1 \equiv \hat{\pi}_2$ asserts that the abstract types $\hat{\pi}_1$ and $\hat{\pi}_2$ are compatible under $\hat{\phi}$ and Δ. The first and second rules expand a Λ-bound type variable according to the (global) abstract type environment; note that these rules must "guess" a satisfying abstract type from among the set of abstract types mapped from the type variable, unlike the corresponding rules in the judgement $\Delta \vdash \pi_1 \equiv \pi_2$, where the expansion is uniquely determined by the (local) run-time type environment. The third rule asserts that two function types are compatible when their argument types are compatible and their result types are compatible. The fifth rule asserts that two universal types (sharing the same $\forall$-bound type variable via α conversion) are compatible when their range types are compatible. The fourth rule asserts that two $\forall$-bound type variables are compatible when they are the same type variable. Note that the first and second rules assert the compatibility of the expanded abstract type under the empty type-variable context; this is because the expanded abstract type is not in the scope of the $\forall$-bound type variables appearing in Δ.

3.1 Flow Soundness

We now show that every acceptable (with respect to a given program) pair of abstract environments soundly approximates the run-time behavior of the program. To formalize the approximation, we introduce "shallow" abstraction functions that take run-time types and values to abstract types and values and that take run-time type and value environments to abstract type and value environments:[7]

$$\begin{array}{rl} |\cdot| :: & \mathit{RType} \to \mathit{AType} \\ |\langle \tau; \phi \rangle| = & \tau \end{array} \qquad \begin{array}{rl} |\cdot| :: & \mathit{RValue} \to \mathit{AValue} \\ |\langle v; \phi; \rho \rangle| = & v \end{array}$$

$$|\cdot| :: \mathit{RTEnv} \to \mathit{ATEnv} \qquad |\phi|(\alpha) = \begin{cases} \{\} & \text{if } \alpha \notin \mathsf{dom}(\phi) \\ \{|\pi|\} & \text{if } \phi(\alpha) = \pi \end{cases}$$

$$|\cdot| :: \mathit{REnv} \to \mathit{AEnv} \qquad |\rho|(x) = \begin{cases} \{\} & \text{if } x \notin \mathsf{dom}(\rho) \\ \{|w|\} & \text{if } \rho(x) = w \end{cases}$$

Theorem 2 (Flow Soundness).
If $\hat{\phi}; \hat{\rho} \vDash e$ *and* $\langle e; \emptyset; \emptyset; \bullet \rangle \longrightarrow^* \langle e'; \phi'; \rho'; \kappa' \rangle$, *then* $|\phi'| \sqsubseteq \hat{\phi}$ *and* $|\rho'| \sqsubseteq \hat{\rho}$.

A proof, using a Preservation (aka, subject reduction) theorem, is given in a companion technical report [11]. In addition to the judgements of Figure 5,

[7] These abstraction functions are "shallow" in the sense that they do not abstract and join the embedded run-time type and value environments of run-time types and values.

we introduce judgements that assert the acceptability of abstract environments with respect to run-time types ($\hat{\phi} \vdash \pi$), run-time type environments ($\hat{\phi} \vdash \phi$), run-time values ($\hat{\phi}; \hat{\rho} \vdash w$), run-time value environments ($\hat{\phi}; \hat{\rho} \vdash \rho$), continuations ($\hat{\phi}; \hat{\rho} \vdash x \rhd \kappa$), and states ($\hat{\phi}; \hat{\rho} \vdash \varsigma$). The judgements $\hat{\phi} \vdash \phi$ and $\hat{\phi}; \hat{\rho} \vdash \rho$ assert that the abstract environments are "deep" abstractions of the run-time environments.

Preservation is straightforward, simplified by the explicit run-time type equality conditions in the operational semantics and the following lemma, which establishes that type compatibility soundly approximates type equality:

Lemma 2 (Run-Time Type Equality implies Analysis-Time Type Compatibility). *If* $\hat{\phi} \vDash \phi_1$, $\hat{\phi} \vDash \phi_2$, *and* $\Delta \vdash \langle \tau_1; \phi_1 \rangle \equiv \langle \tau_2; \phi_2 \rangle$, *then* $\Delta \vdash \hat{\phi} \twoheadrightarrow \tau_1 \approx\!\!\approx \tau_2$.

It is perhaps surprising to note that Flow Soundness of our type- and control-flow analysis does not require a well-typed source program. In essence, the explicit run-time type equality conditions in the instrumented operational semantics ensure that a machine state is "just well-typed enough" to preserve the acceptability of abstract environments across a taken machine transition. A well-typed source program and Type Soundness ensure that machine transitions may be repeatedly taken. This suggests an alternate presentation in which we adopt an uninstrumented operational semantics (i.e., Figure 3 without the shaded terms) and require a well-typed source program for Flow Soundness; Progress (for Type Soundness) would become straightforward, but Preservation (for Flow Soundness) would require a lemma (essentially, a chaining of Lemma 1 and Lemma 2) that establishes that the abstractions of two run-time types may be judged compatible if their induced expansions are syntactically equal, forgoing the run-time type equality judgement entirely. The necessary preconditions for this lemma would be obtained from the well-typedness of the machine state undergoing transition.

3.2 Existence of Minimum, Finite Flows

As is typical for a flow analysis, for a given program, there may be many acceptable pairs of abstract environments. One intuitively acceptable pair of abstract type- and value-environments is the one that maps every Λ-bound type variable that occurs in the program to the set of types that occur in the program and that maps every `let`-, μ-, and λ-bound variable that occurs in the program to the set of values that occur in the program:

$$\hat{\phi}^P_\top(\alpha) = \begin{cases} \{\} & \text{if } \alpha \notin \mathit{TyVar}_P \\ \mathit{Type}_P & \text{if } \alpha \in \mathit{TyVar}_P \end{cases} \qquad \hat{\rho}^P_\top(x) = \begin{cases} \{\} & \text{if } x \notin \mathit{Var}_P \\ \mathit{Value}_P & \text{if } x \in \mathit{Var}_P \end{cases}$$

Note that these abstract environments are "finite", in the sense that they map a finite set of elements to finite sets (and the remaining elements to empty sets).

The following theorem establishes that minimum, finite acceptable abstract environments exist for every program:

Theorem 3 (Minimum, Finite Flows Exist).
For all programs P,
there exist minimum abstract environments $\hat{\phi}_{\min} \sqsubseteq \hat{\phi}^P_\top$ and $\hat{\rho}_{\min} \sqsubset \hat{\rho}^P_\top$
such that $\hat{\phi}_{\min}; \hat{\rho}_{\min} \vDash P$.

A proof is given in a companion technical report [11], first showing that $\hat{\phi}^P_\top$ and $\hat{\rho}^P_\top$ are an acceptable pair of abstract environments and then showing that the greatest lower bound of a (possibly infinite) set of acceptable pairs of abstract environments is an acceptable pair of abstract environments. Furthermore, for a given program P, we may restrict ourselves to considering abstract type environments $\hat{\phi}^P \in \mathit{ATEnv}_P = \mathit{TyVar}_P \to \mathcal{P}(\mathit{Type}_P)$ and abstract value environments $\hat{\rho}^P \in \mathit{AEnv}_P = \mathit{Var}_P \to \mathcal{P}(\mathit{Value}_P)$, which form finite, complete lattices.

3.3 Computability of Minimum, Finite Flows

While Theorem 3 establishes that minimum, finite acceptable abstract environments exist for every program, we would like such abstract environments to be computable. The key concern is the decidability of the $\Delta \vdash \hat{\phi} \Rrightarrow \tau_1 \approx\!\!\!\!\Rightarrow \tau_2$ judgement. Even simply verifying that a pair of abstract environments is acceptable for a given program requires showing that constraints of the form $\hat{\rho}(z) \supseteq \{\hat{w} \in \hat{\rho}(x) \mid \vdash \hat{\phi} \Rrightarrow \hat{w} :\!\approx\!\!\!\!\Rightarrow \tau_z\}$ are satisfied; this, in turn, requires showing, for each an abstract value $\hat{w}$ that is an element of $\hat{\rho}(x)$ but not an element of $\hat{\rho}(z)$, that the judgement $\vdash \hat{\phi} \Rrightarrow \hat{w} :\!\approx\!\!\!\!\Rightarrow \tau_z$ is not derivable.[8]

Due to "recursion" in the abstract type environment, whereby a type variable may be mapped to a set of abstract types in which the type variable itself occurs free, we cannot exhaustively apply the first and second rules of the $\Delta \vdash \hat{\phi} \Rrightarrow \tau_1 \approx\!\!\!\!\Rightarrow \tau_2$ judgement in order to search for a derivation. Consider deciding whether or not $\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \mathsf{int} \to \beta \approx\!\!\!\!\Rightarrow \alpha$ is derivable where $\hat{\phi}^\ddagger(\alpha) = \{\mathsf{int} \to \mathsf{int}, \mathsf{int} \to \alpha\}$ and $\hat{\phi}^\ddagger(\beta) = \{\mathsf{int} \to \mathsf{bool}, \mathsf{int} \to \beta\}$.[9] We must avoid getting "stuck" exploring the infinite non-derivation:

$$\dfrac{\mathsf{int} \to \alpha \in \hat{\phi}^\ddagger(\alpha) \qquad \dfrac{\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \mathsf{int} \approx\!\!\!\!\Rightarrow \mathsf{int} \qquad \dfrac{\mathsf{int} \to \beta \in \hat{\phi}^\ddagger(\beta) \qquad \dfrac{\vdots}{\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \mathsf{int} \to \beta \approx\!\!\!\!\Rightarrow \alpha}}{\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \beta \approx\!\!\!\!\Rightarrow \alpha}}{\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \mathsf{int} \to \beta \approx\!\!\!\!\Rightarrow \mathsf{int} \to \alpha}}{\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \mathsf{int} \to \beta \approx\!\!\!\!\Rightarrow \alpha}$$

To circumvent this issue, we take inspiration from the theory and implementation of regular-tree grammars [12,2,7], which has been used extensively for

[8] Note, however, that this does not require showing, for each abstract value $\hat{w}$ that is an element of both $\hat{\rho}(x)$ and $\hat{\rho}(z)$, that the judgement $\vdash \hat{\phi} \Rrightarrow \hat{w} :\!\approx\!\!\!\!\Rightarrow \tau_z$ is derivable; the constraint is satisfied whether or not the judgement is derivable.

[9] It is not derivable.

flow analysis [23,22,17,16] (including type inference [28,3]), but whereas previous work has applied regular-tree grammars to the analysis of values, we apply regular-tree grammars to the analysis of types.

Given an abstract type environment $\hat{\phi}$, we interpret it as a regular-tree grammar as follows: $\mathsf{dom}(\hat{\phi})$ is the set of non-terminals and $\{\alpha \Rightarrow \hat{\pi} \mid \alpha \in \mathsf{dom}(\hat{\phi}) \wedge \hat{\pi} \in \hat{\phi}(\alpha)\}$ is the set of productions. The language $\mathcal{L}_{\hat{\phi}}(\hat{\pi})$ generated by the grammar $\hat{\phi}$ for the starting term $\hat{\pi}$ is $\mathcal{L}_{\hat{\phi}}(\hat{\pi}) \stackrel{\text{def}}{=} \{\hat{\pi}' \in \overline{AType} \mid \hat{\pi} \Rightarrow^*_{\hat{\phi}} \hat{\pi}'\}$, where $\overline{AType}$ is the set of closed abstract types and $\hat{\pi}_1 \Rightarrow_{\hat{\phi}} \hat{\pi}_2$ is the relation that (capture-avoidingly) substitutes for one (free) non-terminal of $\hat{\pi}_1$ the right-hand side of one of its productions to obtain $\hat{\pi}_2$.[10]

Intuitively, $\emptyset \vdash \hat{\phi}^{\ddagger} \twoheadrightarrow \mathsf{int} \to \beta \approx\!\!\!\approx \alpha$ is not derivable because

$$\mathcal{L}_{\hat{\phi}^{\ddagger}}(\mathsf{int} \to \beta) = \{\mathsf{int} \to \mathsf{int} \to \mathsf{bool}, \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{bool}, \ldots\}$$
$$\mathcal{L}_{\hat{\phi}^{\ddagger}}(\alpha) = \{\mathsf{int} \to \mathsf{int}, \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \ldots\}$$

and $\mathcal{L}_{\hat{\phi}^{\ddagger}}(\mathsf{int} \to \beta) \cap \mathcal{L}_{\hat{\phi}^{\ddagger}}(\alpha) = \emptyset$; there is no closed type that is generated by $\hat{\phi}^{\ddagger}$ from both $\mathsf{int} \to \beta$ and α. Formally,

Theorem 4 (Analysis-Time Type Compatibility iff Languages Intersect). *$\emptyset \vdash \hat{\phi} \twoheadrightarrow \hat{\pi}_1 \approx\!\!\!\approx \hat{\pi}_2$ if and only if $\mathcal{L}_{\hat{\phi}}(\hat{\pi}_1) \cap \mathcal{L}_{\hat{\phi}}(\hat{\pi}_2) \neq \emptyset$.*

A proof is given in a companion technical report [11].

An immediate corollary to Theorem 4 is the the decidability of type compatibility, since regular-tree grammars are closed under intersection and the emptiness of a regular-tree grammar is decidable [12,28]. In turn, we have that the minimum, finite acceptable abstract environments are computable for every program. Furthermore, given a program P, it is straightforward to read the analysis of Figure 5 as defining a monotone function from abstract environments to abstract environments; the "input" abstract environments are used for terms of the form $\hat{w} \in \hat{\rho}(x)$ and $\vdash \hat{\phi} \twoheadrightarrow \hat{w} :\!\approx\!\!\!\approx \tau$, while the "output" abstract environments are formed from the "input" abstract environments joined with $\hat{\phi}_{\perp}[\beta \mapsto \hat{\Pi}]$ and $\hat{\phi}_{\perp}[x \mapsto \hat{W}]$ for terms of the form $\hat{\phi}(\beta) \supseteq \hat{\Pi}$ and $\hat{\rho}(x) \supseteq \hat{W}$. The least fixed point of this monotone function, computable using a standard fixed-point computation, is the minimum, finite acceptable pair of abstract environments for the program P. Further considerations regarding the implementation of our type- and control-flow analysis are given in Section 5.

We briefly sketch implementations of testing emptiness and intersection of regular-tree grammars, based on those given by Aiken and Murphy [2]. Recall that, for a given program P, we may restrict ourselves to finite abstract type environments $\hat{\phi}^P \in ATEnv_P$.

[10] There are some subtleties in the treatment of $\forall$-bound type variables, which are perhaps best dealt with by adopting a locally-nameless representation [4] for types.

We define the function Ψ as follows:

$$\begin{aligned}
\Psi &:: \mathit{AEnv} \to (\mathit{TyVar} \to \mathbb{B}) \\
\Psi(\hat{\phi}) &= \mathrm{lfp}\, F \\
&\quad \text{where} \qquad F :: (\mathit{TyVar} \to \mathbb{B}) \to (\mathit{TyVar} \to \mathbb{B}) \\
&\qquad\qquad F(\psi)(\alpha) = \begin{cases} \top & \text{if } \exists \hat{\pi} \in \hat{\phi}(\alpha).\ \forall \beta \in \mathsf{FTV}(\hat{\pi}).\ \psi(\beta) = \top \\ \bot & \text{if } \forall \hat{\pi} \in \hat{\phi}(\alpha).\ \exists \beta \in \mathsf{FTV}(\hat{\pi}).\ \psi(\beta) = \bot \end{cases}
\end{aligned}$$

If $\hat{\phi}$ is finite, then $\Psi(\hat{\phi})$ is computable using a standard fixed-point computation. The language $\mathcal{L}_{\hat{\phi}}(\hat{\pi})$ is non-empty if and only if $\forall \beta \in \mathsf{FTV}(\hat{\pi}).\ \psi(\beta) = \top$.

In order to intersect the languages generated by the finite regular-tree grammar $\hat{\phi}$ for the starting terms $\hat{\pi}_1$ and $\hat{\pi}_2$, we extend $\hat{\phi}$ with finitely many additional non-terminals and productions to obtain $\hat{\phi}^\star$ and generate a starting term $\hat{\pi}^\star$ such that $\mathcal{L}_{\hat{\phi}}(\hat{\pi}_1) \cap \mathcal{L}_{\hat{\phi}}(\hat{\pi}_2) = \mathcal{L}_{\hat{\phi}^\star}(\hat{\pi}^\star)$. The idea is that each new non-terminal represents the intersection of a type variable in $\mathsf{dom}(\hat{\phi})$ and a type; a global mapping from pairs of type variables and types to new non-terminals is maintained to ensure that the same new non-terminal is used whenever the same pair is encountered.

To illustrate the technique, consider intersecting the languages generated by $\hat{\phi}^\ddagger$ for the starting terms $\mathsf{int} \to \beta$ and α. First, extend the grammar with a new non-terminal Z and no productions (i.e., extend $\hat{\phi}^\ddagger$ with the mapping $Z \mapsto \{\}$); the non-terminal Z will serve as starting term for an empty language. We are trying to intersect $\mathsf{int} \to \beta$ and α; since α is a non-terminal, generate a new non-terminal A_0 mapped from the pair $\langle \mathsf{int} \to \beta; \alpha \rangle$, add the triple $\langle A_0; \{\mathsf{int} \to \beta\}; \hat{\phi}^\ddagger(\alpha) \rangle$ to a work list; and return A_0 as the result of intersecting $\mathsf{int} \to \beta$ and α. The work list contains new non-terminals whose productions should be generated by intersecting all pairs of elements from the two sets. Therefore, we next add productions corresponding to $A_0 \Rightarrow \mathsf{int} \to \beta \mathbin{⍉} \mathsf{int} \to \mathsf{int}$ and $A_0 \Rightarrow \mathsf{int} \to \beta \mathbin{⍉} \mathsf{int} \to \alpha\}$. Intersecting $\mathsf{int} \to \beta$ and $\mathsf{int} \to \mathsf{int}$ generates a new non-terminal A_1 mapped from $\langle \beta; \mathsf{int} \rangle$, adds $\langle A_1; \hat{\phi}^\ddagger(\beta); \{\mathsf{int}\} \rangle$ to the worklist, and returns $\mathsf{int} \to A_1$. Intersecting $\mathsf{int} \to \beta$ and $\mathsf{int} \to \alpha$ generates a new non-terminal A_2 mapped from $\langle \beta; \alpha \rangle$, adds $\langle A_1; \hat{\phi}^\ddagger(\beta); \hat{\phi}^\ddagger(\alpha) \rangle$ to the worklist, and returns $\mathsf{int} \to A_2$. Therefore, we extend with the mapping $A_0 \mapsto \{\mathsf{int} \to A_1\} \cup \{\mathsf{int} \to A_2\}$. Returning to the work list, we next add productions corresponding to $A_1 \Rightarrow \mathsf{int} \to \mathsf{bool} \mathbin{⍉} \mathsf{int}$ and $A_1 \Rightarrow \mathsf{int} \to \beta \mathbin{⍉} \mathsf{int}$. Intersecting $\mathsf{int} \to \mathsf{bool}$ and int returns Z (since clearly the intersection of the languages generated from these two starting terms is empty), as does intersecting $\mathsf{int} \to \beta$ and int; therefore, we extend with the mapping $A_1 \mapsto \{Z\} \cup \{Z\}$. Returning to the work list, we next add productions corresponding to $A_2 \Rightarrow \mathsf{int} \to \mathsf{bool} \mathbin{⍉} \mathsf{int} \to \mathsf{int}$ (this intersection returns Z), $A_2 \Rightarrow \mathsf{int} \to \mathsf{bool} \mathbin{⍉} \mathsf{int} \to \alpha$ (this intersection generates a new non-terminal A_3 mapped from $\langle \mathsf{bool}; \alpha \rangle$, adds $\langle A_3; \{\mathsf{bool}\}; \hat{\phi}^\ddagger(\alpha) \rangle$ to the worklist, and returns $\mathsf{int} \to A_3$), $A_2 \Rightarrow \mathsf{int} \to \beta \mathbin{⍉} \mathsf{int} \to \mathsf{int}$ (this intersection returns $\mathsf{int} \to A_1$, using the global map), and $A_2 \Rightarrow \mathsf{int} \to \beta \mathbin{⍉} \mathsf{int} \to \alpha$ (this intersection returns $\mathsf{int} \to A_2$); therefore, we extend with the mapping

$A_2 \mapsto \{Z\} \cup \{\mathsf{int} \to A_3\} \cup \{\mathsf{int} \to A_1\} \cup \{\mathsf{int} \to A_2\}$. Finally, we add productions corresponding to $A_3 \Rightarrow \mathsf{bool} \mathbin{\&} \mathsf{int} \to \mathsf{int}$ (this intersection returns Z) and $A_3 \Rightarrow \mathsf{bool} \mathbin{\&} \mathsf{int} \to \alpha$ (this intersection returns Z); therefore, we extend with the mapping $A_3 \mapsto \{Z\} \cup \{Z\}$. In summary, we have

Global map	New productions
$\langle \mathsf{int} \to \beta; \alpha \rangle \mapsto A_0$	$A_0 \mapsto \{\mathsf{int} \to A_1\} \cup \{\mathsf{int} \to A_2\}$
$\langle \beta; \mathsf{int} \rangle \mapsto A_1$	$A_1 \mapsto \{Z\} \cup \{Z\}$
$\langle \beta; \alpha \rangle \mapsto A_2$	$A_2 \mapsto \{Z\} \cup \{\mathsf{int} \to A_3\} \cup \{\mathsf{int} \to A_1\} \cup \{\mathsf{int} \to A_2\}$
$\langle \mathsf{bool}; \alpha \rangle \mapsto A_3$	$A_3 \mapsto \{Z\} \cup \{Z\}$
	$Z \mapsto \{\}$

To conclude, we return $\hat{\phi}^\star$ equal to $\hat{\phi}^\ddagger$ extended with the new productions and $\hat{\pi}^\star$ equal to A_0. Finally, note that $\Psi(\hat{\phi}^\star)(\hat{\pi}^\star) = \bot$, confirming that $\mathcal{L}_{\hat{\phi}^\ddagger}(\mathsf{int} \to \beta) \cap \mathcal{L}_{\hat{\phi}^\ddagger}(\alpha) = \emptyset$.

4 Related Work

There is surprisingly little work on control-flow analyses for statically-typed languages with polymorphic types. Control-flow analyses have typically been formulated for dynamically- or simply-typed languages.[11] Production implementations of control-flow analyses for Standard ML, a language with rank-1 polymorphism (i.e., "let"-polymorphism), typically handle the polymorphism either by monomorphisation [5] (explicitly eliminating the polymorphism before analysis) or by polyvariance [16] (implicitly eliminating the polymorphism during analysis).

The most closely related work is the "Type-Directed Flow Analysis for Typed Intermediate Languages" of Jagannathan, Weeks, and Wright [20], which describes a framework for polyvariant flow analyses of Λ_i, the predicative subset of System F extended with recursive procedures. A specific analysis called $S_{\mathcal{RT}}$ uses types to control polyvariance; essentially, $S_{\mathcal{RT}}$ introduces a distinct polyvariance context for each closed type at which a polymorphic function is applied. Furthermore, $S_{\mathcal{RT}}$ respects types, meaning that if $\hat{v} \in F(x)$ (the abstract value $\hat{v}$ is assigned to x by the analysis) and $x : \sigma$ (the type scheme σ is assigned to x by the type system), then $[\![\hat{v}]\!] \subseteq [\![\sigma]\!]$, where $[\![\cdot]\!]$ denotes a set of values. Unfortunately, $S_{\mathcal{RT}}$ does not terminate on programs that use polymorphic recursion [30,18,24]; such programs may instantiate a polymorphic function at an infinite number of closed types during execution. In contrast, our type- and control-flow analysis is computable for all programs in (impredicative) System F extended with recursive functions.

Another closely related work is the "Type-sensitive Control-Flow Analysis" of Reppy [37], which describes an extension of Serrano's version of 0CFA [42]

[11] Again, we draw a distinction between flow analyses expressed as sophisticated type systems and flow analyses of languages with sophisticated type systems.

that uses a program's type information to compute more precise results. Serrano's and Reppy's analyses are modular and use an abstract value $\top$ to denote an unknown value; variables bound outside the unit of analysis are assigned $\top$, as are the parameters of functions that escape the unit of analysis. Reppy's insight is that values of an abstract type can only be created within their defining module; hence, "unknown" values of the abstract type can be soundly approximated by the set of escaping values of the abstract type (a subset of the set of values of the abstract type created within the defining module). This leads to a type-indexed family of abstract values for unknown values, in addition to the $\top$ abstract value. Reppy's analysis is formulated for a simply-typed language with top-level abstract types; he suggests extending the analysis to a language with polymorphism by mapping type variables to the $\top$ abstract value. Our type- and control-flow analysis is a whole-program analysis, but has a more precise treatment of type variables.

5 Future Work

There are many directions for future work.

While Section 3.3 established the computability of the minimum, finite acceptable pair of abstract environments for every program, we would our type- and control-flow analysis to be efficiently computable. A popular approach for computing control-flow analyses is as a constraint-based analysis [1]; an initial phase generates constraints that a solution to the analysis must satisfy, while a subsequent phase solves the constraints.[12] The syntax-directed 0CFA that we adapt to our type- and control-flow analysis has an $O(n^3)$ algorithm following this approach [32, Section 3.4]. However, algorithms for solving a set of constraints are sensitive to the syntax of constraints; the filtering of sets by the derivability of the type-compatibility judgement may prove problematic, especially since the derivability of the type-compatibility judgement depends upon the abstract type environment.

Independent of the overall approach to computing our type- and control-flow analysis, it seems clear that we will need to efficiently decide the derivability of type-compatibility judgements with respect to abstract type environments. Section 3.3 established that this decision could be made by intersecting and testing the emptiness of regular-tree grammars. Both operations are (worst-case) quadratic time in the size of the regular-tree grammar. Aiken and Murphy [2, Section 4] suggest maintaining a regular-tree grammar with an invariant that makes testing the emptiness (of a non-terminal) constant time. Aiken and Murphy [2, Section 5.3] also suggest that the algorithm given previously, which generates only the intersections necessary to express the result, performs well in the typical case. We further observe that, for a fixed abstract type environment, we can maintain the global map from pairs of type variables and types to new non-terminals (where each new non-terminal represents the intersection

[12] More sophisticated approaches exist where additional constraints are generated during the solving phase.

of the expansions of the type variable under the abstract type environment and the type) across decisions of the derivability of type-compatibility. Hence, the (worst-case) quadratic time bounds all queries with a given abstract type environment, not each query. We may also be able to exploit the fact that we are computing the emptiness of an intersection of regular-tree grammars and are not interested in the intersection itself.

Another direction of future work is to extend the type- and control-flow analysis to handle unknown and escaping values. It should be straightforward to introduce a $\top$ abstract type and a $\top$ abstract value; conservatively, the $\top$ abstract type should be judged compatible with any other abstract type. A more interesting direction is to consider primitives that make essential use of higher-rank polymorphism, such as Haskell's `runST` [25,26].

Yet another direction is to extend the monovariant type- and control-flow analysis to a polyvariant analysis.

Finally, we would like to extend type- and control-flow analysis to languages with even more sophisticated type systems. Of particular interest is System F with guarded algebraic data types (GADTs), as we are interested in combining the flow-directed defunctionalization of Cejtin, Jagannathan, and Weeks [5] with the polymorphic typed defunctionalization of Pottier and Gauthier [35]. Also of interest is System F_ω, the higher-order polymorphic lambda-calculus: System F_ω has used as a target language for the elaboration of a full-featured, higher-order ML-like module language [41] and System F_ω extended with type equality coercions [45] is used as a typed intermediate language in the Glasgow Haskell Compiler (GHC).

References

1. Aiken, A.: Introduction to set constraint-based program analysis. Science of Computer Programming 35(2-3), 79–111 (1999)
2. Aiken, A., Murphy, B.R.: Implementing regular tree expressions. In: Hughes, J. (ed.) Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science, vol. 523, pp. 427–447. Springer-Verlag, Cambridge, Massachusetts (Aug 1991)
3. Aiken, A., Murphy, B.R.: Static type inference in a dynamically typed language. In: Cartwright, R.C. (ed.) Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages. pp. 279–290. Orlando, Florida (Jan 1991)
4. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th Annual ACM Symposium on Principles of Programming Languages. pp. 3–15. ACM, San Francisco, California (Jan 2008)
5. Cejtin, H., Jagannathan, S., Weeks, S.: Flow-directed closure conversion for typed languages. In: Smolka, G. (ed.) Proceedings of the Ninth European Symposium on Programming. Lecture Notes in Computer Science, vol. 1782, pp. 56–71. Springer-Verlag, Berlin, Germany (Mar 2000)
6. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Robinet, B. (ed.) Proceedingsof the Second International Symposium on Programming. pp. 106–130. Paris, France (Apr 1976)

7. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: Peyton Jones, S. (ed.) Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture. pp. 170–181. La Jolla, California (Jun 1995)
8. Danvy, O.: Three steps for the CPS transformation. Tech. Rep. CIS-92-2, Kansas State University, Manhattan, Kansas (1991)
9. Faxén, K.F.: Polyvariance, polymorphism and flow analysis. In: Dam, M. (ed.) Selected papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages. Lecture Notes in Computer Science, vol. 1192, pp. 260–278. Springer-Verlag (1997)
10. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Cartwright, R. (ed.) Proceedings of the ACM SIGPLAN 1993 Conference on Programming Languages Design and Implementation. pp. 237–247. ACM, Albuquerque, New Mexico (Jun 1993)
11. Fluet, M.: A type- and control-flow analysis for System F. Tech. rep., Rochester Institute of Technology (August 2012), `http://www.cs.rit.edu/~mtf/research/tcfa/IFL12/techrpt.pdf`
12. Gecseg, F., Steinby, M.: Tree Automata. Akademiai Kiado, Budapest, Hungary (1984)
13. Girard, J.Y.: Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In: Fenstad, J.E. (ed.) Proceedings of the 2nd Scandinavian Logic Symposium. pp. 63–92. Amsterdam, Netherlands (1971)
14. Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Lee, P. (ed.) Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages. pp. 130–141. ACM, ACM, San Francisco, California (Jan 1995)
15. Harrison III, W.L.: Compiler analysis of the value ranges for variables. IEEE Transactions on Software Engineering SE-3(3), 243–250 (May 1977)
16. Heintze, N.: Set-based program analysis of ML programs. In: Talcott, C.L. (ed.) Proceedings of the 1994 ACM Conference on Lisp and Functional Programming. pp. 306–317. LISP Pointers, Vol. VII, No. 3, Orlando, Florida (Jun 1994)
17. Heintze, N., Jaffar, J.: A finite presentation theorem for approximating logic programs. In: Hudak, P. (ed.) Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages. pp. 197–209. San Francisco, California (Jan 1990)
18. Henglein, F.: Type inference with polymorphic recursion. ACM Transactions on Programming Languages and Systems 15(2), 253–289 (1993)
19. Holdermans, S., Hage, J.: Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In: Weirich, S. (ed.) Proceedings of the Fifteenth ACM SIGPLAN International Conference on Functional Programming (ICFP'10). pp. 63–74. ACM, Baltimore, Maryland (Sep 2010)
20. Jagannathan, S., Weeks, S., Wright, A.K.: Type-directed flow analysis for typed intermediate languages. In: Hentenryck, P.V. (ed.) Static Analysis, 4th International Symposium, SAS '97. Lecture Notes in Computer Science, vol. 1302, pp. 232–249. Springer-Verlag, Paris, France (Sep 1997)
21. Jones, N.D.: Flow analysis of lambda expressions (preliminary version). In: Even, S., Kariv, O. (eds.) Automata, Languages and Programming, 8th Colloquium, Acre (Akko). Lecture Notes in Computer Science, vol. 115, pp. 114–128. Springer-Verlag, Israel (Jul 1981)

22. Jones, N.D.: Flow analysis of lazy higher-order functional programs. In: Abramsky, S., Hankin, C. (eds.) Abstract Interpretation of Declarative Languages, pp. 103–122. Ellis Horwood (1987)
23. Jones, N.D., Muchnick, S.S.: Flow analysis and optimization of LISP-like structures. In: Rosen, B.K. (ed.) Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages. pp. 244–256. San Antonio, Texas (Jan 1979)
24. Kfoury, A., Tiuryn, J., Urzyczyn, P.: Type reconstruction in the presence of polymorphic recursion. ACM Transactions on Programming Languages and Systems 15(2), 290–311 (1993)
25. Launchbury, J., Peyton Jones, S.: Lazy functional state threads. In: Sarkar, V. (ed.) Proceedings of the ACM SIGPLAN 1994 Conference on Programming Languages Design and Implementation. pp. 24–35. SIGPLAN Notices, Vol. 29, No 6, ACM, ACM, Orlando, Florida (Jun 1994)
26. Launchbury, J., Peyton Jones, S.: State in Haskell. Lisp and Symbolic Computation 8(4), 293–341 (1995)
27. Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys 44(3), 10:1–10:33 (Jun 2012)
28. Mishra, P., Reddy, U.S.: Declaration-free type checking. In: Van Deusen, M.S., Galil, Z., Reid, B.K. (eds.) Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages. pp. 7–21. ACM, ACM, New Orleans, Louisiana (Jan 1985)
29. Mossin, C.: Exact flow analysis. Mathematical Structures in Computer Science 13(1), 125–156 (2003)
30. Mycroft, A.: Polymorphic type schemes and recursive definitions. In: Paul, M., Robinet, B. (eds.) Proceedingsof International Symposium on Programming. Lecture Notes in Computer Science, vol. 167, pp. 217–228. Springer-Verlag, Toulouse, France (Apr 1984)
31. Nielson, F., Nielson, H.R.: Interprocedural control flow analysis. In: Swierstra, S.D. (ed.) Proceedings of the Eighth European Symposium on Programming. Lecture Notes in Computer Science, vol. 1576, pp. 20–39. Springer-Verlag, Amsterdam, The Netherlands (Mar 1999)
32. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
33. Palsberg, J.: Type-based analysis and applications. In: Field, J., Snelting, G. (eds.) PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 20–27 (2001)
34. Peyton Jones, S.: Compiling Haskell by program transformation: A report from the trenches. In: Nielson, H.R. (ed.) Proceedings of the Sixth European Symposium on Programming. Lecture Notes in Computer Science, vol. 1058, pp. 18–44. Springer-Verlag, Linköping, Sweden (Apr 1996)
35. Pottier, F., Gauthier, N.: Polymorphic typed defunctionalization and concretization. Higher-Order and Symbolic Computation 19(1), 125–162 (2006), a preliminary version was presented at the Thirty-First Annual ACM Symposium on Principles of Programming Languages (POPL 2004)
36. Rehof, J., Fähndrich, M.: Type-based flow analysis: from polymorphic subtyping to CFL-reachability. In: Nielson, H.R. (ed.) Proceedings of the 28th Annual ACM Symposium on Principles of Programming Languages. pp. 54–66. London, United Kingdom (Jan 2001)

37. Reppy, J.: Type-sensitive control-flow analysis. In: Kennedy, A., Pottier, F. (eds.) ML'06: Proceedings of the ACM SIGPLAN 2006 workshop on ML. pp. 74–83 (Sep 2006)
38. Reynolds, J.: Towards a theory of type structure. In: Robinet, B. (ed.) Proceedingsof Programming Symposium (Colloque sur la Programmation). Lecture Notes in Computer Science, vol. 19, pp. 408–425. Springer-Verlag, Paris, France (Apr 1974)
39. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of 25th ACM National Conference. pp. 717–740. Boston, Massachusetts (1972), reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [40]
40. Reynolds, J.C.: Definitional interpreters revisited. Higher-Order and Symbolic Computation 11(4), 355–361 (1998)
41. Rossberg, A., Russo, C., Dreyer, D.: F-ing modules. In: Benton, N. (ed.) TLDI'10: Proceedings of the Fifth ACM SIGPLAN Workshop on Types in Language Design and Implementation. pp. 89–102 (Jan 2010)
42. Serrano, M.: Control flow analysis: a functional languages compilation paradigm. In: Proceedings of the 1995 ACM Symposium on Applied Computing. pp. 118–122. Nashville, Tennessee (Feb 1995)
43. Sestoft, P.: Replacing function parameters by global variables. In: Stoy, J.E. (ed.) Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. pp. 39–53. London, England (Sep 1989)
44. Shivers, O.: Control-flow analysis in Scheme. In: Schwartz, M.D. (ed.) Proceedings of the ACM SIGPLAN 1988 Conference on Programming Languages Design and Implementation. pp. 164–174. Atlanta, Georgia (Jun 1988)
45. Sulzmann, M., Chakravarty, M.M.T., Peyton Jones, S., Donnelly, K.: System F with type equality coercions. In: Necula, G. (ed.) TLDI'07: Proceedings of the Third ACM SIGPLAN Workshop on Types in Language Design and Implementation. pp. 53–66 (Jan 2007)
46. Tarditi, D.: Design and implementation of code optimizations for a type-directed compiler for Standard ML. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, Pensylvania (1997)
47. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: Til: a type-directed optimizing compiler for ml. In: Fischer, C. (ed.) Proceedings of the ACM SIGPLAN 1996 Conference on Programming Languages Design and Implementation. pp. 181–192. ACM, Philadelphia, Pennsylvania (May 1996)
48. Tolmach, A., Oliva, D.P.: From ML to Ada: Strongly-typed language interoperability via source translation. Journal of Functional Programming 8(4), 367–412 (1998)
49. Wells, J.B., Dimock, A., Muller, R., Turbak, F.: A calculus with polymorphic and polyvariant flow types. Journal of Functional Programming 12(3), 183–227 (2002)
50. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (1994)

Verified and Executable Semantics in Coq

Ken Madlener and Sjaak Smetsers

Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands
{k.madlener,s.smetsers}@cs.ru.nl

Abstract. Operational semantics is a well-known tool used to describe the semantics of a programming language. For operational rules in the syntactic well-behaved GSOS format it is possible to generically prove adequacy of operational and denotational semantics. This has been pioneered by Turi and Plotkin in a categorical setting. This paper contributes an implementation of their framework in the theorem prover Coq and presents a formal proof of the adequacy theorem. The main advantages of having a Coq formalization are that it facilitates both formal reasoning about, and experimentation with the semantics of programming languages.

1 Introduction

Operational and denotational semantics are two well-known tools for assigning a formal meaning to programming languages and process algebras. Both are crucial for a complete description of the semantics and should be consistent. Around 15 years ago, Turi and Plotkin [19] developed a framework that unifies both styles of semantics. Using the language of category theory, they managed to strip away language-specific details such as concrete syntax and behavior. Given a set of operational rules, they derived both the operational and denotational semantics from a distributive law corresponding to a set of operational rules. It is imperative that these operational rules adhere to the syntactic, well-behaved *GSOS format* [5].

Turi and Plotkin's work has attracted a great deal of attention since its introduction. Since then, the framework has been extended, for example to deal with general recursion and to support syntax with variable binding; [10] offers an overview. The proofs and even the examples in recent papers which apply this framework are becoming increasingly complex. We therefore believe that it is worth the effort to formalize the framework in a functional language that has theorem proving support.

This paper implements Turi and Plotkin's original work as two Coq programs: one corresponding to the operational semantics, and the other to the denotational semantics, and we verify that these programs (functions) are extensionally equal. This *adequacy proof* is important, as it allows the user to develop meta-theoretical concepts as well as properties of specific programs based on either operational or denotational semantics. Theorem proving approaches

to operational and denotational semantics have traditionally been more *ad hoc*, the consistency is usually proved about a specific language, see e.g. [3, 4]. Moreover, as Coq is in fact a fully-fledged programming language which enables the execution of the semantics provided by the user our implementation facilitates further experimentation.

Implementations of Turi and Plotkin's work in Haskell have been developed by Jaskelioff, Hutton and Ghani in [9] and by Hutton in [7]. An important tenet of most theorem provers, including Coq, is that every function must terminate, otherwise the underlying logic would be inconsistent. This seemingly superficial difference with Haskell has profound implications for the development presented in this paper. In order to satisfy syntactic checks which Coq performs on definitions to guarantee termination, the types representing the syntax and behavior of the language must be carefully chosen carefully. As we will see, Coq's support for dependent types can be put to good use. The semantic domain potentially consists of infinite objects, as shown in the examples provided in this paper. In contrast to Haskell, there is a clear distinction between finite and infinite worlds in Coq. The standard (syntactic) equality of Coq is not general enough for serious proofs about infinite objects. We have based the Coq formalization on the use of *setoids*, i.e. types packaged with a user-defined notion of equality and a proof of well-behavedness of the equality.

This paper is arranged as follows. Sections 2 and 3 serve as an overview of the computational side of Turi and Plotkin's framework, illustrating operational and denotational styles of semantics using a simple language for streams as its running example. To make the content in this paper accessible to readers with no prior experience with Coq, we use Haskell in these two sections. A more involved example featuring non-determinism is provided towards the end of this paper in Section 5. We switch to Coq in Section 4, and explain what setoids are and how they are applied in the equational reasoning within our formalization. We then develop a modular, reusable theory of terms, allowing us to prove properties about the terms "once and for all". We also discuss how to deal with higher-order type constructors that are used in the Haskell specifications, but which cannot be translated directly into Coq. The proof of the adequacy theorem will be given in Section 4. We do this first for a simple rule format and then extend the proof to the GSOS format in Section 5. An overview of related work is provided in Section 6 and finally conclusions are drawn in Section 7.

The reader is expected to have a modest familiarity with category theory. All notions and properties given in this paper, including those specified in Haskell (see Section 2), have been formalized and proven in Coq. The files of the development are available on the web via `http://www.cs.ru.nl/~kmadlene/adequacy/`.

2 A Simple Stream Language

In this section we discuss the relation between operational and denotational semantics similar to the way they arise in the framework of Turi and Plotkin [19].

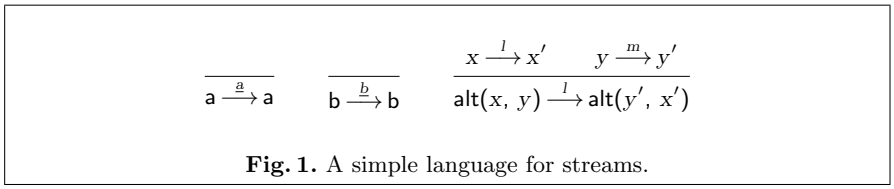


Fig. 1. A simple language for streams.

We use a simple language about streams as our running example; a more involved language featuring non-determinism will be given in Section 5. Since we expect that the reader might not be familiar with Coq, we start with using Haskell as specification language; in Section 4 we switch to Coq.

Consider the simple stream language defined by the operational rules in Figure 2. These rules inductively define a transition relation $\rightarrow \subseteq \mathrm{T} \times L \times \mathrm{T}$, where T are the closed terms and $L := \{\underline{a}, \underline{b}\}$ is the set of labels.. The operations a and b generate the elementary streams $\underline{aaaa}\cdots$ and $\underline{bbbb}\cdots$, and the operation alt, provided with two streams, produces the alternation between them.

We can specify the syntax of the language by a *signature*: a set of function symbols each equipped with a fixed arity. Such a signature is encoded as a functor which we will call the *signature functor*. The signature functor of our stream language, and the terms generated from this signature are:

```
data Σ p = AS | BS | Alt p p
instance Functor Σ where
  fmap f AS = AS
  fmap f BS = BS
  fmap f (Alt p1 p2) = Alt (f p1) (f p2)
data T = App (Σ T)
```

The separation of Σ from T will be important in the rest of the paper. To encode the transition relation we also have to represent the *behavior* of the system. This is again done with a special functor, which we call the *behavior functor*. The data type L corresponds to our label set.

```
data L = A | B
data B p = L :*: p
instance Functor B where
  fmap f (x :*: y) = x :*: f y
```

We can now define a transition system as a $\mathcal{B}$*-coalgebra*, i.e. a pair consisting of an *object* X and a *structure map* $X \rightarrow \mathcal{B}\ X$.

```
OM :: T → B T
OM (App app) = fmap App
  case app of
    AS      → A :*: AS
    BS      → B :*: BS
```

```
Alt p1 p2 → let
                (p1a : ∗ : p1p) = OM p1
                (p2a : ∗ : p2p) = OM p2
            in p1a : ∗ : Alt p2p p1p
```

One may think of *OM* as a *model* of the operational semantics of the language in question, as it specifies for each state what the next step would be. A term can be *run* by (coiteratively) unfolding it to obtain a stream of labels. The streams are actually the *greatest fixpoint* of the present behavior functor. This leads to the following definitions:

```
data N f = N (f (N f))
unfold   :: Functor b ⇒ (x → b x) → x → N b
unfold g = N ∘ fmap (unfold g) ∘ g
run :: T → N B
run = unfold OM
```

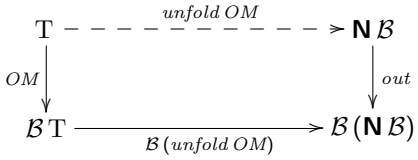
In this paper we will be interested in behavior functors $\mathcal{B}$ that admit a *final coalgebra*. We encode such functors by an additional class constraint containing this (unique) coalgebra named *out*

```
class Functor f ⇒ FinCoAlg f where
  out :: (N f) → f (N f)
instance FinCoAlg B where
  out (N (x : ∗ : y)) = x : ∗ : y
```

Indeed, now we can show

Lemma 1. *The coalgebra out for the functor* $\mathcal{B}$ *is* final.

Thus, there is a unique function making the following diagram[1] commute, and this function is *run* (= unfold OM).



The intuition behind the above diagram is that splitting a label off the stream generated by unfolding *OM* is the same as performing one step and unfolding *OM* on the resulting term. The finality statement in Lemma 1 is actually more general in the sense that *OM* can be replaced by an arbitrary $\mathcal{B}$-coalgebra. See also [8] for more background information about coalgebras.

[1] In the diagrams of this paper we will adopt the 'categorical notation' for functors by writing F instead of *fmap*, for some functor F, i.e., we explicitly indicate the instance type, and leave *fmap* itself implicit.

As in [19] we consider the denotational semantics as a dual version of the operational semantics. The underlying *denotational model* actually operates directly on elements of the semantic domain of our language, i.e. $\mathbf{N}\ \mathcal{B}$. For the present example this means that it prescribes how the operations of the language, when applied to streams, yield new streams. The semantic functions corresponding to the operations of Σ are:

$$
\begin{aligned}
&\mathit{semAS}, \mathit{semBS} :: \mathbf{N}\ \mathcal{B} \\
&\mathit{semAS} = \mathbf{N}\ (A : * : \mathit{semAS}) \\
&\mathit{semBS} = \mathbf{N}\ (B : * : \mathit{semBS}) \\
&\mathit{semAlt} :: \mathbf{N}\ \mathcal{B} \to \mathbf{N}\ \mathcal{B} \to \mathbf{N}\ \mathcal{B} \\
&\mathit{semAlt}\ (\mathbf{N}\ (\mathit{s1h} : * : \mathit{s1t}))\ (\mathbf{N}\ (\_ : * : \mathit{s2t})) = \mathbf{N}\ (\mathit{s1h} : * : \mathit{semAlt}\ \mathit{s2t}\ \mathit{s1t})
\end{aligned}
$$

It is now trivial to define the denotational model:

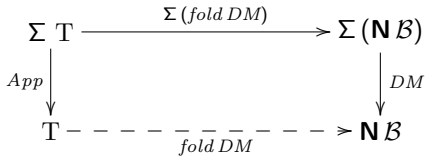
$$
\begin{aligned}
&\mathit{DM} :: \Sigma\ (\mathbf{N}\ \mathcal{B}) \to (\mathbf{N}\ \mathcal{B}) \\
&\mathit{DM}\ \mathit{AS} \quad\quad\quad = \mathit{semAS} \\
&\mathit{DM}\ \mathit{BS} \quad\quad\quad = \mathit{semBS} \\
&\mathit{DM}\ (\mathit{Alt}\ \mathit{s1}\ \mathit{s2}) = \mathit{semAlt}\ \mathit{s1}\ \mathit{s2}
\end{aligned}
$$

Evaluation of terms is dual to *run*: instead of unfolding the coalgebra OM, we *fold* the algebra DM over a given term.

$$
\begin{aligned}
&\mathit{fold} :: (\Sigma\ a \to a) \ \to \mathrm{T} \to a \\
&\mathit{fold}\ h\ (\mathit{App}\ \mathit{app}) = h\ (\mathit{fmap}\ (\mathit{fold}\ h)\ \mathit{app}) \\
&\mathit{eval} :: \mathrm{T} \to \mathbf{N}\ \mathcal{B} \\
&\mathit{eval} = \mathit{fold}\ \mathit{DM}
\end{aligned}
$$

Lemma 2. *The* initial algebra *for the functor* Σ *(previously called in) is App.*

Thus, there is a unique function making the following diagram commute, and this function is *eval* (= *fold DM*).



The adequacy statement for the present example says that executing *run* and *eval* on the same term yields the same stream. With the definitions as they stand a proof would proceed by induction on T, and would be rather ad hoc. A more structured development will be laid out in the rest of this paper, where we use a distributive law, derived from a set of operational rules, as the common source for both operational and denotational models.

3 Turi and Plotkin's Framework

The present section generalizes the example of the previous section.

3.1 Generalized Terms

A more general version of T that does not directly depend on a specific signature is obtained by making T parametric in the signature. Moreover, to allow for *open terms* (used later on to represent meta-variables in the operational rules) a *Var* constructor has been added.

```
data T f x = Var x | App (f (T f x))
```

It is straightforward to make these terms into a Functor, by defining:

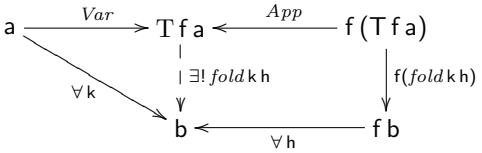
```
instance Functor f ⇒ Functor (T f) where
  fmap f (Var x)     = Var (f x)
  fmap f (App app) = App (fmap (fmap f) app)
```

Likewise, we generalize the *fold* defined in Section 2.

```
fold :: Functor f ⇒ (a → b) → (f b → b) → (T f a) → b
fold k h (Var x)     = k x
fold k h (App app) = h (fmap (fold k h) app)
```

The *fold* operation provides a recursive definition principle that avoids explicit recursion (e.g. see also [13]): one only has to specify a mapping of the variables $k :: a \to b$, and an algebra $h :: f\ b \to b$. This result is attributed to the following lemma.

Lemma 3. *Let $k :: a \to b$, and $h :: f\ b \to b$. Then fold k h is the unique function making the following diagram commute:*



If we choose the empty type for a, we obtain the set of closed terms as in Section 2. In HASKELL[2]:

```
data Empty
empty :: Empty → a
empty _ = ⊥
```

Now the left part of the diagram can then be ignored and the remaining square says precisely that $App :: f\ (\mathrm{T}\ f\ Empty) \to \mathrm{T}\ f\ Empty$ is the initial algebra for functor f. Observe that the diagram in Lemma 2 can be obtained by taking $\mathbf{N}\ \mathcal{B}$ for b, and Σ for f.

[2] Not standard HASKELL; requires the compiler option `-XEmptyDataDecls` of GHC.

3.2 Distributive Laws

It is common in the field of process algebra to enforce well-behavedness of the semantics through syntactic formats on the operational rules. In particular, the GSOS format [5] is important, as it enjoys the adequacy property while being sufficiently liberal to express a wide variety of semantics. Lenisa, Power and Watanabe [12] (refining the work of Turi and Plotkin) have shown that the GSOS format corresponds precisely to a *distributive law of a monad over a functor*. We will first consider simpler distributive laws, ones that distribute a functor over another functor, leaving GSOS to Section 5. Distributive laws of the latter type are functions of the type $\Sigma\ (\mathcal{B}\ a) \to \mathcal{B}\ (\Sigma\ a)$ that happen to be natural transformations (see Section 4.1).

We will replace the the operational as well as the denotational model introduced in the previous section with models that are derived from a common distributive law SM, which corresponds to the operational rules.

$$
\begin{array}{ll}
SM :: (\Sigma\ (\mathcal{B}\ a)) \to \mathcal{B}\ (\Sigma\ a) & \\
SM\ AS & = A : * : AS \\
SM\ BS & = B : * : BS \\
SM\ (Alt\ (s1h : * : s1t)\ (s2h : * : s2t)) & = s1h : * : Alt\ s2t\ s1t
\end{array}
$$

The function SM takes an operation (an element from the signature) as argument. In the case of Alt this operation is applied to two arguments that both consist of a pairing of an action and a variable. This corresponds to premise of a rule. The result is a pairing of an action and an operation applied to variables, corresponding to the target of the conclusion of each rule. The polymorphism in a ensures that SM does not depend on a concrete choice of the set of variables.

In summary, the type of SM says that each operation in the language, as it is applied to behaviors on the variables, yields a behavior on an operation applied to variables. Therefore, SM can be seen as the *(semantic) model* of the language.

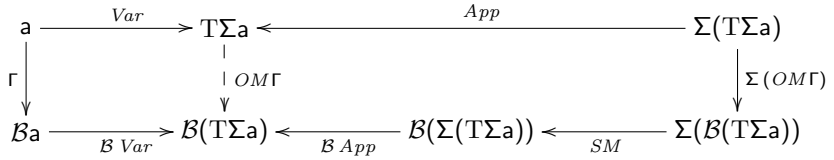
As said before, each set of operational rules has its own semantic model SM. For each concrete SM it should be verified that it is a natural transformation; see Section 4.

3.3 Operational and denotational models

In the standard relational approach to operational semantics, the validity of a transition step is proved by the construction of a derivation tree. The nodes correspond to applications of the operational rules, and the leafs correspond applications of the hypotheses.

We can mimic this with the help of the definition principle for terms (the *fold* operation) combined with the semantic model. Suppose that we have a map $\Gamma :: a \to \mathcal{B}\ a$, representing the *behavior environment*: the hypotheses about the variables in the premises. If we encounter an application of an operation, then

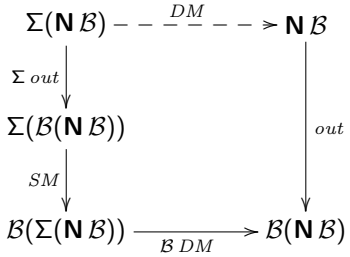
we apply SM, and if we encounter a variable we apply Γ. In a diagram,



which concretely is

$$OM :: (a \to \mathcal{B}\ a) \to \mathrm{T}\ \Sigma\ a \to \mathcal{B}\ (\mathrm{T}\ \Sigma\ a)$$
$$OM\ \Gamma = fold\ (fmap\ Var \circ \Gamma)\ (fmap\ App \circ SM)$$

The denotational model can be obtained in a dual fashion, i.e. by unfolding the semantic model. Recall that the dual of App is out.



Concretely,

$$DM :: \Sigma\ (\mathbf{N}\ \mathcal{B}) \to \mathbf{N}\ \mathcal{B}$$
$$DM = unfold\ (SM \circ fmap\ out)$$

The denotational model operates directly on elements of the semantic domain. It tells how the operations of the language, applied to denotations, form new denotations. Observe that the set of hypotheses does not play a role in the denotational model, but will come into play when we construct the evaluation function *eval*. Running a term according to the operational model, and evaluating a term according to the denotational model is defined in same way as in Section 2:

$$run, eval :: (a \to \mathcal{B}\ a) \to (\mathrm{T}\ \Sigma\ a) \to \mathbf{N}\ \mathcal{B}$$
$$run\ \Gamma = unfold\ (OM\ \Gamma)$$
$$eval\ \Gamma = fold\ (unfold\ \Gamma)\ DM$$

The distributivity property of SM will be needed prove adequacy of *run* end *eval*. There are many sensible operational rules that do not fit in the format described by rules in the "plain" format of SM. For example, suppose we extend our simple stream language with an operation zip that merges two input streams. This operation differs slightly from alt in the sense that at each transition it does

not discard the head element of the second stream. The corresponding transition rule is:

$$\frac{x \xrightarrow{l} x'}{\mathsf{zip}(x,\, y) \xrightarrow{l} \mathsf{zip}(y,\, x')}$$

If we try to encode this rule in our semantic model, for instance by adding the following alternative to *SM*:

$$SM\ (Zip\ (s1h : * : s1t)\ s) = s1h : * : Zip\ s\ s1t$$

then the model does not type check anymore. The main problem is that variables used on both the left- and right-hand sides should receive a polymorphic type, which is not the case for the variable s. Also replacing s by a pattern match, as in the case for *Alt* will not work because then we have to reconstruct the first argument of *Zip* on the right-hand side out of the constituents. In Section 5 we discuss the more liberal *GSOS* rule format admitting operations like *Zip*.

4 Formalizing semantics in Coq

In this section we will discuss the formalization of the semantic framework introduced in the preceding section in Coq. However, due to the different natures of Haskell and (the specification language of) Coq not all notions and concepts can be translated directly. More specifically, higher-order type constructors and (possibly) infinite data structures will require a special treatment in Coq. But first we will take a closer look on how to deal with infinite objects.

4.1 Equational Reasoning with Setoids

Infinite objects, as in most theorem provers, live in a world separate from finite objects, and do not adhere to Coq's standard notion of equality. One often works instead with *bisimulation*, a weaker notion of equality on infinite objects. Coq does not support user-defined extensions of its standard notion of equality (i.e. quotient types) as it would endanger the decidability of type checking. To overcome this issue, it is common practice to work with *setoids*, types that are packaged with a user-defined notion of equality and a proof of well-behavedness of the equality. The commuting diagrams in Section 2 use bisimulation as the underlying notion of equality in the Coq formalization. Finally, *setoid morphisms* are functions whose domain and codomain are setoids and respect those equalities.

The recent addition of type-classes to Coq [16] enables the use of canonical names for standard mathematical notation. These type classes are first-class as they are powered by proof search and implicit arguments. Declaring instances of the *Equiv*, *Setoid* and *Setoid* Morphism type classes enables fluent rewriting modulo setoid equality in proofs. In this paper we tacitly overload the canonical name "=" with setoid equality.

First, we introduce setoid counterparts for the standard categorical notions of functor and natural transformation. The *setoid functor* is taken from the MathClasses library [20, 17]. It consists of an object map M and two classes: a class containing a function map and a class carrying properties about the object map and the function map[3].

Class *SFmap* $(M : Type \to Type) :=$
  $sfmap : \forall\ \{X\}\ \{Y\}\ (f : X \to Y), M\ X \to M\ Y$
Class *SFunctor* $(M : Type \to Type)$
  $`\{\forall `\{Equiv\ X\}, Equiv\ (M\ X)\}`\{SFmap\ M\} := \{...\}$

For reasons of space, we omit the full definition of *SFunctor*. In brief, it carries two sanity properties stating that the object map makes a setoid on X into a setoid on $M\ X$, and that the function map is a setoid morphism in its function argument (allowing us to rewrite equivalent functions with one another), and the following two familiar properties about the function map[4].

$$sfmap_{\mathsf{M}}\ id = id$$
$$sfmap_{\mathsf{M}}\ (\mathsf{f} \circ \mathsf{g}) = sfmap_{\mathsf{M}}\ \mathsf{f} \circ sfmap_{\mathsf{M}}\ \mathsf{g}$$

Moreover, the types in the diagrams will be written in a right-associative manner, and type arguments of functions are written as subscript, as above.

Given two object maps M and N, one can define a family of functions:

Notation $M \Rightarrow N := \forall\ X, M\ X \to N\ X$ (*at level* 90, *right associativity*)

The family of functions $\eta : M \to N$ is a *setoid natural transformation* if η_{X} is a setoid morphism whenever X is a setoid, and if it satisfies a commutation law:

$$\eta_{\mathsf{Y}} \circ sfmap_{\mathsf{M}}\ \mathsf{f} = sfmap_{\mathsf{N}}\ \mathsf{f} \circ \eta_{\mathsf{X}} \tag{1}$$

Again for reasons of brevity, we no not include the corresponding type class definition *SNatural*.

4.2 Encoding terms

Attempting to encode terms as is done in the Haskell code in Section 2 directly in Coq will result in an error, as this definition violates Coq's syntactic check for positivity (which guarantees termination of structurally recursive functions).

We can bypass this issue by exploiting the fact that signatures of binding-free languages have a fairly simple structure. That is, a term on such a signature is

[3] Unlike Haskell, Coq admits variable names starting with an uppercase letter. Furthermore, the backtick causes Coq to automatically generalize missing variables.

[4] To avoid confusion, we use the convention to add the object map (M in this case) as a subscript to the function map (here *sfmap*). We will use this notation also for other mappings, particularly in properties and diagrams

essentially a general tree, in which each parent node has an arbitrary number of child nodes, dictated by the arity of the operation corresponding to the parent node. A leaf of the tree is either a parent node with zero children nodes, or a variable, if we consider open terms.

We introduce an auxiliary type family *fin* representing the set of natural numbers up to some natural number.

```
Inductive fin : nat → Set :=
  | first : ∀ n, fin (S n)
  | next : ∀ n, fin n → fin (S n)
```

The signature is nothing more than an assignment of an arity to each of the language's operations, of which there are *size* many.

```
Variable size : nat
Variable ar : fin size → nat
Definition Σ X := {i : fin size & vector X (ar i)}
```

A parent node in the tree is described by a dependent pair, consisting of the index i corresponding to an operation together with a vector of length the arity of the operation (this is essentially a richly typed version of *list*, akin to *fin*). One can think of the notation "{_ & _}" as a type-theoretic variant of set comprehension. Dependent pairs can be crafted using the notation "(_ & _)". Finally, we can provide a type for the open terms. The parameter X represents variables.

```
Inductive T X := var : X → T | app : Σ T → T
```

In Section 2 we defined a instance of Σ for the functor class. The adjusted instance for the setoid version, *SFmap*, is:

```
Instance : SFmap Σ :=
λX Y (f : X → Y) (σ : Σ X) →
  match σ with (s & v) ⇒ (s & map f v) end
```

The proofs of the setoid functor properties are contained in a separate instance of *SFunctor*. Now, we can show that the following property holds:

Lemma 4. *Σ is a setoid functor.*

The proof of Lemma 3 is obtained by use of the full (dependently-typed) induction principle for T. The full principle has been used in our development to prove the properties in this section by induction on the structure of T.

In the remainder of this section we show that T is a monad in the categorical sense. To this end, we need to show that it has a *unit* (which is simply $var : X \to \mathrm{T}\ X$) and a multiplication operation

```
Definition join X : T (T X) → T X := fold id (app_X)
```

and show that it satisfies the two standard coherence conditions of monads.

Lemma 5. *The functor* T *is a monad, i.e. the following identities hold:*

$$join_{\mathsf{X}} \circ sfmap_{\mathrm{T}}\, join_{\mathsf{X}} = join_{\mathsf{X}} \circ join_{\mathrm{T}\,\mathsf{X}}$$
$$join_{\mathsf{X}} \circ sfmap_{\mathrm{T}}\, var_{\mathsf{X}} = join_{\mathsf{X}} \circ var_{\mathrm{T}\,\mathsf{X}} = id$$

We call this monad the *term monad*.

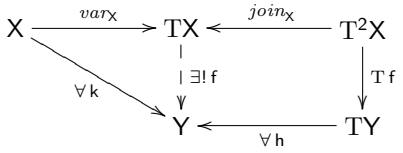
It is a well-known fact from category theory that the category of Σ-algebras is isomorphic to the category of *algebras for the term monad*. These are "plain" algebras h for the functor T, with two additional properties:

$$\mathsf{h} \circ var_{\mathsf{Y}} = id$$
$$\mathsf{h} \circ sfmap_{\mathrm{T}}\, \mathsf{h} = \mathsf{h} \circ join_{\mathsf{Y}}$$

A T-algebra homomorphism is a homomorphism of the underlying algebra.

We end this section by giving an auxiliary definition principle for open terms, similar to Lemma 3.

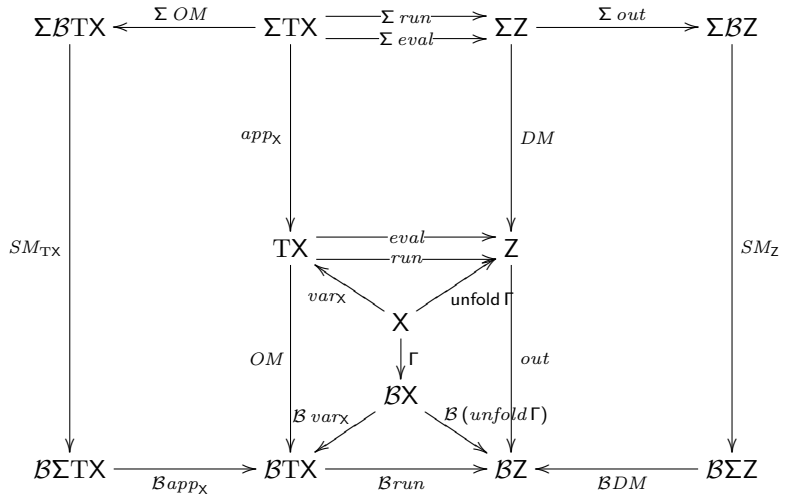
Lemma 6. *Let* $\mathsf{k} : \mathsf{X} \to \mathsf{Y}$*, and* $\mathsf{h} : \mathrm{T}\,\mathsf{Y} \to \mathsf{Y}$*. Set* $\mathsf{f} = fold\ \mathsf{k}\ (\mathsf{h} \circ app_{\mathsf{Y}} \circ sfmap_{\Sigma}\ var_{\mathsf{Y}})$*. Then* f *is the unique function making the following diagram commute:*



We have now set up a theory for syntax. Similarly, we could develop a theory for behavior. We do not pursue this goal for two reasons. First, since our presentation is essentially a deep embedding of SOS rules, most of it hinges on a structured encoding of the terms. Secondly, one may expect that more variation in the behavior is desired to model phenomena such as time or probability (see [1]; see also [10] for an overview). Moreover, finality proofs such as Lemma 1 can be tricky to carry out in Coq due to guardedness restrictions that it puts on corecursive definitions.

4.3 Adequacy theorem (for rules in plain format)

We have introduced a structured way to obtain the operational and denotational models from a single semantic model. Now we are ready to prove the adequacy theorem. Suppose that we have a coalgebra $\Gamma : X \to \mathcal{B}\ X$ representing the hypotheses on the variables. We can combine the diagrams of the operational and denotational models as indicated in the following diagram.

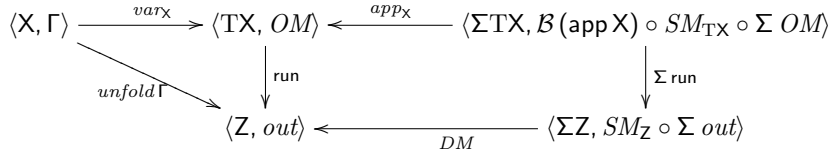


The following theorem holds for *open* terms, which is a mild generalization of what has been presented in the literature [19, 10, 1].

Theorem 1 (Adequacy).

$$run = eval.$$

Proof. Consider the following diagram in the category of $\mathcal{B}$-coalgebras. That is, the objects are pairs (consisting of the object and the structure map) and the arrows are coalgebra homomorphisms.



Except for the arrow $\Sigma\ run$, it is trivial to see that each of the arrows are in fact coalgebra homomorphisms, as it can be directly read off the complete diagram above. To show this for $\Sigma\ run$, one uses naturality of SM and the fact that run is a coalgebra homomorphism. Commutativity of the diagram follows from the finality of out. The theorem then follows from applying the forgetful functor to the above diagram and the definition principle used to define $eval$. □

5 Rules in GSOS Format

So far we have considered "plain" laws that distribute a signature functor over a behavior functor. There are many sensible operational rules that do not fit in this format.

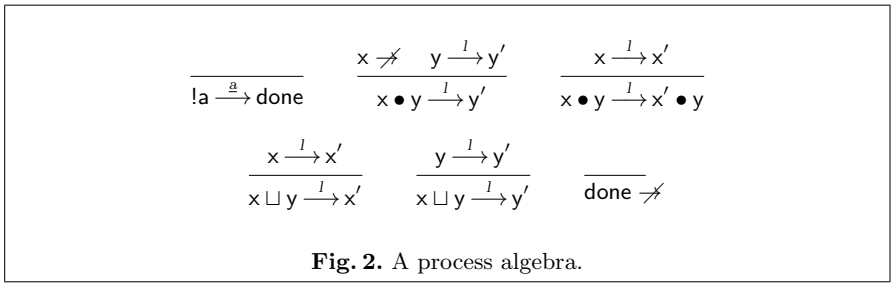


Fig. 2. A process algebra.

Consider the process algebra language of Figure 5. The parallel composition operation "$\sqcup$" and the possibility of a state not having any outgoing transitions clearly make this a non-deterministic language. We can reuse the fin type family from Section 4.2 to define the behavior functor.

Definition $\mathcal{B}\ X := \{\, n : \mathit{nat}\ \&\ \mathit{fin}\ n \to A\ \times\ X \,\}$

The object of the corresponding final are the cotrees (also known as *rose trees*):

CoInductive $\mathit{cotree} := \mathit{node} : B\ \mathit{cotree} \to \mathit{cotree}$

Just like for the streams, we can prove the following lemma.

Lemma 7. *cotree is the object of the final coalgebra for the functor* $\mathcal{B}$*.*

We remark that it is perhaps more straightforward to set $\mathcal{B}\ X := \mathit{list}\ (A\ \times\ X)$, but this does not seem to work. We were unable to provide a guarded corecursive definition of *unfold* for that choice of $\mathcal{B}$.

We run into problems when we attempt to encode the rules for the sequencing operation "$\bullet$" as a plain distributive law. The first rule has the variable y' as the outgoing state of its conclusion, which is not in the required form of an operation applied to some variables. Thus we would like to have the liberty of writing terms for outgoing states, in which case we can apply $\mathit{var}\ y'$. The second rule refers to y in the outgoing state of the conclusion, which again is not possible in the plain format. We need the arguments of the operation to be pairings of both the variable and the behavior on that variable. The following type will allow us to encode our process language:

$$\rho : \Sigma \circ \mathcal{D} \Rightarrow \mathcal{B} \circ \mathrm{T}$$

where we have used the notation $\mathcal{D}\ X := X\ \times\ \mathcal{B}\ X$. We call natural transformations of this type *rules in abstract GSOS format*[5].

The implementation of the corresponding rules is fairly complicated to read but we include it for reference.

[5] Literature on process algebra often refers to the specialization of $\mathcal{B}$ being a finite powerset as the the GSOS format. We call this format *abstract GSOS* to avoid confusion.

```
Definition ρ : Σ ∘ 𝒟 ⇒ ℬ ∘ T =
  λX σ →
    match σ with
      | ! a       ⇒ (1 & (λ_ → (a, app done)))
      | x ⊔ y ⇒ (_ & merge ([id, var] ∘ projT2 (snd x))
                           ([id, var] ∘ projT2 (snd y)))
      | (_, (0 & _)) • (_, b)   ⇒ sfmap ℬ var b
      | (_, b)       • (y, _)   ⇒ sfmap ℬ (λx' → app (var x' • var y)) b
      | done   ⇒ (0 & case0 _)
    end
```

Here *merge* and *case0* have the obvious meaning (we omit the code):

```
Definition merge'(f : fin n → X)'(g : fin m → X) : fin (n + m) → X
Definition case0 (f : fin 0 → Type) (i : fin 0), f i
```

5.1 From GSOS rules to a distributive law

The symmetry of the (co)domains was vital to for the adequacy theorem. The rules ρ have to undergo a two-step transformation to obtain a distributive law of T over $\mathcal{D}$. First expand ρ's codomain:

```
Definition τ : Σ ∘ 𝒟 ⇒ 𝒟 ∘ T :=
  λX σ → (app_X (sfmap_Σ (var_X ∘ fst) σ), ρ σ)
```

To obtain *SM* we apply *fold*.

```
Definition SM : T ∘ 𝒟 ⇒ 𝒟 ∘ T :=
  λX → fold (sfmap_ℬ (var_X)) (sfmap_ℬ (join_X) ∘ τ_(T X))
```

A general proof in the Coq development shows that the definition principle for terms yields natural transformations. From this fact, together with the assumption that ρ is a natural transformation, it is straightforward to show that *SM* is natural as well.

The obtained distributive law, which distributes the term monad over the functor $\mathcal{D}$, enjoys more structure than the plain distributive laws.

Proposition 1. *The following two identities hold:*

$$SM_{\mathsf{X}} \circ var_{\mathcal{D}\,\mathsf{X}} = sfmap_{\mathcal{D}}\, var_{\mathsf{X}}$$
$$SM_{\mathsf{X}} \circ join_{\mathcal{D}\,\mathsf{X}} = sfmap_{\mathcal{D}}\, join_{\mathsf{X}} \circ SM_{\mathrm{T}\,\mathsf{X}} \circ sfmap_{\mathrm{T}}\, SM_{\mathsf{X}}.$$

The intuition behind the second identity is that applying *SM* to a joined term is the same as applying *SM* to both the inner and outer term and then joining the result. The previous proposition is a key ingredient in proving the next two lemmas.

5.2 Adequacy theorem for rules in GSOS format

Recall that we used the definition principle of terms to obtain an operational model from the plain distributive laws. We repeat this construction for our new distributive laws, with the difference that we use the alternative definition principle. Before we can do so, we need to verify the following fact:

Lemma 8. *$sfmap_{\mathcal{D}}\, join_{\mathsf{X}} \circ SM_{\mathrm{T}\mathsf{X}}$ is an algebra for the term monad.*

The $\mathcal{D}$-coalgebras are isomorphic to the $\mathcal{B}$-coalgebras, and it is straightforward to verify that if $\langle \mathsf{Z}, \mathit{out}\rangle$ is a final $\mathcal{B}$-coalgebra, then $\langle \mathsf{Z}, \langle \mathit{id}, \mathit{out}\rangle\rangle$ is a final $\mathcal{D}$-coalgebra. Hence, the denotational model for GSOS rules and *run_GSOS* can be obtained by finality, analogous to Section 3.2. We obtain *eval_GSOS* by making use of the alternative definition principle for terms, and thus we need to verify the following fact:

Lemma 9. *DM is an algebra for the term monad.*

We have now arrived at a situation that is entirely analogous to Section 4.3. That is, in the main diagram $\mathcal{B}$ should be replaced with $\mathcal{D}$, Σ replaced with T, and *app* replaced with *join*. The rest of the adequacy theorem for GSOS rules is along the lines of the proof of Theorem 1.

Theorem 2 (Adequacy for GSOS rules).

$$\mathit{run\_GSOS} = \mathit{eval\_GSOS}.$$

6 Related Work

The work of Turi and Plotkin [19] proves the adequacy result for the more general situation of a distributive law of a monad over a comonad. Although these laws provide the most abstract perspective of well-behaved rules, they have not yet been applied in concrete studies of rule formats [10]. Lenisa, Power and Watanabe [11] show that rules in the abstract GSOS format correspond precisely to distributive laws of a free monad over a cofree copointed functor.

Bartels' PhD thesis [1] provides an elaborate overview of rule formats and their categorical meaning. The aforementioned papers are part of a field called *bialgebraic semantics*, where bialgebras, which inherit both an algebra and a coalgebra structure, play an important role.

An implementation of Turi and Plotkin's work [19] has been developed by Hutton [7] in Haskell and extended for modularity by Jaskelioff, Ghani and Hutton [9]. Both implementations define the terms and the object of the final coalgebra (i.e. streams and cotrees in this paper) as the greatest fixpoint of a functor. Analogous definitions are not possible in Coq as it would interfere with the ability to guarantee that each function terminates, required by the underlying logic.

Niqui [14] extends the class of productive specifications definable in Coq by developing the λ-coiteration scheme in Coq. In the further work section of his paper he mentions that adding monadic, pointed or cofree structure on the bialgebraic nature of λ-coiteration can help to build even more powerful schemes.

7 Conclusions and Further Work

We have shown how operational and denotational semantics can be obtained in a structured way in Coq, if one provides an encoding of operational rules in the syntactic GSOS format. The adequacy theorem, which has been formally proved, says that these functions are extensionally equivalent. Our work allows one to both execute semantics inside Coq and enables formal reasoning. We therefore expect that it will be useful to others who wish to work in the field of bialgebraic semantics.

Further work on the formalization is needed to support syntax with variable binding. Work in this direction has already been done by Fiore, Plotkin and Turi [6], but an implementation does not yet exist. A modular theory of terms was presented in this paper, the development of a similar general theory on the side of behavior is left for future work. Another interesting direction of further work would be to develop a theory for language extensions through modular operational rules [9].

Acknowledgements The authors wish to thank Robbert Krebbers for his help proving Lemma 7.

References

1. F. Bartels. *On generalised coinduction and probabilistic specification formats.* PhD thesis, CWI, Amsterdam, April 2004.
2. G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *J. Funct. Program.*, 13(2):261–293, 2003.
3. Y. Bertot, V. Capretta, and K. D. Barman. Type-theoretic functional semantics. In *TPHOLs*, volume 2410 of *LNCS*, pages 83–98. Springer, 2002.
4. Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin, editors. *Theorem proving support in programming language semantics*, chapter 15, pages 337–361. Cambridge University Press, 2009.
5. B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, 1995.
6. M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. 14^{th} LICS Conf.*, pages 193–202. IEEE, 1999.
7. G. Hutton. Fold and unfold for program semantics. In *In Proc. 3rd ACM SIGPLAN ICFP*, pages 280–288. ACM, 1998.
8. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.
9. M. Jaskelioff, N. Ghani, and G. Hutton. Modularity and implementation of mathematical operational semantics. *Electr. Notes Theor. Comput. Sci.*, 229(5):75–95, 2011.
10. B. Klin. Bialgebras for structural operational semantics: An introduction. *Theor. Comput. Sci.*, 412(38):5043–5069, 2011.
11. M. Lenisa, J. Power, and H. Watanabe. Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads. *Electr. Notes Theor. Comput. Sci.*, 33:230–260, 2000.

12. M. Lenisa, J. Power, and H. Watanabe. Category theory for operational semantics. *Theor. Comput. Sci.*, 327(1-2):135–154, 2004.
13. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144. Springer-Verlag New York, Inc., 1991.
14. M. Niqui. Coalgebraic reasoning in Coq: Bisimulation and the λ-Coiteration scheme. In *TYPES*, volume 5497 of *LNCS*, pages 272–288. Springer, 2009.
15. G. D. Plotkin. A structural approach to operational semantics. *J. of Logic and Algebraic Programming*, 60-61:17–139, 2004.
16. M. Sozeau and N. Oury. First-class type classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008.
17. B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *MSCS*, 21:1–31, 2011.
18. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, 2010. `http://coq.inria.fr`.
19. D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Proc. 12th LICS Conf.*, pages 280–291. IEEE, Computer Society Press, 1997.
20. E. van der Weegen, B. Spitters, R. Krebbers, M. Sozeau, T. Prince, and J. Herold. *Math Classes, Coq Library for basic mathematical structures*, 2011. `http://coq.inria.fr/cocorico/MathClasses`.

Dependently-typed programming in scientific computing

Cezar Ionescu[1] and Patrik Jansson[2]

[1] Potsdam Institute for Climate Impact Research
[2] Chalmers University of Technology

1 Extended abstract

In 2006, Herbert Gintis [2] announced the discovery of a mechanism that would explain price formation and disequilibrium adjustment without requiring the presence of a central authority or omniscience on part of the agents, as is currently assumed in mainstream economics. Gintis' results were, as he put it "empirical rather than theoretical: we have created a class of economies and investigated their properties for a range of parameters." They were obtained by computer simulations. Due to the importance of this result, two groups of researchers, one at PIK, the other in Chalmers [1], independently attempted to do something which should perhaps be routine, but is hardly ever done: to re-implement the model described in the paper and reproduce the results. After initial attempts failed and Gintis graciously provided the source code, both groups discovered several ways that his implementation diverged from the description in the paper, only one of which could be called a "bug". Much more problematic was the ambiguity left open by the model description given in the paper, which consisted of a mixture of prose and mathematical equations.

The example of the Gintis model was chosen because it is well documented in recent literature, not because it is unique. It is quite typical for scientists to believe that the mathematical equations used to develop a model are sufficient specification for the implementation of that model, but that is rarely the case. Discretizations, approximations, choices of integration methods, and many other similar steps come between the mathematical description and the program. This is a gap that must be bridged if we are to be able to check correctness of implementations, re-implement models, or replicate results.

Sooner or later, everyone who considers this problem comes to constructive mathematics and Martin-Löf's type theory, which seems to be made to order for this purpose. Here is for example a quote from the programmatic article "Constructive Mathematics and Computer Programming" (Martin-Löf, 1982):

> Now, it is the contention of the intuitionists (or constructivists, I shall use these terms synonymously) that the basic mathematical notions, above all the notion of function, ought to be interpreted in such a way that the cleavage between mathematics, classical mathematics, that is, and programming that we are witnessing at present disappears.

Specifications are also mentioned explicitly:

> [Type theory] provides a precise notation not only, like other programming languages, for the programs themselves but also for the tasks that the programs are supposed to perform. Thus the correctness of a program written in the theory of types is proved formally at the same time as it is being synthesized.

The ideal of correctness put forward here is very enticing, assuming that type theory has the expressive power to formulate the usual mathematical concepts which modelers use as specifications. In the next section we show that this assumption is indeed justified. Together with economists at PIK, we have formalized basic building blocks of economic theory, used in almost all economic models today, concepts such as Pareto efficiency, Walrasian equilibrium, Nash equilibrium, and a host of others, together with the relations between them (for example, Walrasian equilibria are Pareto efficient). The resulting formalizations are pleasantly close to the mathematical formulations the modelers are used to, so we can hope they could use them in specifications. Moreover, as it often happens, the effort to formalize improves ones understanding and can lead to discovery of errors (or at least dubious formulations) even in such elementary contexts.

The bad news is most of these concepts are classical in nature: economics is currently a non-constructive theory (and even the so-called "computable general equilibrium models" turn out to be non-computable). Therefore, the specifications turn out to be non-implementable and the distance between the mathematics and the programming is still there.

But, as we argue in the rest of the paper, we are now in a better position to decrease it. First, the constructively valid parts can be separated from the non-constructive ones and can be implemented in the verified way that corresponds to Martin-Löf's ideal. For example, non-linear optimization is in most cases non-computable, but it is computable over finite (or at least "reasonably-sized") sets. Second, the typical approximations going from the mathematical descriptions to the implementations can be made completely explicit, so that ambiguities such as in the Gintis description do not arise, and their use can then be at least partially checked. For example, continuous optimization problems are routinely replaced with discrete ones, but the assumption that the discrete problem has the same solution as the continuous one is rarely stated (perhaps because it rarely holds). In such cases, we can *postulate* this property, forcing the type checker to accept the solution of the discrete problem. The advantage is that the type checker can then ensure that we do the correct thing with this (potentially incorrect) solution. (A similar example arises every time one uses an external, non-verified library function.) Third, the postulated properties are clearly highlighted points for further improvement. For example, one could replace the requirement that the discrete problem has the exact same solution with the continuous one by the weaker (and in many applications actually achievable) requirement that the distance between them is smaller than a given tolerance. Finally, we would be in a better position to communicate with the constructive mathematics community

and to take advantage of advances made in the area of numerical methods (such as might come out, for example, from the ForMath project).

We close the paper with a discussion of some consequences of this approach. First, there are immediate constraints on the implementation of type theory to be used. A postulational mechanism that still allows the compilation and execution of programs is essential, as is a foreign function interface that allows users to take advantage of the standard numerical libraries they are used to. Moreover, there are consequences on the programming style. Two styles have emerged in the relatively recent dependently-typed programming community, which can be named after the two major textbooks where they are presented: the Nordström et al. style suggests developing the proof *within* the implementation, the Thompson style advocates developing the proof *alongside* the implementation. The former is easier for proofs, the latter seems easier for newcomers and apparently fits better here. Unfortunately, adopting the Thompson style leads to difficulties in doing the proofs, and in the absence of a powerful reflection mechanism, sometimes prevents one from doing the proofs altogether.

References

1. P. Evensen and M. Märdin. An Extensible and Scalable Agent-Based Simulation of Barter Economics. Master Thesis. Chalmers Techn. Univ. & U. Gothenburg., 2009.
2. H. Gintis. The emergence of a price system from decentralized bilateral exchange. *The B.E. Journal of Theoretical Economics*, 6(1):13, 2006.

Applications of Reflection in Agda

Paul van der Walt and Wouter Swierstra

paul@denknerd.org, W.S.Swierstra@uu.nl
Department of Computer Science, Utrecht University

Abstract. This paper explores the recent addition to Agda enabling *reflection*, in the style of Lisp, MetaML, and Template Haskell. It illustrates several applications of reflection that arise in dependently typed programming.

1 Introduction

The dependently typed programming language Agda [1,2] has recently been extended with a *reflection mechanism* for compile time meta programming in the style of Lisp [3], MetaML [4], Template Haskell [5], and C++ templates [6]. Agda's reflection mechanisms make it possible to convert a program fragment into its corresponding abstract syntax tree and vice versa. In tandem with Agda's dependent types, this provides promising new programming potential.

This paper starts exploring the possibilities and limitations of this new reflection mechanism. It describes several case studies, exemplative of the kind of problems that can be solved using reflection. More specifically it makes the following contributions:

- This paper documents the current status of the reflection mechanism. The existing documentation is limited to a paragraph in the release notes [7] and comments in the compiler's source code. In Section 2 we give several short examples of the reflection API in action.
- This paper illustrates how to use Agda's reflection mechanism to automate certain categories of proofs (Section 3). The idea of *proof by reflection* is certainly not new, but still worth examining in the context of this new technology.
- In the final version of this paper, we will also show how to guarantee *type safety of meta-programs*. To illustrate this point, we will develop a type safe translation from the simply typed lambda calculus to combinatory logic.
- Finally, the final version will also discuss some of the limitations of the current implementation of reflection.

The code and examples presented in this paper all compile using the latest version of Agda 2.3.0.1 and are available on github.[1]

[1] http://www.github.com/toothbrush/reflection-proofs

2 Reflection in Agda

Agda's reflection API defines several data types which represent terms, types, and sorts. These definitions take into account various features, including hidden arguments and computationally irrelevant definitions. An overview of the core data types involved has been included in Figure 1. In addition to these data types that represent *terms*, there is some support for reflecting *definitions* as opposed to terms.

There are several new keywords that can be used to quote and unquote Term values: quote, quoteTerm, quoteGoal, and unquote. The quote keyword allows the user to access the internal representation of any identifier. This internal representation can be used to query the type or definition of the identifier. The examples discussed in this paper will not illustrate quote. The other quotation forms, quoteTerm and quoteGoal, will be used.

The easiest example of quotation uses the quoteTerm keyword to turn a fragment of concrete syntax into a Term data type. Note that the quoteTerm keyword reduces like any other function in Agda. As an example, the following unit test type checks:

```
example : quoteTerm (λ x → x) ≡ lam visible (var 0 [])
example = refl
```

Furthermore, quoteTerm type checks and normalizes its term before returning the required Term, as the following example demonstrates:

```
example : quoteTerm ((λ x → x) 0) ≡ con (quote Data.Nat.ℕ.zero) []
example = refl
```

The quoteGoal is slightly different. It is best explained using an example:

```
exampleQuoteGoal : ℕ
exampleQuoteGoal = quoteGoal e in {!!}
```

The quoteGoal keyword binds the variable e to the Term representing the type of the current goal. In this example, the value of e in the hole will be def ℕ [], i.e., the Term representing the type ℕ.

The unquote keyword converts a Term data type back to concrete syntax. Just as quoteGoal and quoteGoal, it type checks and normalizes the Term before it is spliced into the program text.

3 Proof by Reflection

The idea behind proof by reflection is simple: given that type theory is both a programming language and a proof system, it is possible to define functions that compute proofs. Reflection is an overloaded word in this context, since in programming language technology reflection is the capability of converting some

```
postulate Name : Set

-- Arguments may be implicit, explicit, or inferred
data Visibility : Set where
  visible hidden instance : Visibility

-- Arguments can be relevant or irrelevant.
data Relevance : Set where
  relevant irrelevant : Relevance

-- Arguments.
data Arg A : Set where
  arg : (v : Visibility) (r : Relevance) (x : A) → Arg A
-- Terms.
mutual
  data Term : Set where
    -- A bound variable applied to a list of arguments
    var  : (x : ℕ) (args : List (Arg Term)) → Term
    -- Constructor applied to a list of arguments
    con  : (c : Name) (args : List (Arg Term)) → Term
    -- Identifier applied to a list of arguments
    def  : (f : Name) (args : List (Arg Term)) → Term
    -- Lambda abstraction
    lam  : (v : Visibility) (t : Term) → Term
    -- Dependent function types
    pi   : (t₁ : Arg Type) (t₂ : Type) → Term
    -- Sorts
    sort : Sort → Term
    -- Anything else
    unknown : Term

  data Type : Set where
    el : (s : Sort) (t : Term) → Type

  data Sort : Set where
    -- A Set of a given (possibly neutral) level.
    set  : (t : Term) → Sort
    -- A Set of a given concrete level.
    lit  : (n : ℕ) → Sort
    -- Anything else.
    unknown : Sort
```

Fig. 1. The data types for reflecting terms

piece of concrete program syntax into a syntax tree object which can be manipulated in the same system. Here we will present two case studies illustrating proof by reflection and how Agda's reflection mechanism can make the technique more usable and accessible.

3.1 Simple Example: Evenness

As a first example, we will cover an example taken from Chlipala [8], where we develop a procedure to prove that a number is even automatically. We start by defining the property Even below. There are two constructors: the first constructor says that zero is even; the second constructor states that if n is even, then so is $2 + n$.

```
data Even : ℕ → Set where
  isEvenZ  :                        Even 0
  isEvenSS : {n : ℕ} → Even n → Even (2 + n)
```

Using these rules to produce the proof that some large number n is even can be very tedious: the proof that $2 \times n$ is even requires n applications of the isEvenSS constructor. For example, here is the proof that 6 is even:

```
isEven6 : Even 6
isEven6 = isEvenSS (isEvenSS (isEvenSS isEvenZ))
```

To automate this, we will show how to *compute* the proof required. We start by defining a predicate even? that returns the unit type when its input is even and bottom otherwise:

```
even? : ℕ → Set
even? zero          = ⊤
even? (suc zero)    = ⊥
even? (suc (suc n)) = even? n
```

Next we need to show that the even? function is *sound*. To do so, we prove that when even? n returns ⊤, the type Even n is inhabited. This is done in the function soundnessEven. What is actually happening here is that we are giving a recipe for constructing proof trees, such as the one we manually defined for isEven6.

```
soundnessEven : {n : ℕ} → even? n → Even n
soundnessEven {0}           tt = isEvenZ
soundnessEven {1}           ()
soundnessEven {suc (suc n)} s  = isEvenSS (soundnessEven s)
```

Note that in the case branch for 1, we do not need to provide a right-hand side of the function definition. The assumption, even? 1, is uninhabited, and we discharge this branch using Agda's absurd pattern ().

Now that this has been done, if we need a proof that some arbitrary n is even, we only need to instantiate soundnessEven. Note that the value of n is an implicit argument to soundnessEven. The only argument we need to provide to our soundnessEven lemma is a proof that even? n is inhabited. For any closed term, such as the numbers 28 or 8772, this proof obligation can be reduced to proving $\top$, which is proven by the single constructor it has, tt.

```
isEven28   : Even 28
isEven28   = soundnessEven tt
isEven8772 : Even 8772
isEven8772 = soundnessEven tt
```

Now we can easily get a proof that arbitrarily large numbers are even, without having to explicitly write down a large proof tree. Note that it's not possible to write something with type Even 27, or any other uneven number, since the parameter even? n cannot be instantiated, thus tt would not be accepted where it is in the Even 28 example. This will produce a $\top$!=< $\bot$ type error at compile-time.

Since the type $\top$ is a simple record type, Agda can infer the tt argument, which means we can turn the assumption even? n into an implicit argument, meaning a user could get away with writing just soundnessEven as the proof, letting the inferrer do the rest. For clarity this is not done here, but the complete implementation available on github does use this trick.

3.2 Second Example: Boolean Tautologies

Another application of the proof by reflection technique is boolean expressions which are a tautology. We will follow the same recipe as for even naturals, with one further addition. In the previous example, the input of our decision procedure even? and the problem domain were both natural numbers. As we shall see, this need not always be the case.

Take as an example the boolean formula in equation 1.

$$(p_1 \vee q_1) \wedge (p_2 \vee q_2) \Rightarrow (q_1 \vee p_1) \wedge (q_2 \vee p_2) \tag{1}$$

It is trivial to see that this is a tautology, but proving this using deduction rules for booleans would be rather tedious. It is even worse if we want to check if the formula always holds by trying all possible variable assignments, since this will give 2^n cases, where n is the number of variables.

To automate this process, we will follow a similar approach to the one given in the previous section. We start by defining an inductive data type to represent boolean expressions with n free variables.

```
data BoolExpr (n : ℕ) : Set where
  Truth     :                                  BoolExpr n
  Falsehood :                                  BoolExpr n
  And       : BoolExpr n → BoolExpr n → BoolExpr n
```

```
  Or     : BoolExpr n → BoolExpr n → BoolExpr n
  Not    : BoolExpr n              → BoolExpr n
  Imp    : BoolExpr n → BoolExpr n → BoolExpr n
  Atomic : Fin n                   → BoolExpr n
```

There is nothing surprising about this definition; we use the type Fin n to ensure that variables (represented by Atomic) are always in scope. If we want to evaluate the expression, however, we will need some way to map variables to values. Enter Env n, it has fixed size n since a BoolExpr n has n free variables.

```
Env : ℕ → Set
Env = Vec Bool
```

Now we can define our decision function, which decides if a given boolean expression is true or not, under some assignment of variables. It does this by evaluating the formula's AST. For example, And is converted to the boolean function _∧_, and its two arguments in turn are recursively interpreted.

```
⟦_⊢_⟧ : ∀ {n : ℕ} (e : Env n) → BoolExpr n → Bool
⟦ env ⊢ Truth      ⟧ = true
⟦ env ⊢ Falsehood  ⟧ = false
⟦ env ⊢ And be be₁ ⟧ =   ⟦ env ⊢ be ⟧ ∧ ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Or be be₁  ⟧ =   ⟦ env ⊢ be ⟧ ∨ ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Not be     ⟧ = ¬ ⟦ env ⊢ be ⟧
⟦ env ⊢ Imp be be₁ ⟧ =   ⟦ env ⊢ be ⟧ ⇒ ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Atomic n   ⟧ = lookup n env
```

Recall our decision function even? in the previous section. It returned ⊤ if the proposition was valid, ⊥ otherwise. Looking at ⟦_⊢_⟧, we see that we should just translate true to the unit type and false to the empty type, to get the analogue of the even? function.

We call this function P, the string parameter serving to give a clearer type error to the user, if possible.

```
data Error (e : String) : Set where

So : String → Bool → Set
So _   true  = ⊤
So err false = Error err

P : Bool → Set
P = So "Argument expression does not evaluate to true."
```

Now that we have these helper functions, it is easy to define what it means to be a tautology. We quantify over a few boolean variables, and wrap the formula in our P decision function. If the resulting type is inhabited, the argument to P is a tautology, i.e., for each assignment of the free variables the entire equation still evaluates to true. An example encoding of such a theorem is Figure 3.2.

```
exampletheorem : Set
exampletheorem = (p1 q1 p2 q2 : Bool) → P ((p1 ∨ q1) ∧ (p2 ∨ q2)
                                            ⇒ (q1 ∨ p1) ∧ (q2 ∨ p2)
                                            )
```

Fig. 2. Example encoding of a tautology.

Here a complication arises, though. We are quantifying over a list of boolean values *outside* of the decision function P, so proving P to be sound will not suffice. We just defined a decision function ($[\![\_ \vdash \_]\!]$) to take an environment, an expression, and return a boolean. In Figure 3.2, though, we effectively quantified over all possible environments. We are going to need a way to lift our decision function to arbitrary environments.

The way we do this is the function foralls. This function represents the real analogue of even? in this situation: it returns a type which is only inhabited if the argument boolean expression is true under all variable assignments. This is done by generating a full binary tree of unit values $\top$, the single possible value which only exists if the interpretation function $[\![\_ \vdash \_]\!]$ evaluates to true in every leaf. This corresponds precisely to b being a tautology.

The Diff argument is unfortunately needed to prove that forallsAcc will eventually produce a tree with depth equal to the number of free variables in an expression.

```
forallsAcc : {n m : ℕ} → BoolExpr m → Env n → Diff n m → Set
forallsAcc b acc (Base  ) = P ⟦ acc ⊢ b ⟧
forallsAcc b acc (Step y) =
  forallsAcc b (true :: acc) y × forallsAcc b (false :: acc) y
foralls : {n : ℕ} → BoolExpr n → Set
foralls {n} b = forallsAcc b [] (zeroleast 0 n)
```

Now we finally know our real decision function, we can set about proving its soundness. Following the evens example, we want a function something like this.

```
sound : {n : ℕ} → (b : BoolExpr n) → foralls b → ...
```

What should the return type of the sound lemma be? We would like to prove that the argument b is a tautology, and hence, the sound function should return something of the form (b1 ... bn : Bool) → P B, where B is an expression in the image of the interpretation $[\![\_ \vdash \_]\!]$. For instance, the statement exampletheorem is a statement of this form.

The function proofObligation, given a BoolExpr n, generates the corresponding proof obligation. That is, it gives back the type which should be equal to the theorem one wants to prove. It does this by first introducing m universally quantified boolean variables. These variables are accumulated in an environment. Finally, when m binders have been introduced, the BoolExpr is evaluated under this environment.

```
proofObligation : (n m : ℕ) → Diff n m → BoolExpr m → Env n → Set
proofObligation .m m (Base  ) b acc = P ⟦ acc ⊢ b ⟧
proofObligation n   m (Step y) b acc =
  (a : Bool) →
    proofObligation (suc n) m y b (a :: acc)
```

Now that we can interpret a BoolExpr n as a theorem using proofObligation, and we have a way to decide if something is true for a given environment, we still need to show the soundness of our decision function foralls. That is, we need to be able to show that a formula is true if it holds for every possible assignment of its variables to true or false.

```
soundnessAcc : {m : ℕ} →       (b : BoolExpr m) →
               {n : ℕ} →       (env : Env n) →
               (d : Diff n m) → forallsAcc b env d →
               proofObligation n m d b env
soundnessAcc   bexp env Base H with ⟦ env ⊢ bexp ⟧
soundnessAcc   bexp env Base H | true  = H
soundnessAcc   bexp env Base H | false = Error – elim H
soundnessAcc {m} bexp {n} env (Step y) H =
  λ a → if {λ b → proofObligation (suc n) m y bexp (b :: env)} a
    (soundnessAcc bexp (true :: env)  y (proj₁ H))
    (soundnessAcc bexp (false :: env) y (proj₂ H))


soundness : {n : ℕ} → (b : BoolExpr n) → foralls b
          → proofObligation 0 n (zeroleast 0 n) b []
soundness {n} b i = soundnessAcc b [] (zeroleast 0 n) i
```

If we look closely at the definition of soundnessAcc (which is where the work is done – soundness merely calls soundnessAcc with some initial input, namely the BoolExpr n, an empty environment, and the proof that the environment is the size of the number of free variables) – we see that we build up a function that, when called with the values assigned to the free variables, builds up the environment and eventually returns the leaf from foralls which is the proof that the formula is a tautology in that specific case.

Now, we can prove theorems by calling soundness b p, where b is the representation of the formula under consideration, and p is the evidence that all branches of the proof tree are true. Agda is convinced that the representation does in fact correspond to the concrete formula, and also that soundness gives a valid proof. In fact, we need not even give p explicitly; since the only valid values of p are pairs of tt, the argument can be inferred automatically, if it is inhabited.

If the module passes the type checker, we know our formula is both a tautology, and that we have the corresponding proof object at our disposal afterwards, as in the following example.

```
rep      : BoolExpr 2
rep      = Imp (And (Atomic (suc zero)) (Atomic zero)) (Atomic zero)
someTauto : (p q : Bool) → P (p ∧ q ⇒ q)
someTauto = soundness rep _
```

The only part we still have to do manually is to convert the concrete Agda representation (p ∧ q ⇒ q, in this case) into our abstract syntax (rep here). This is unfortunate, as we end up typing out the formula twice. We also have to count the number of variables ourselves and convert them the to De Bruijn indices. This is error-prone given how cluttered the abstract representation can get for formulae containing many variables. It would be desirable for this process to be automated. In Sec. 3.3 a solution is presented using Agda's recent reflection API.

3.3 Adding Reflection

We can get rid of the aforementioned duplication using Agda's reflection API. More specifically, we will use the quoteGoal keyword to inspect the current goal. Given the Term representation of the goal, we can convert it to its corresponding BoolExpr.

The conversion between a Term and BoolExpr is achieved using the concrete2abstract function:

```
concrete2abstract : (t   : Term) → (n : ℕ)
                  → {pf  : isSoExprQ (stripPi t)}
                  → {pf2 : isBoolExprQ n (stripPi t) pf}
                  → BoolExpr n
```

Note that not every Term can be converted to a BoolExpr. The concrete2abstract function requires additional assumptions about the Term: it should only contain functions such as _∧_ or _∨_, and boolean variables. This is ensured by the assumptions isBoolExprQ and friends.

The concrete2abstract function is rather verbose, and is mostly omitted. A representative snippet is given in Fig. 3.3. The functions isBoolExprQ and isSoExprQ simply traverse the Term to see if it fulfills the requirements of being a boolean expression preceded by a series of universally quantified boolean variables.

All these pieces are assembled in the proveTautology function.

```
proveTautology : (t   : Term) →
                 {pf : isSoExprQ (stripPi t)} →
                 let n = freeVars t in
                   {pf2 : isBoolExprQ n (stripPi t) pf} →
                   let b = concrete2abstract t n {pf} {pf2} in
                     foralls b →
                     proofObligation 0 n (zeroleast 0 n) b []
proveTautology t i =
  soundness (concrete2abstract t (freeVars t)) i
```

```
term2boolexpr n (con tf []) pf with tf ≟ -Name quote true
term2boolexpr n (con tf []) pf | yes p = Truth
...
term2boolexpr n (def f []) ()
term2boolexpr n (def f (arg v r x :: [])) pf with f ≟ -Name quote ¬_
term2boolexpr n (def f (arg v r x :: [])) pf | yes p = Not (term2boolexpr n x pf)
...
```

Fig. 3. An illustration of converting a Term into a BoolExpr.

The proveTautology function converts a raw Term to a BoolExpr n format and calls the soundness lemma. It uses a few auxiliary functions such as freeVars, which counts the number of variables (needed to be able to instantiate the n in BoolExpr n), and stripSo & stripPi, which peel off the universal quantifiers and the function So with which we wrap our tautologies. These helper functions have been ommitted for brevity, since they are rather cumbersome and add little to the understanding of the subject at hand.

These are all the ingredients required to automatically prove that formulae are tautologies. The following code illustrates the use of the proveTautology functions; we can omit the implicit arguments for the reasons outlined in the previous section.

```
exclMid : (b : Bool) → P (b ∨ ¬ b)
exclMid = quoteGoal e in proveTautology e _

peirce  : (p q : Bool) → P (((p ⇒ q) ⇒ p) ⇒ p)
peirce  = quoteGoal e in proveTautology e _

mft     : exampletheorem
mft     = quoteGoal e in proveTautology e _
```

This shows that the reflection capabilities recently added to Agda are quite useful for automating certain tedious tasks, since the programmer now need not encode the boolean expression twice in a slightly different format. The conversion now happens automatically, without loss of expressive power or general applicability of the proofs resulting from soundness. Furthermore, by using the proof by reflection technique, the proof is generated automatically.

4 Discussion

This paper has presented two simple applications of proof by reflection. In the final version, we will show how Agda's reflection API has several other applications.

References

1. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
2. Norell, U.: Dependently typed programming in agda. Advanced Functional Programming (2009) 230–266
3. Pitman, K.: Special forms in Lisp. In: ACM Symposium on Lisp and Functional Programming. (1980)
4. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. PEPM '97 (1997)
5. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. (2002) 1–16
6. Alexandrescu, A.: Modern C++design. Addison Wesley (2001)
7. Agda developers: Agda 2.2.8 release notes. The Agda Wiki: `http://wiki.portal.chalmers.se/agda/agda.php?n=Main.Version-2-2-8` (2012) [Online; accessed 6-April-2012].
8. Chlipala, A.: Certified programming with dependent types. MIT Press (2011)

Agda Meets Accelerate

Extended Abstract

Peter Thiemann[1] and Manuel M. T. Chakravarty[2]

[1] University of Freiburg, Germany,
thiemann@informatik.uni-freiburg.de
[2] University of New South Wales, Sydney, Australia,
chak@cse.unsw.edu.au

Abstract. Embedded languages in Haskell benefit from a range of type extensions, such as type families, that are subsumed by dependent types. However, even with those type extensions, embedded languages for data parallel programming lack desirable static guarantees, such as static bounds checks in indexing and collective permutation operations.
This raises the question whether embedded languages for data parallel programming would benefit from fully-fledged dependent types, such as those available in Agda. We explored that question by designing and implementing an Agda frontend to Accelerate, an embedded language for data parallel programming aimed at GPUs. We discuss the potential of dependent types in this domain, describe some of the limitations that we encountered, and share some insights from our preliminary implementation.

Keywords: programming with dependent types, data parallelism

1 Introduction

Generative approaches to programming parallel hardware promise to combine high-level programming models with high performance. They are particularly attractive for targeting restricted architectures, such as GPUs (graphics processor units), that cannot efficiently execute code aimed at conventional multicore CPUs. Instead, GPUs require a high degree of data parallelism, restricted control flow, and carefully tailored data access patterns to be efficient. Previous work —for example, Accelerator [14], Copperhead [2], and Accelerate [3]— demonstrates that embedded array languages with a custom code generator can meet those GPU constraints with carefully designed language constructs.

Given a host languages with an expressive type system, it is attractive to leverage that type system to express static properties of the embedded language. For example, Accelerate, an embedded array language for Haskell, uses Haskell's recent support for type-level programming like GADTs and type families in that manner [3]. This design choice is important for approaches relying on runtime code generation: compile-time faults in the embedded language should be avoided

because it corresponds to at application runtime. Moreover, static guaranties hold the potential to improve the predictability of parallel performance.

Dependent types are an emerging approach to certified programming, where invariants are established in the form of types and proven at compile time. Many of Haskell's type-level extensions used in Accelerate approximate various aspects of dependently-typed programming. Hence, it it is natural to ask whether fully-fledged dependent types, such as those provided by Agda, improve the specification of an embedded language like Accelerate, whether they increase the scope of static guarantees, and whether they may be leveraged to predict performance more accurately.

This paper is a first investigation into this topic. It reports on a partial port of Accelerate to a new, dependently-typed host language, Agda [1, 9]. Agda is particularly suited to this port because of its foreign function interface to Haskell, which enables it to directly invoke the functionality of Accelerate.

Our investigation has the following structure. After recalling some background on Agda and Accelerate in Section 2 and describing related work in Section 3, Section 4 discusses potential uses of dependent types in an array-oriented data parallel language like Accelerate and how they were realized in our implementation. Section 5 considers conceptual problems and limitations that we ran into when constructing the Agda frontend for Accelerate. Section 6 explains some technical details of the implementation and discusses some example code.

2 Background

2.1 Agda

Agda [1, 9] is a dependently-typed functional programming language. Its basis is a dependently-typed lambda calculus extended with inductive data type families, dependent records, and parameterized modules. At the same time, Agda is also a proof assistant for interactively constructing proofs in an intuitionistic type theory based on the work of Per Martin-Löf [8].

One attractive feature of Agda's inductive data type families is the ability to construct indexed data types. A familiar example for such an indexed data type is the type `Vec A n` of vectors of fixed length `n` and elements of type `A`. This vector data types can be equipped with an access operation that restricts the index to the actual length of the vector at compile time.[3]

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

[3] An identifier can be an almost arbitrary string of Unicode characters except spaces, parentheses, and curly braces. Agda also supports mixfix syntax with the position of arguments indicated by underscores in the defining occurrence of an identifier.

```
data Vec (A : Set) : Nat -> Set where
  []   : Vec A zero
  _::_ : {n : Nat} -> A -> Vec A n -> Vec A (suc n)
```

The above defines the type `Nat` of natural numbers and an indexed data type `Vec A n` where `A` is a type and `n` is a natural number. The latter type comes with two constructors, `[]` for the vector of length zero and `_::_` for the infix cons operator that increases the length by one.

One way of writing a safe access operation first defines an indexed type that encodes the required less-than relation on natural numbers.

```
data _<_ : Nat -> Nat -> Set where
  z<s : {n : Nat} -> zero < suc n
  s<s : {m n : Nat} -> m < n -> suc m < suc n
```

Lines two and three of the definition encode named inference rules for the cases that $0 < n + 1$ (for all n) and that $m + 1 < n + 1$ if $m < n$ (for all m, n).

The access operation takes a vector of length `n`, an index `m`, and a proof of `m < n` (a derivation tree) to produce an element of the vector.

```
get : {A : Set} {n : Nat} -> Vec A n -> (m : Nat) -> m < n -> A
get []       _       ()      -- impossible case
get (x :: xs) zero    p       = x
get (x :: xs) (suc m) (s<s p) = get xs m p
```

This code cannot fail at run time because a caller has to construct the proof tree for `m < n` before invoking `get`. (In Agda, arguments in curly braces are *implicit arguments* that will be inferred if omitted in an application.) Thus, an "index out of bounds" error cannot happen.

2.2 Accelerate

Accelerate [3] is a data-parallel array language embedded into Haskell, which targets GPUs. It is a *generative library*, as its data-parallel array operations are not executed directly, but instead construct abstract syntax trees (AST) representing an entire data-parallel subcomputation. These *computation representations* are executed using a `run` operation that accepts such a representation (of type `Acc a`), compiles it to GPU kernels, uploads it to the device, executes it, and retrieves the results.[4]

```
CUDA.run :: Arrays a => Acc a -> a
```

The type class constraint `Arrays a` restricts the result type to a single array or a tuple of arrays.

[4] To distinguish Haskell code from Agda code, we display Haskell code in blue and with a vertical bar on the left side.

As computation representations of type `Acc a` are compiled at application runtime, all `Acc` compilation errors are effectivly *runtime errors* of the application. Hence, Accelerate uses a range of Haskell type system extensions to statically type Accelerate expressions, such that these runtime errors are avoided where possible. In particular, Accelerate uses GADTs [6], associated types [4], and type families [11].

As a simple example of an Accelerate program, consider a function implementing a dot product:

```
dotp :: Vector Float -> Vector Float -> Acc (Scalar Float)
dotp xs ys = let xs' = use xs
                 ys' = use ys
             in  fold (+) 0 (zipWith (*) xs' ys')
```

The types `Vector` and `Scalar` represent one- and zero-dimensional arrays. Plain arrays, such as `Vector Float` are conventional Haskell arrays, using an unboxed representation to improve performance. However, when they are wrapped into the constructor `Acc`, such as in `Acc (Scalar Float)`, they represent arrays of the embedded language and are allocated in GPU memory, which in current high-performance GPUs is physically separate from CPU memory.

The `use` operation makes a Haskell array available in the embedded language by wrapping it into the `Acc` constructor and copying it to GPU memory.[5] The operations `fold` and `zipWith` represent collective operations on Accelerate arrays, effectively producing a representation of an array computation yielding a value of `Scalar Float`; i.e., a single float value. This code relies heavily on (type class) overloading: `0`, `(+)`, and `(*)` are overloaded to just construct abstract syntax.

The types `Scalar` and `Vector` are type synonyms instantiating a shape-parameterised array type to the special case of zero and one dimensional arrays:

```
type Scalar e = Array DIM0 e
type Vector e = Array DIM1 e
```

The general type for `use` is

```
use :: Elt e => Array sh e -> Acc (Array sh e)
```

where the class `Elt` characterises all types that may be held in Accelerate arrays. These are currently primitive types and tuples.

Common dimensions, such as `DIM0`, `DIM1`, and so on, are predefined, but to enable shape polymorphic computations, along the lines pioneered in the Haskell array library Repa [7], shapes are inductively defined using type-level snoc lists built from the data types `Z` and `:.`:

```
data Z       = Z
data sh :. i = sh :. i
```

[5] Accelerate employs a caching strategy to avoid the transfer of arrays, which are already available in GPU memory.

Hence, the definitions of the dimensions:

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
-- <and so on>
```

3 Related Work

Peebles formalised parts of the API of the Haskell array library Repa using Agda [10]. The formalisation uses the same shape structure as we are using in Accelerate, but array computations are neither embedded nor can parallel high-performance code be generated.

Swierstra and Altenkirch investigated the use of dependent types for *distributed* array programming [12, 13]. Their notation for distributed arrays was inspired by X10 and the main focus is on expressing locality aware algorithms.

Dependent ML is an ML dialect with a restricted form of dependent types, which, among other applications, may be used to statically check array bounds in array operations [15]. However, only simple indexing and array updating are considered and not aggregate array operations, such as those provided by Accelerate.

Accelerator [14] enables embedded GPU computations in C# programs; it subsequently also added F# support. However, no attempt is made to track properties of array programs statically. Similarly, Copperhead [2] embedded an array language into Python, but does not attempt to track information statically.

4 Dependent Types for Accelerate

In this section, we investigate the potential uses of dependent typing in a language like Accelerate and point out how they may be implemented in Agda. First, we review some basics of the embedding.

4.1 Embedding of Haskell Types

Accelerate supports a wide range of numeric types, characterized by type class `Elt`, as base types for array computations. Almost all of these types have no suitable counterpart in Agda, which only supports natural numbers in unary encoding. For that reason, our embedding keeps the Haskell types abstract in Agda. To specify type signatures and in particular functions that are polymorphic in such a Haskell type or depend on it in some way, we have reified these types in an Agda type `Element`.

```
data Element : Set where
  Bool   : Element
  Int    : Element
```

```
  Float  : Element
  Double : Element
  Pair   : Element -> Element -> Element
  -- and so on
```

Corresponding to Haskell type classes that are used in Accelerate, our embedding supplies predicates that characterize subsets. For example, the set of numeric types is defined by a predicate `IsNumeric`.[6]

```
IsNumeric : Element -> Set
IsNumeric Int = ⊤
IsNumeric Float = ⊤
IsNumeric Double = ⊤
IsNumeric _ = ⊥
```

The embedding declares further subsets all in the same style.

4.2 Array Types

To see the Agda embedding in action, we translate the dot product example from Section 2.2 to Agda.

```
dotp : {E : Element} {{p : IsNumeric E}} {n : Nat}
     -> PreVector n E -> PreVector n E -> Scalar E
dotp{E} xs ys =
  let xs' = use xs
      ys' = use ys
  in
  fold _+_ ("0" ::: E) (zipWith _*_ xs' ys')
```

Unlike the Accelerate code, this function is polymorphic with respect to the array element type, provided it is numeric.[7] It furthermore takes the length n as a parameter thus ensuring that the two input vectors have the same size. The `PreVector` type of the arguments corresponds to the plain `Vector` type in Accelerate, whereas the result type `Scalar E` corresponds to `Acc (Scalar E)`—a piece of abstract syntax.

The `use` function works as before, but its type includes more information:

```
use : {sh : Shape}{E : Element} -> PreArray sh E -> Array sh E
```

Like `E`, the index `sh` is now an element of an ordinary type instead of having to rely on type-level snoc lists:[8]

[6] $\top$ is a one-element type, whereas $\bot$ is a type without elements. These types customarily represent truth and falsity.

[7] In Agda, arguments in double curly braces are *instance arguments* [5]. We use them much like type class constraints are used in Haskell.

[8] Recent work on Haskell's type system manages to avoid this issue [16].

```
data Shape : Set where
  Z     : Shape
  _:<_> : Shape -> Nat -> Shape
```

Asking for arrays of equal shape, as in the signature of `use`, means that the arrays have to have the exact same layout. The `PreVector` and `Vector` types are just synonyms as in Haskell:

```
PreVector n E = PreArray (Z :< n >) E
Vector n E    = Array    (Z :< n >) E
```

The functions `fold`, `zipWith`, and `:::` are discussed in the subsequent subsections. The functions `_+_` and `_*_` both have the same type:

```
_+_ : {E : Element} {{p : IsNumeric E}} -> Exp E -> Exp E -> Exp E
```

They are restricted to arguments of numeric type and construct abstract syntax for an addition or a multiplication by delegating to the corresponding Accelerate functions. The type `Exp E`, which denotes an AST of an expression of type `E`.

4.3 Exact Checking of Array Bounds

Accelerate's API features expressive type constraints that describe the shape of the array arguments and results. These constraints ensure that no shape mismatches occur (e.g., a one-dimensional array cannot be considered two-dimensional). However, they do not ensure at compile time that the sizes of the dimensions match up.

As an example, consider the function `reshape`. It takes a target shape `sh` and an array of source shape `sh'` and changes the layout of that array to `sh`.

```
reshape :: Exp sh -> Acc (Array sh' e) -> Acc (Array sh e)
```

For this reshaping to work correctly, the underlying number of elements must remain the same. For example, while it makes sense to reshape a two-dimensional 3×4-array to a vector of size 12 or to a three-dimensional $3 \times 2 \times 2$-array, an attempt to reshape to a 2×5-array should be rejected at compile time.

As `Shape` is an ordinary data type in Agda, we can define a `size` function that computes the number of elements stored in an array of a certain shape.

```
size : Shape -> Nat
size Z = 1
size (sh :< n >) = size sh * n
```

Now we can state an accurate type for `reshape` in Agda, which involves an extra argument with a proof that the source and target shapes have the same size.

```
reshape : {sh : Shape} {E : Element}
       -> (sh' : Shape) -> Array sh E -> (size sh ≡ size sh')
       -> Array sh' E
```

There is a subtle difference to the original signature. In Accelerate, the first argument is an *expression* that produces a value of type `sh` at run time, whereas the Agda `reshape` requires a `Shape` as its first argument.

Furthermore, functions like `map` and `zipWith` obtain more precise types. The type of `map` tells us that the input shape is identical to the output shape:

```
map : {A B} {sh} -> (Exp A -> Exp B) -> Array sh A -> Array sh B
```

Similarly, the type of `zipWith` restricts its input arrays to identical shapes:

```
zipWith : {A B C} {sh} -> (Exp A -> Exp B -> Exp C)
        -> Array sh A -> Array sh B -> Array sh C
```

The latter type is more restrictive than the Accelerate implementation of `zipWith`. Instead of checking the sizes of the input arrays, it truncates them to the respective minima. A corresponding Agda type could be developed easily. It just requires a binary function that computes the minimum of two shapes, which is a simple exercise.

4.4 Associativity of Operations

Some parallel reduction operations require their base operation to be associative to return a predictable result. Here are two examples from Accelerate.

```
fold  :: (Shape ix, Elt a) =>
         (Exp a -> Exp a -> Exp a) -> Exp a ->
         Acc (Array (ix :. Int) a) -> Acc (Array ix a)
fold1 :: (Shape ix, Elt a) =>
         (Exp a -> Exp a -> Exp a) ->
         Acc (Array (ix :. Int) a) -> Acc (Array ix a)
```

In both cases, the text of the documentation says that "the first argument needs to be associative" and the `fold1` documentation "requires the reduced array to be non-empty". The second requirement can be enforced by asking for a suitable proof object on each call of `fold1`:

```
fold1 : ... -> Array (sh :< n >) E -> (size sh * n > 0)
            -> Array sh E
```

The first requirement can be rephrased to saying that the first two parameters of `fold` together form a monoid, which requires an associative operation with a unit element. The concept of a monoid can be formalized in Agda, which has indeed been done in the standard library. Unfortunately, the formalization from the library cannot be used because Accelerate deals with ASTs, not with values. So, a formalization is required that states that the meaning of an AST-encoded function is associative and the meaning of another AST-encoded constant is its unit element. Given that Accelerate encodes AST construction using higher-order abstract syntax, such a formalization is not straightforward. Moreover,

even given expressions with a fixed meaning, there is no general shape for associative functions, so that proofs can only be done for special cases.

In any case, providing such information would be done by including an additional argument that holds a suitable proof object, as in

```
fold : {E}{sh}{n} -> (f : Exp E -> Exp E -> Exp E) -> (e : Exp E)
     -> Array (sh :< n >) E -> IsMonoid f e -> Array sh E
```

where

```
IsMonoid : {E} -> (f : Exp E -> Exp E -> Exp E) -> (e : Exp E) -> Set
IsMonoid f e = ( IsAssociative f , IsUnit f e)
```

4.5 Embedding of Constants

Accelerate relies on Haskell's built-in support for the type classes `Num` and `Fractional` to embed constants. The Haskell compiler reads each integer literal as a value of type `Integer`, which is a built-in type of arbitrary precision integers. To this value, Haskell applies the function `fromInteger` that converts to the type expected by the context. Similarly, floating point constants are read as values of type `Rational` (`Integer` fractions) and then converted using `fromRational`. Accelerate provides instances of these type classes that define `fromInteger` and `fromRational` to produce suitable AST fragments.

Because of Agda's limited support for numeric data types, we embed more ambitious numeric literals for floating point numbers using a string with an explicit type annotation that determines the parsing of the string. Here are some example embeddings:

```
"3.1415926" ::: Float
"6.0221415E23" ::: Double
```

Recall that `Float` and `Double` are not types, but rather values of type `Element`. All magic of the embedding is hidden in the `:::` operation:

```
_:::_ : (s : String) -> (E : Element)
     -> {{nu : IsNumeric E}} -> {p : T (s parsesAs E)} -> Exp E
s ::: E = Ex (constantFromString (EltDict E) (ReadDict E) s)
```

The arguments `s` and `E` are explicit, but the remaining ones are to be inferred by Agda. As mentioned before, the argument `nu` is an instance argument; it is automatically filled-in with a suitably typed value that is currently in scope [5]. Here, the predicate `IsNumeric` plays the role of a type class that characterizes the numeric types.

The function `parsesAs` dispatches on its "type" argument and checks whether the string is a constant of the expected type. The function `constantFromString` is imported from Accelerate. It is an overloaded function that requires two type dictionaries, which are computed from `E` using the functions `EltDict` and `ReadDict`.

5 Limitations

In a number of places, Accelerate's generativity limits the applicability of dependent typing. We already mentioned that the formalization of associativity or of the concept of a monoid gets unmanageable because such properties have to be asserted for abstract syntax.

For a related problem, consider an implementation of the `filter` operation that takes a predicate and a source array and returns an array that only contains the elements of the source array fulfilling the predicate. First of all, filtering only makes sense for one-dimensional arrays, that is, for vectors. To see the second catch, let's try to write down a dependent type signature for `filter`.

```
filter : {n m : Nat}{E : Element}
       -> Vector n E -> (Exp E -> Exp Bool) -> Vector m E
```

The problem is that the size of the result cannot be determined statically. In fact, the only thing we know about `m` is that it must be less than or equal to `n`. However, we cannot prove this from the code because of a staging restriction. The Accelerate implementation computes the length of the result only when the generated GPU code is executed. The function `Exp E -> Exp Bool` maps abstract syntax to abstract syntax; it does not directly implement a Boolean predicate. Hence, we cannot use the predicate in the result type of `filter` to more accurately constrain the size of the resulting vector — unless we include an evaluator for Accelerate expressions.

We might contemplate emplying an existential type like

```
exists Nat (\ m -> m <= n -> Vector m E)
```

However, it is not possible to build such an existential package because the evidence `m` is not available when the package would have to be constructed.

However, an alternative encoding of arrays can be used which is compatible with filtering of elements. The idea of this encoding is to keep all elements but mark those which are no longer present because they have been filtered out. There are several ways of implementing this idea. The simplest approach is to wrap each element in a maybe type or pair up each element with a boolean flag that indicates its presence.[9]

```
FVector : Nat -> Element -> Set
FVector n E = Vector n (Pair Bool E)
```

Now filtering becomes quite simple because the length of the`FVector` does not change. Furthermore, filtering could be extended to multi-dimensional arrays, although it is not clear if an array that includes non-existing elements is a sensible notion.

[9] A `Maybe` type is on one hand the better option, but it has to be coded without using pattern matching.

```
filterF : {n : Nat}{E : Element}
        -> (Exp E -> Exp Bool) -> FVector n E -> FVector n E
filterF {n}{E} pred vec =
  map g vec
  where
  g : Exp (Pair Bool E) -> Exp (Pair Bool E)
  g bx = pair ((fst bx) && p (snd bx)) x
```

However, mapping becomes more complicated because it either has to materialize a dummy result for each absent element in the argument vector or apply the function to absent elements, too.[10]

```
mapF : {n : Nat}{E F : Element}
     -> Exp F -> (Exp E -> Exp F) -> FVector n E -> FVector n F
mapF {n}{E}{F} defaultF f vec =
  map g vec
  where
  g : Exp (Pair Bool E) -> Exp (Pair Bool F)
  g bx = if (fst bx) then (pair (fst bx) (f (snd bx)))
                     else (pair (fst bx) defaultF)
```

On the positive side, some operations can get rid of the absent elements. In particular, a fold operation which reduces a filtered vector with a monoid returns a single value. In Accelerate, such a value has type `Scalar`, which is a synonym for an array of dimension 0.

```
foldF : {n : Nat}{E : Element}
      -> (Exp E -> Exp E -> Exp E) -> Exp E
      -> FVector n E -> Scalar E
foldF f e vec =
  fold f e (map (\ bx -> if (fst bx) then (snd bx) else e) vec)
```

Other operations like `fold1` and the scan operations present in Accelerate can also be lifted to this representation, but they retain a notion of absent elements and do not allow to revert to a non-filtered representation.

In the end, such a representation may not be a loss on a GPU. As long as all computations take the same path, all processing elements work in unison. As soon as there are different paths in the same computation step, then some elements will be idle for part of the computation step. So it would be most advantageous to organize work as uniformly as possible.

6 Implementation

Ordinarily, Agda is an interactive tool for constructing proofs and verified programs. Programs may be run, which amounts to normalizing Agda expressions, but this process is not very efficient.

[10] This complication could be avoided with the `Maybe` type.

Alternatively, an interactively developed program may be compiled to Haskell using the Alonzo compiler. This compiler supports a Haskell foreign function interface (FFI), which enables Agda programs to invoke Haskell functions.

Using this interface amounts to declaring a typed identifier in Agda and then binding the identifier to a suitably typed Haskell function. As an example, consider the import of the `use` function.

```
postulate
  useHs : {E : Set}
      -> HsEltDict E -> HsArray HsDIM1 E -> Acc (AccArray HsDIM1 E)
  {-# COMPILED useHs       (\ _ -> Accel.use) #-}
```

The first three lines introduce the typed identifier `useHs` and the last line is a pragma for the Alonzo compiler that binds the Agda identifier `useHs` to the Haskell expression on the right. But wait, this type looks very unpleasant and quite different to the one mentioned in Section 4.2. This difference arises because the type translation of Alonzo is unable to cope with the index type `Shape`. For that reason, the interface uses a simplified array type and adapter functions are required, in the worst case, both on the Agda side and on the Haskell side of the interface.

At the foreign function interface level, all arrays are considered as one-dimensional arrays. Additional arguments are passed to encode the shape information as far as it is needed. The Agda adapter provides the encoding of this structure and the Haskell adapter decodes it again.

We believe that these adapations only have a minor performance impact because (1) most functions just manipulate abstract syntax, so that only AST construction is affected, and (2) internally, Accelerate considers all arrays as one-dimensional so that operations like `reshape` are no-ops at run time.

Here is the Agda adapter for `use`:

```
use : {sh : Shape}{E : Element} -> PreArray sh E -> Array sh E
use {sh}{E} (PA y) = Ar (useHs (EltDict E) y)
```

It makes use of two wrapper types. `PreArray` wraps a one-dimensional Haskell array using the constant `HsDIM1` (the `DIM1` type shown in Section 2.2 imported from Haskell via FFI) and the function `EltType` (not shown), which interprets a value of type `Element` as a Haskell type. The latter types are also imported via FFI.

```
data PreArray (E : Element) : Shape -> Set where
  PA : {sh : Shape} -> HsArray HsDIM1 (EltType E) -> PreArray sh E
```

The `Array` type wraps an AST reference for an Accelerate array, where `Acc` and `AccArray` are types imported from Haskell.

```
data Array (E : Element) : Shape -> Set where
  Ar : {sh : Shape} -> Acc (AccArray HsDIM1 (EltType E)) -> Array sh E
```

The `EltDict` function translates a value `(E : Element)` into a Haskell expression that evaluates to a dictionary for the Haskell type of `E` for the Haskell type class `Elt`. Such a dictionary is passed, whenever that corresponding Haskell function has type class constraints.

```
EltDict : (E : Element) -> HsEltDict (EltType E)
```

The Haskell side of the adapter has several purposes. First, it materializes the type class dictionaries from the encoding that we just discussed. Second, it reconstructs sufficient information about the array shape so that the intended operation can execute. Here is the code for `Accel.use`, where the module name `A` is a shorthand for `Data.Array.Accelerate`.

```
use :: EltDict e -> Array A.DIM1 e -> A.Acc (A.Array A.DIM1 e)
use EltDict (ARRAY ar) = (A.use ar)
```

It does not have to reconstruct any information except the type class constraint. This constraint is materialized using the type `EltDict` below.

```
data EltDict e where
  EltDict :: (A.Elt e) => EltDict e
```

This datatype is built such that each value captures the `Elt` dictionary of type `e`. It remains to build such values for all types that we want to transport across the FFI. These are the values used by the (Agda) `EltDict` function. Here are two examples.

```
eltDictBool :: EltDict Bool
eltDictBool = EltDict

eltDictInt :: EltDict Int
eltDictInt = EltDict
```

As an example for a function that requires more work on either side, consider the `fold` operation.

```
fold : {E}{sh}{n}
     -> (Exp E -> Exp E -> Exp E)
     -> Exp E
     -> Array (sh :< n >) E
     -> Array sh E
fold {E}{sh}{n} f (Ex e) (Ar a) =
  Ar (foldHs (EltDict E) (toHsInt (size sh)) (toHsInt n)
             (unwrap2 f) e a)
```

As values of type `Exp` also need a wrapper type in Agda (it is not possible to import type constructors via the FFI), there is some unwrapping going on for the `e` and `f` arguments. The implementation of `fold` just calls the `foldHs` function and encodes the information about the shape in two integer arguments. Here,

`size sh` is the size of the result and `n` is the size of the dimension that is folded. As these values are initially available as Agda natural numbers, they need to be converted to Haskell numbers using the function `toHsInt`.

The `foldHs` function is defined via the FFI.

```
postulate
  foldHs : {A : Set}
           -> HsEltDict A
           -> HsInt
           -> HsInt
           -> (AccExp A -> AccExp A -> AccExp A)
           -> AccExp A
           -> Acc (AccArray HsDIM1 A)
           -> Acc (AccArray HsDIM1 A)
  {-# COMPILED foldHs      (\_ -> Accel.fold) #-}
```

The Haskell adapter reconstructs the `Elt` dictionary as before, but it also needs to reshape the one-dimensional array representation into a two-dimensional one for executing the fold operation. The two size arguments are required for exactly this reshape operation. With that insight, the code is straightforward.

```
fold :: EltDict a
     -> Int -> Int
     -> (A.Exp a -> A.Exp a -> A.Exp a)
     -> A.Exp a
     -> A.Acc (A.Array A.DIM1 a)
     -> A.Acc (A.Array A.DIM1 a)
fold EltDict size2 size1 f e a =
     (A.reshape (A.lift (A.Z A.:. size2))
      (A.fold f e
       (A.reshape (A.lift (A.Z A.:. size2 A.:. size1)) a)))
```

Fortunately, the `fold` example is about as complicated as the adapter code gets. There are also many cases where at least one side of the adapter code is trivial. However, each case must be considered separately.

7 Conclusion

We have build an experimental Agda frontend for the Accelerate language. The goal of this experiment was to explore potential uses of dependently-typed programming for data-parallel languages.

At the moment, the outcome of the experiment is mixed. It is successful, because we have been able to construct Agda functions for a representative sample of Accelerate's functionality. However, there was less scope for encoding extra information in the dependent types than we had hoped for. Exact matching of array bounds works, but results in restrictions (like the problems with `zipWith` and filtering) that were not anticipated.

Exploiting algebraic properties did not work out in the intended way, mainly because it boils down to asserting that some AST denotes an associative function. However, these assertions cannot be proven: the proof would have to apply the semantics to the AST, but the AST is an abstract type in our implementation. An AST representation in Agda might give us a better handle at this problem.

In some places, the Agda frontend is less dynamic than Accelerate. In a number of places, Accelerate accepts a run-time value of type `Exp sh` for a shape argument, where the Agda frontend requires a value of type `Shape`. To address this problem, we would have to include a `Shape`-indexed encoding of the `Shape` type in the `Element` type so that we can describe the type of an expression whose value has a certain shape.

Finally, the type translation of Agda's FFI has many shortcomings that caused a number of problems for transporting information between Agda and Haskell. One part of the problem is, unfortunately, the rich type structure of Accelerate which already encodes much useful information. An alternative, untyped (or less-typed) interface to Accelerate would make the adaptation to an Agda frontend much simpler.

References

1. A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda - a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78, Munich, Germany, 2009. Springer.
2. B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. Technical Report UCB/EECS-2010-124, University of California, Berkeley, 2010.
3. M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In M. Carro and J. H. Reppy, editors, *Workshop on Declarative Aspects of Multicore Programming, DAMP 2011*, pages 3–14, Austin, TX, USA, Jan. 2011. ACM.
4. M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In B. C. Pierce, editor, *Proceedings International Conference on Functional Programming 2005*, pages 241–253, Tallinn, Estonia, Sept. 2005. ACM Press, New York.
5. D. Devriese and F. Piessens. On the bright side of type classes: Instance arguments in Agda. In O. Danvy, editor, *Proceedings International Conference on Functional Programming 2011*, pages 143–155, Tokyo, Japan, Sept. 2011. ACM Press, New York.
6. S. L. P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In J. Lawall, editor, *ICFP*, pages 50–61, Portland, Oregon, USA, Sept. 2006. ACM Press, New York.
7. G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *ICFP '10: Proc. of the 15th ACM SIGPLAN Intl. Conf. on Functional Programming*. ACM, 2010.
8. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.
9. U. Norell. Dependently typed programming in Agda. In P. W. M. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *Advanced Functional Programming*,

volume 5832 of *Lecture Notes in Computer Science*, pages 230–266, Heijen, The Netherlands, 2008. Springer.
10. D. Peebles. A dependently typed model of the Repa library in Agda. https://github.com/copumpkin/derpa, 2011.
11. T. Schrijvers, S. L. Peyton Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In P. Thiemann, editor, *Proceedings International Conference on Functional Programming 2008*, pages 51–62, Victoria, BC, Canada, Oct. 2008. ACM Press, New York.
12. W. Swierstra. More dependent types for distributed arrays. *Higher-Order and Symbolic Computation*, pages 1–18, 2010.
13. W. Swierstra and T. Altenkirch. Dependent types for distributed arrays. In *Trends in Functional Programming*, volume 9, 2008.
14. D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proc. of the 12th Intl. Conf. on Architectural Support for Programming Lang. and Operating Systems*, pages 325–335. ACM, 2006.
15. H. Xi. Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming*, 12(2), Mar. 2007.
16. B. A. Yorgey, S. Weirich, J. Cretin, S. L. P. Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In B. C. Pierce, editor, *Proceedings of TLDI 2012*, pages 53–66, Philadelphia, PA, USA, Jan. 2012. ACM.

On Binomial Expansions in Moessner's Theorem

Olivier Danvy
Aarhus University

Moe Masuko
JSPS Research Fellow
Ochanomizu University

Extended abstract submitted to IFL 2012

Abstract

In our formalization in Coq of Moessner's theorem, we have observed that each instance of this theorem at index $\mathtt{i}$ gives rise to a master lemma that involves each of the monomials of the binomial expansion of $(\mathtt{a}+1)^{\mathtt{i}}$, where $\mathtt{a}$ is an accumulator. We have written a code generator, in ML, mapping an index to a Coq script that states and proves the instance of Moessner's theorem at that index. In this paper, we report this experiment and situate it in the landscape of studies of Moessner's theorem.

1 Overview

Following Hinze's study of Moessner's theorem [2] and Niqui and Rutten proof by co-induction [3], we are formalizing Moessner's theorem and its proof in Coq [1]. As a stepping stone, we have considered its instance at a given index. Given

- `stream_of_ones`, a constant stream of the natural number 1, i.e., $(1, 1, 1, \ldots)$,
- `stream_of_positive_powers_of_3`, the stream of the successive powers of 3, i.e., $(1^3, 2^3, 3^3, \ldots)$, `stream_of_positive_powers_of_4`, the stream of the successive powers of 4, i.e., $(1^4, 2^4, 3^4, \ldots)$, etc.,
- `skip_2`, `skip_3`, `skip_4`, the stream transducers that respectively skip every second, third, fourth, etc. element of a given stream,
- `sums`, the stream transducer that maps a stream to the stream of its partial sums, and `sums_aux`, a version of `sums` that uses an accumulator, and
- `stream_bisimilar`, a bisimilarity predicate,

we state and prove, e.g., Moessner's theorem at index 4, using the master lemma `Moessner_4_aux` specified below:

```
Theorem Moessner_4 :
  stream_bisimilar
    stream_of_positive_powers_of_4
    (sums
      (skip_2
        (sums
          (skip_3
            (sums
              (skip_4
                (sums
                  stream_of_ones)))))))).
```

```
Proof.
  unfold stream_of_positive_powers_of_4.
  unfold sums.
  apply (Moessner_4_aux 0).
Qed.
```

The skeleton of the master lemma at index 4 reads as follows, where we have left the starting accumulators blank:

```
Lemma Moessner_4_aux :
  forall (a : nat),
    stream_bisimilar
      (make_stream_of_nats (S a)
                           (fun i => i * i * i * i)
                           S)
      (sums_aux ???
         (skip_2
          (sums_aux ???
            (skip_3
              (sums_aux ???
                (skip_4
                  (sums_aux ???
                    stream_of_ones)))))))).
```

As it happens, these starting accumulators correspond to the succcessive monomials of Newton's binomial expansion of $a + 1$ at index 4:

$$(a+1)^4 = a^4 + 4 \cdot a^3 + 6 \cdot a^2 + 4 \cdot a + 1$$

```
Lemma Moessner_4_aux :
  forall (a : nat),
    stream_bisimilar
      (make_stream_of_nats (S a)
                           (fun i => i * i * i * i)
                           S)
      (sums_aux (a * a * a * a)
         (skip_2
          (sums_aux (4 * a * a * a)
            (skip_3
              (sums_aux (6 * a * a)
                (skip_4
                  (sums_aux (4 * a)
                    stream_of_ ones )))))))).
```

This property appears to hold for arbitrary indices. The corresponding number and structure of auxiliary lemmas is so regular that we have characterized them inductively and written, in ML, a code generator mapping an index to a Coq script that states and proves the instance of Moessner's theorem at that index. The full version of this extended abstract will detail this characterization and situate our experiment in the landscape of studies of Moessner's theorem.

Acknowledgments: This work was initiated during a visit of the second author to the first author, in the spring of 2012.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[2] Ralf Hinze. Scans and convolutions – a calculational proof of Moessner's theorem. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages, 20th International Workshop, IFL 2008*, number 5836 in Lecture Notes in Computer Science, pages 1–24, Hatfield, UK, September 2008. Springer.

[3] Milad Niqui and Jan Rutten. A proof of Moessner's theorem by coinduction. *Higher-Order and Symbolic Computation*, 2012. To appear.

Functional implementation of well-typings in Java$_\lambda$

Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart
Department of Computer Science
Florianstraße 15, D–72160 Horb
pl@dhbw.de

Abstract. In the last decade Java has been extended by some features, which are well-known from functional programming languages. In Java 8 the language will be expanded by λ–expressions. We have extended a subset of Java 7 by λ–expressions and function types. We call this language Java$_\lambda$. For Java$_\lambda$ we presented a type inference algorithm. In this contribution we present a prototypical implementation of the type inference algorithm implemented in Haskell.

1 Introduction

In the late eighties Fuh and Mishra have presented a type inference algorithm for a small function programming language with subtyping and without overloading [FM88].
In Java$_\lambda$ we have a similiar situation. Subtyping is allowed and functions, which are declared by λ–expressions, are not overloaded.
We have adapted the Fuh and Mishra algorithm to a type inference algorithm for Java$_\lambda$ [Plü11]. The main difference is the definition of the subtyping ordering. Therefore follows that the unification in [FM88] had to be substituted by our type unification [Plü09].
The type inference algorithm consists of three functions:

TYPE: The function **TYPE** types each sub-term of the λ–expressions by type variables and determines corecions (subtype pairs), which have to be solved.
MATCH: The function **MATCH** adapts the structure of the types of each subtype pair and reduces the coercions to atomic coercions. An atomic coercion is a subtype pair, where the types consists only of type variables and type constants.
CONSISTENT: The function **CONSISTENT** determines iteratively solutions for the atomic coercions. If there is at least one solution, the result is true. This means that there is a correct typing for the λ–expressions. Otherwise, the algorithm fails.

The algorithm itself is given as:

WTYPE: `TypeAssumptions` $\times$ `class` $\rightarrow \{\,$`WellTyping`$\,\} \cup \{\,fail\,\}$

$\mathbf{WTYPE}(\, Ass, \mathsf{Class}(\, cl, \mathsf{extends}(\, \tau' \,), fdecls, ivardecls \,) \,) =$
  let
    $(\{\, f_1 : a_1, \ldots, f_n : a_n \,\}, CoeS) =$
      $\mathbf{TYPE}(\, Ass, \mathsf{Class}(\, cl, \mathsf{extends}(\, \tau' \,), fdecls, ivardecls \,) \,)$
    $(\sigma, AC) = \mathbf{MATCH}(\, CoeS \,)$
  in
    if CONSISTENT$(\, AC \,)$ **then**
      $\{\, (AC, Ass \vdash f_i : \sigma(\, a_i \,)) \mid 1 \leqslant i \leqslant n \,\}$
    else $fail$

The result of the algorithm is the set of well-typings:

$$\{\, (AC, Ass \vdash f_i : \sigma(\, a_i \,)) \mid 1 \leqslant i \leqslant n \,\}$$

where

- AC is a set of coercions,
- Ass is a set of type assumptions,
- f_i are function names, and
- $\sigma(\, a_i \,)$ are types.

It is a problem that well-typings are not included in the Java type-system.
If we consider **CONSISTENT** more detailed, we will recognize, that for all types, which are in relation with a non-variable type, all possible instances are determined. We call a function, which gives these instances as result, **SOLUTIONS**. This means that the set of corecions could be reduced to a set AC' consisting only type variables. These pairs could be expressed by bounded type variables in Java. Here is a small extention necessary, e.g. parameters of a function could also be a bound of another parameter.
Hence the algorithm looks like this:

WTYPE: `TypeAssumptions` $\times$ `class` $\to \{\,$ `WellTyping` $\,\} \cup \{\, fail \,\}$

$\mathbf{WTYPE}(\, Ass, \mathsf{Class}(\, cl, \mathsf{extends}(\, \tau' \,), fdecls, ivardecls \,) \,) =$
  let
    $(\{\, f_1 : a_1, \ldots, f_n : a_n \,\}, CoeS) =$
      $\mathbf{TYPE}(\, Ass, \mathsf{Class}(\, cl, \mathsf{extends}(\, \tau' \,), fdecls, ivardecls \,) \,)$
    $(\sigma, AC) = \mathbf{match}(\, CoeS \,)$
    $((\tau_1, \ldots, \tau_m), AC') = \mathbf{SOLUTIONS}(\, AC \,)$
  in
    $\{\, (AC', Ass \vdash \{\, f_i : \tau_j \circ \sigma(\, a_i \,) \mid 1 \leqslant i \leqslant n \,\}) \mid 1 \leqslant j \leqslant m \,\}$

2 The language

The language Java_λ is an extension of our language in [Plü07] by λ–expressions and function types. Java_λ is the core of the language, which is described by

$$
\begin{array}{lll}
\mathit{Source} & := & \mathit{class}* \\
\mathit{class} & := & \mathsf{Class}(\mathit{simpletype}, [\ \mathsf{extends}(\ \mathit{simpletype}\ ),]\mathit{IVarDecl}*, \underline{\mathit{FunDecl}*}) \\
\mathit{IVarDecl} & := & \mathsf{InstVarDecl}(\ \mathit{simpletype}, \mathit{var}\ ) \\
\underline{\mathit{FunDecl}} & := & \underline{\mathsf{Fun}(\ \mathit{fname}, [\boldsymbol{\mathit{type}}], \mathit{lambdaexpr}\ )} \\
\mathit{block} & := & \mathsf{Block}(\ \mathit{stmt}*\ ) \\
\mathit{stmt} & := & \mathit{block} \mid \mathsf{Return}(\ \mathit{expr}\ ) \mid \mathsf{While}(\ \mathit{bexpr}, \mathit{block}\ ) \mid \\
 & & \mathsf{LocalVarDecl}(\ \mathit{var}[, \boldsymbol{\mathit{type}}]\ ) \mid \mathsf{If}(\ \mathit{bexpr}, \mathit{block}[, \mathit{block}]\ ) \mid \mathit{stmtexpr} \\
\underline{\mathit{lambdaexpr}} & := & \underline{\mathsf{Lambda}(\ ((\mathit{var}[, \boldsymbol{\mathit{type}}]))*, (\mathit{stmt} \mid \mathit{expr})\ )} \\
\underline{\mathit{stmtexpr}} & := & \underline{\mathsf{Assign}(\ \mathit{vexpr}, \mathit{expr}\ )} \mid \underline{\mathsf{New}(\ \mathit{simpletype}, \mathit{expr}*\ )} \mid \underline{\mathsf{Eval}(\ \mathit{expr}, \mathit{expr}*\ )} \\
\mathit{vexpr} & := & \mathsf{LocalOrFieldVar}(\ \mathit{var}\ ) \mid \mathsf{InstVar}(\ \mathit{expr}, \mathit{var}\ ) \\
\mathit{expr} & := & \underline{\mathit{lambdaexpr}} \mid \mathit{stmtexpr} \mid \mathit{vexp} \mid \mathsf{this} \mid \mathsf{This}(\ \mathit{simpletype}\ ) \mid \mathsf{super} \mid \\
 & & \underline{\mathsf{InstFun}(\ \mathit{expr}, \mathit{fname}\ )} \mid \mathit{bexp} \mid \mathit{sexp}
\end{array}
$$

Fig. 1. The abstract syntax of Java_λ

Reinhold's in [lam10]. In (Fig. 1) an abstract representation is given, where the additional features are underlined. Beside instance variables functions can be declared in classes. A function is declared by its name, optionally its type, and a λ–expression. Methods are not considered in this framework, as methods can be expressed by functions. A λ–expression consists of an optionally typed variable and either an statement or an expression. Furthermore, the statement expressions respectively the expressions are extended by evaluation-expressions, the λ–expressions, and instances of functions.
The concrete syntax in this paper of the λ–expressions is oriented at [Goe10], while the concrete syntax of the function types and closure evaluation is oriented at [lam10].
The optional type annotations [***type***] are the types, which can be inferred by the type inference algorithm.

Definition 1 (Types). *Let Ty_p be a set of Java 5.0 types ([GJSB05], Section 4.5), where BTV is an indexed set of bounded type variables. Then the set of Java_λ types $\mathsf{Type}_{TS}(\ BTV\ )$ is defined by*

- $Ty_p \subseteq \mathsf{Type}_{TS}(\ BTV\ )$
- *For* $ty, ty_i \in \mathsf{Type}_{TS}(\ BTV\ )$

$$\#\, ty\, (ty_1, \ldots, ty_n) \in \mathsf{Type}_{TS}(\ BTV\ )^1$$

Example 1. We consider the class `Matrix`.

```
class Matrix extends Vector<Vector<Integer>> {

  op = #{ m -> #{ f -> f(Matrix.this, m) } }

}
```

[1] Often function types $\#\, ty\, (ty_1, \ldots, ty_n)$ are written as $(ty_1, \ldots, ty_n) \ \rightarrow\ ty$.

`op` is a curried function with two arguments. The first one is a matrix and the second one is a function which takes two matrices and returns another matrix. The function `op` applies its second argument to its own object and its first argument. The function `op` is untyped. The first argument `m` and the second argument `f` are also untyped. The first idea for a correct typing could be that `m` gets the type `Matrix` and `f` gets $\#\,\texttt{Matrix}\,(\texttt{Matrix},\texttt{Matrix})$, which mean that the function `f` has the type $\#\,\#\,\texttt{Matrix}\,(\texttt{Matrix},\texttt{Matrix})\,(\texttt{Matrix})$.

The main difference between Java_λ and the corresponding core of Java 8 [Goe11] is the typing of λ–expressions. While in Java 8 the types are given as functional interfaces (Java interfaces with one method) in Java_λ the types of λ–expressions are given as real function types.

3 Implementation

In the following context it is described how to implement the algorithm **WTYPE** in Haskell. The background was explained in the introduction (Section 1). The algorithm for the Java_λ itself is given in [Plü11].

3.1 Abstract syntax

The data-structure for a class is given as

```
data Class = Class(SType, --name
                   [SType], -- extends
                   [IVarDecl], -- instancevariables
                   [FunDecl]) -- functiondeclarations
```

The first argument is the class-name, the second argument the super-class, respectively the implemented interfaces, the third argument the list of instancs variables, and the fourth argument the function declarations.

```
data FunDecl = Fun(String, Maybe Type, Expr)
```

A function is declared by its name, an optionally type and an expression. The optionally type will be inferred by the type-inference algorithm.
We consider only the new construtions of the data-structures `Expr` for expressions and `StmtExpr` for statement-expressions. The data-structure `Stmt` for statements is unchanged.

```
data Expr = Lambda([Expr], Lambdabody)
          | InstFun(Expr, String, String)
          | ...

data Lambdabody = StmtLB(Stmt)
                | ExprLB(Expr)
```

An expressions could be a λ–expression, the first argument is a list of parameters and the second argument, the λ–body, is either a statement or an expression.
The other considered constructor is the instance of a function. The first argument is the expression, which represents the class-instance, which comprises the function. The second argument represents the class name. This is necessary, as the algorithm allows no overloading. The third argument finally is the function name.

```
data StmtExpr = Eval(Expr, [Expr])
              | ...
```

The first argument of the constructor `Eval` is an expression, which represents a function. The second argument is a list of arguments. `Eval` stands for the evaluation of the functions application to the arguments.

Example 2. The abstract syntax of the class `Matrix` (Example 1) is given as:

```
[Class(TC ("Matrix",[]),
       [TC ("Vector",[TC ("Vector",[TC ("Integer",[])])])],
       [],
       [Fun(
         "op",
         Nothing,
         Lambda([LocalOrFieldVar "m",],
                ExprLB(
                  Lambda([LocalOrFieldVar "f"],
                         ExprLB(StmtExprExpr(
                           Eval(LocalOrFieldVar "f",
                                [ThisStype "Matrix",
                                 LocalOrFieldVar "m"])))))))])]
```

3.2 Parser

The parser is defined by a HAPPY–File. HAPPY is the LR-parser-generating–tool of Haskell. The syntax is similar to yacc. In Figure 2 a part of the specification is given.
Against to yacc in HAPPY the commands of the rules are given as return-expressions. This means that no `$$` is necessary to return a value.
The function `divideFuncInstVar` divides declarations of instance-variables and functions, as in Java mixed declarations are allowed.
`FType` is the constructor for the function type, the representation of $\#\,rettype\,(argtypes)$.
`TypeSType` is the boxed representation of Java 5.0 types in the set of all types.

```
classdeclaration : CLASS IDENTIFIER classpara classbody
                   { Class(TC($2, $3), [], fst $4, snd $4) }

classbody        : LBRACKET RBRACKET  { ([], []) }
                 | LBRACKET classbodydeclarations RBRACKET
                   { divideFuncInstVar $2 ([], []) }

fundeclaration   : funtype IDENTIFIER ASSIGN expression SEMICOLON
                   { Fun($2, Just $1, $4)}
                 | IDENTIFIER ASSIGN expression SEMICOLON
                   { Fun($1, Nothing, $3)}

funtype          : SHARP funtypeortype LBRACE funtypelist RBRACE
                   { FType($2, $4) }

funtypeortype    : funtype { $1 }
                 | type { TypeSType $1 }
```

Fig. 2. Part of the Java_λ HAPPY–File

3.3 The function TYPE

The function **TYPE** introduces fresh type variables to each sub-term of the expressions and determines the coercions (subtype pairs). The function needs a set of type-assumptions and a unique number for the next fresh type variable. We encapsulate these in a monad (Figure 3).

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    ...

data M a = Mon((TypeAssumptions, Int) -> (a, (TypeAssumptions, Int)))

instance Monad (M) where
    return coe_lexpr = Mon(\ta_nr -> (coe_expr, ta_nr))
    (>>=) (Mon f1) f2 = Mon (\ta_nr ->
                             let (coe_lexpr, ta_nr') = f1 ta_nr
                             in getCont(f2 coe_lexpr) ta_nr')

getCont:: M a -> ((TypeAssumptions, Int) -> (a, (TypeAssumptions, Int)))
getCont (Mon f) = f
```

Fig. 3. Monad for the function **TYPE**

The function `return` encapsulates a pair (coercion set, expression) to a functions which takes a pair (type-assumptions, number) and returns the pair of pairs ((coercion set, expression), (type-assumptions, number)).

The function >>= (bind-operator) takes an encapsulated function and another function. The result is the encapsulated function, which concatenates both functions. The function getCont decapsulate the content of the monad.
For expressions, statements and statement-expressions in each case a function is needed, which takes an expression, a statement, or a statement-expression, respectively and returns a corresponding monad. A Java-program consists of different functions, which are declared by (λ–)expressions. Therefore an additional function is necessary, which filters the expressions and calls the TYPE–function. In Figure 4 a part is presented. The main principle of monadic application is

```
tYPEClass :: Class -> M (CoercionSet, Class)
tYPEClass (Class(this_type, extends, instvar, funs)) =
  let
    funexprlist = map (\(Fun(op, typ, lambdat) -> lambdat) funs
  in
    tYPEExprList funexprlist

tYPEExprList :: [Expr] -> M (CoercionSet, [Expr])
tYPEExprList (e : es) = (tYPEExpr e)
   >>= (\coe_lexpr1 -> (tYPEExprList es)
      >>= \coe_lexp2 ->
          return ((fst coe_lexp1) ++ (fst coe_lexp2),
                  (snd coe_lexp1) : (snd coe_lexp2)))
tYPEExprList [] = return ([], [])

tYPEExpr :: Expr -> M (CoercionSet, Expr)

...

tYPEStmtExpr :: StmtExpr -> M (CoercionSet, StmtExpr)

...

tYPEStmt :: Stmt -> M(CoercionSet, Stmt)
...
```

Fig. 4. The **TYPE**–function

shown in function tYPEExprList. First tYPEExpr is applied to the first expression. By the bind-operator >>= the result is introduced in the recursive call of tYPEExprList. Finally, the results of both are summarized in the result of the whole function by dividing the corecions and the typed expressions.

Example 3. If we apply tYPEClass to the class Matrix (Example 1), we get the set of coercions:

```
([(FType (TypeSType (TFresh "V3"),[TypeSType (TFresh "V2")]),
```

```
   TypeSType(TFresh "V1")),
  (FType (TypeSType (TFresh "V8"),[TypeSType (TFresh "V4")]),
   TypeSType (TFresh "V3")),
  (TypeSType (TFresh "V4"),
   FType(TypeSType (TFresh "V7"),
         [TypeSType(TFresh "V6"),TypeSType (TFresh "V5")])),
  (TypeSType (TFresh "V7"),TypeSType (TFresh "V8")),
  (TypeSType (TC ("Matrix",[])),TypeSType (TFresh "V6")),
  (TypeSType (TFresh "V2"),TypeSType (TFresh "V5"))]
```

and the typed class

```
class Matrix extends Vector<Vector<Integer>> {

  V1 op = # { (V2 m) -> # { (V4 f) -> (f).(Matrix.this, m) } };

}
```

In the abstract representation all typed sub-terms could be considered.

```
[Class(TC ("Matrix",[]),
       [TC ("Vector",[TC ("Vector",[TC ("Integer",[])])])],
       [],
       [Fun(
         "op",
         Just(TypeSType (TFresh "V1")),
         TypedExpr(
           Lambda([TypedExpr(LocalOrFieldVar "m",
                             TypeSType (TFresh "V2"))],
                  ExprLB(TypedExpr(
                    Lambda(
                      [TypedExpr(LocalOrFieldVar "f",
                                 TypeSType(TFresh "V4"))],
                      ExprLB(TypedExpr(StmtExprExpr(
                        TypedStmtExpr(
                          Eval(TypedExpr(LocalOrFieldVar "f",
                                         TypeSType(TFresh "V4")),
                               [TypedExpr(ThisStype "Matrix",
                                          TypeSType(TC ("Matrix",[]))),
                                TypedExpr(LocalOrFieldVar "m",
                                          TypeSType(TFresh "V2"))]),
                          TypeSType(TFresh "V8"))),
                        TypeSType(TFresh "V8")))),
                    TypeSType(TFresh "V3")))),
           TypeSType(TFresh "V1")))])]
```

3.4 The function MATCH

The function **MATCH** unifies the coercions and reduces them. The result is a substitution and a set of atomic (reduced) coercions. Atomic coercions consist of pairs of Java 5.0 types.

While in the original algorithm of Fuh and Mishra the ordinary unification is used, for Java$_\lambda$ our type unification [Plü09] is necessary. Our type unification processes also wildcard types.
In Figure 5 the data-structures of MATCH are presented. The type `Subst` repre-

```
type Subst = [(Type, Type)]
type EquiTypes = [[Type]]
data Rel = Kl | Kl_QM | Eq | Gr | Gr_QM
type CoercionSetMatch = [(Type, Rel, Type)]

--Monade
data M a = Mon((EquiTypes, Int) -> (a, (EquiTypes, Int)))
instance Monad (M) where
    return subst_aCoes = Mon(\eq_nr -> (subst_aCoes, eq_nr))
    (>>=) (Mon f1) f2 = Mon (\eq_nr ->
                             let (subst_aCoes, eq_nr') = f1 eq_nr
                             in getCont(f2 subst_aCoes) eq_nr')
getCont:: M a -> ((EquiTypes, Int) -> (a, (EquiTypes, Int)))
getCont (Mon f) = f
```

Fig. 5. Data-structure of **MATCH**

sents the substitution. The type `EquiTypes` is necessary for Java 5.0 types which can be considered as equivalent. `Rel` are the different relations, which are used. `QM` stands for question mark, the wildcard type in Java.
In the algorithm again a monad is used. `(EquiTypes, Int)` is the pair of the equivalent types and the number of the next fresh type variable. `subst_aCoes` is the result, a substitution and a set of atomic coercions.

The algorithm itself consists of five cases:

```
mATCH :: CoercionSetMatch -> CoercionSetMatch -> FC
         -> M(Subst, CoercionSetMatch)

-- decomposition
mATCH aCoes ((FType(ret1, args1), Kl, FType(ret2, args2)):coes) fc = ...

-- reduce
mATCH aCoes ((TypeSType(TC(n1, args1)),rel,
             TypeSType(TC(n2, args2))):coes) fc = ...

-- expansion
mATCH aCoes ((TypeSType(TFresh(name)), rel,
             FType(ret2, args2)) : coes) fc = ...

-- atomic elimination
mATCH aCoes (((TypeSType(TFresh(name))), rel,
```

```
            javafivetype) : coes) fc = ...

-- recursion base
mATCH aCoes [] fc = (return ([], aCoes))
```

Decomposition: The function-type constructor is erased and the arguments respectively the result types are identified.
Reduce: The type-constructors `n1` and `n2` are reduced.
Expansion: The fresh type variable `name` is expanded, such that the type can be unfied with the function type on the right hand side.
Atomic elimination: The types are introduced in the set of equivalent types.

`FC` respresents the finite closure of the extends-relation [Plü07].

Example 4. `mATCH` applied to the coercions of Example 3 gives:

The substitution:

```
[(TypeSType (TFresh "V14") ↦
  FType(TypeSType (TFresh "V21"),
        [TypeSType (TFresh "V22"),TypeSType (TFresh "V23")])),
 (TypeSType (TFresh "V12") ↦
  FType(TypeSType (TFresh "V18"),
        [TypeSType (TFresh "V19"),TypeSType (TFresh "V20")])),
 (TypeSType (TFresh "V4") ↦
  FType(TypeSType (TFresh "V15"),
        [TypeSType (TFresh "V16"),TypeSType (TFresh "V17")])),
 (TypeSType (TFresh "V9") ↦
  FType (TypeSType (TFresh "V13"),
         [FType (TypeSType (TFresh "V21"),
                 [TypeSType (TFresh "V22"),TypeSType (TFresh "V23")])])),
 (TypeSType (TFresh "V3") ↦
  FType (TypeSType (TFresh "V11"),
         [FType (TypeSType (TFresh "V18"),
                 [TypeSType (TFresh "V19"),TypeSType (TFresh "V20")])])),
 (TypeSType (TFresh "V1") ↦
  FType (FType (TypeSType (TFresh "V13"),
         [FType (TypeSType (TFresh "V21"),
                 [TypeSType (TFresh "V22"),TypeSType (TFresh
                 "V23")])]),
         [TypeSType (TFresh "V10")]))],
```

If we apply the substitution to the typed in the typed program `Matrix`, we get:

```
class Matrix extends Vector<Vector<Integer>> {

  ##V13(#V21(V22, V23))(V10)
    op = # { (V2 m) -> # { (#V15(V16, V17) f) -> (f).(Matrix.this, m) } };

}
```

The set of atomic coercions:

```
[(TypeSType (TFresh "V2"),Kl,TypeSType (TFresh "V5")),
 (TypeSType (TC ("Matrix",[])),Kl,TypeSType (TFresh "V6")),
 (TypeSType (TFresh "V7"),Kl,TypeSType (TFresh "V8")),
 (TypeSType (TFresh "V21"),Kl,TypeSType (TFresh "V18")),
 (TypeSType (TFresh "V19"),Kl,TypeSType (TFresh "V22")),
 (TypeSType (TFresh "V20"),Kl,TypeSType (TFresh "V23")),
 (TypeSType (TFresh "V18"),Kl,TypeSType (TFresh "V15")),
 (TypeSType (TFresh "V16"),Kl,TypeSType (TFresh "V19")),
 (TypeSType (TFresh "V17"),Kl,TypeSType (TFresh "V20")),
 (TypeSType (TFresh "V15"),Kl,TypeSType (TFresh "V7")),
 (TypeSType (TFresh "V6"),Kl,TypeSType (TFresh "V16")),
 (TypeSType (TFresh "V5"),Kl,TypeSType (TFresh "V17")),
 (TypeSType (TFresh "V11"),Kl,TypeSType (TFresh "V13")),
 (TypeSType (TFresh "V8"),Kl,TypeSType (TFresh "V11")),
 (TypeSType (TFresh "V10"),Kl,TypeSType (TFresh "V2"))])
```

3.5 The function SOLUTIONS

The function CONSISTENT in the original algorithm is in our approach substituted by the function SOLUTIONS. CONSISTENT determines iteratively all possible solutions until it is obvious, that there is a solution. The result is then true, otherwise false. We extend this algorithm such that all possible solutions are determined.

```
sOLUTIONS :: [(Type, Rel, Type)] -> FC -> [[(Type, Type)]]
```

The input is the set of atomic corecions and the finite closure of the extends-relation. The result is the list of correct substitutions.
The algorithm itself has two phases. First all type variables are initialized by '*'. Then in some iterations steps over all coercions all correct instatiations are determined. The result is a list of substitutions, where all type variables, which are not in relation to a non-variable type, are remained instantiated by '*'. These variables can be instantiated by any type, only constraints are given by the coercions.

Example 5. The completion of the `Matrix` example is given by the application of `sOLUTIONS` to the result of `mATCH` (Example 4). There are three different solutions. Applied to the typed program we get:

```
class Matrix extends Vector<Vector<Integer>> {

##V13(#V21(Matrix, V23))(V10)
  op = #{ (V2 m) -> #{ (#V15(Matrix, V17) f) -> (f).(Matrix.this, m) } };

}

class Matrix extends Vector<Vector<Integer>> {
```

```
  ##V13(#V21(Vector<Vector<Integer>>, V23))(V10)
    op = #{ (V2 m) -> #{ (#V15(Matrix,V17) f) -> (f).(Matrix.this, m) } };

}

class Matrix extends Vector<Vector<Integer>> {

 ##V13(#V21(Vector<Vector<Integer>>, V23))(V10)
  op = #{ (V2 m) ->
         #{ (#V15(Vector<Vector<Integer>>,V17) f) -> (f).(Matrix.this,m) } };

}
```

The type variables `V13`, `V21`, `V23`, `V10`, `V2`, `V15`, and `V17` are not in relation to a non-variable type. This means that these types can be instantiated by an type, but there are coercions, which contrains the possible instatiations. E.g.

```
(TypeSType (TFresh "V10"),Kl,TypeSType (TFresh "V23"))
(TypeSType (TFresh "V21"),Kl,TypeSType (TFresh "V13"))
(TypeSType (TFresh "V2"),Kl,TypeSType (TFresh "V17"))
(TypeSType (TFresh "V15"),Kl,TypeSType (TFresh "V13"))
(TypeSType (TFresh "V10"),Kl,TypeSType (TFresh "V2"))
```

If we compare this result with the assuption in Example 1, we recognize, that this result is more principal. On the one hand the type of `m` is a type variable and on the other hand the first argument of `f` could be `Matrix` and `Vector<Vector<Integer>>`.

4 Conclusion and Future Work

In this paper we presented the implementation of the adapted Fuh and Mishra's type inference algorithm **WTYPE** to Java_λ. We gave the implementation in Haskell. We presented the parser done by the generating tool HAPPY and the functions **TYPE**, **MATCH**, and **SOLUTIONS**, where **TYPE** and **MATCH** are implementated by a state monad.The result is a well-typing. Well-typings are unknown in Java so far. Constrains of type variables, as the corecions in our approach, can be given in Java by bounds of parameters of classes and functions. A bound can only be a non-variable type. This means to introduce well-typings in the Java type system, the concept of bounds should be extended.
Finally, we show, how the `Matrix` example could be implemented, with extended bounds.

```
class Matrix extends Vector<Vector<Integer>> {

  <V10 extends V23, V21 extends V13, V2 extends V17, V15 extends V13,
   V10 extends V2, V23, V13, V17>
  ##V13(#V21(Matrix, V23))(V10)
    op = #{ (V2 m) -> #{ (#V15(Matrix, V17) f) -> (f).(Matrix.this, m) } };
```

}

References

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. Proceedings 2nd European Symposium on Programming (ESOP '88), pages 94–114, 1988.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java^{TM} Language Specification. The Java series. Addison-Wesley, 3rd edition, 2005.

[Goe10] Brian Goetz. State of the lambda. 10 October 2010.

[Goe11] Brian Goetz. State of the lambda. December 2011.

[lam10] Project lambda: Java language specification draft. 2010. Version 0.1.5.

[Plü07] Martin Plümicke. Typeless Programming in Java 5.0 with wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, 5th International Conference on Principles and Practices of Programming in Java, volume 272 of ACM International Conference Proceeding Series, pages 73–82, September 2007.

[Plü09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers, volume 5437 of Lecture Notes in Artificial Intelligence, pages 223–240. Springer-Verlag Heidelberg, 2009.

[Plü11] Martin Plümicke. Well-typings for Java$_\lambda$. In Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ '11, pages 91–100, New York, NY, USA, 2011. ACM.

User-Defined Shape Constraints in SAC

Fangyong Tang and Clemens Grelck

Institute of Informatics
University of Amsterdam
Science Park 904
1098XH Amsterdam, Netherlands
f.tang@uva.nl c.grelck@uva.nl

Abstract. We propose a method called user-defined constraints specifically for shape-generic multi-dimensional array programming. Our proposed technique allows programmers to make implicit constraints in the domain and codomain of functions explicit. This method can help compilers to generate more reliable code, improve performance through better optimization and improve software documentation.
We propose and motivate a syntax extension for the functional array language SaC and describe steps to systematically transform source-level constraints into existing intermediate code representations. We discuss ways of statically resolving constraints through aggressive partial evaluation and propose some form of syntactic sugar that blurs the line between user-defined constraints and fully-fledged dependent types.

1 Introduction

SaC (Single Assignment C) is an array programming language that supports shape-generic programming[1], i.e., functions may accept argument arrays with statically unknown size in a statically unknown number of dimensions. This generic array programming style brings many software engineering benefits, from ease of program development to ample code reuse opportunities.

However, generic array programming also introduces some subtle pitfalls. Many array operations are characterized by implicit shape constraints on parameters or return values. For example, matrix multiplication requires the second axis of the first parameter to be as long as the first axis of the second parameter; in element-wise arithmetic (e.g. sums and products of two arrays) it is often desirable to ask for shape equality of arguments. If programmers do not express implicit constraints in the code, merely depending on inferring shape information of parameters by compiler (which is really hard) is not enough to generate reliable executable code. Insufficient information about relationship of function parameters and result values is a problem. Program execution may encounter runtime errors, e.g. out-of-bound array indexing, if implicit constraints are violated, or program execution may simply yield erroneous results.

How to express constraints of functions is a question. For instance, programmers may add conditional code around function applications or in the definition

of the applied function. However, both approaches have their disadvantages. In the first case, programmers write redundant code for each function application, which violates software engineering principles. In the second case, compilers have little opportunity to prove constraints, and most constraints have to be left for dynamic checks. The lack of distinction between functionally relevant and constraint checking code in both scenarios further complicates a compiler's job.

Therefore, it is useful if programmers can explicitly supply constraints in code. We propose user-defined constraints that explicitly annotate relations between parameters and/or return values of functions. This approach facilitates program error detection and helps to infer more precise type information for smooth optimization. Using this approach, we obtain the following benefits:

1. User-defined constraints can help a compiler to generate more reliable code. For instance, a generic function *vector add* takes two vectors of the same size as parameters and yields a new vector whose values are the sums of the corresponding elements of the two argument vectors. In the absence of shape constraints the only way to avoid potential out-of-bound indexing into one of the argument vectors is to generate a vector whose length equals that of the shorter of the two argument vectors. With the proposed user-defined shape constraints, we instead could exlicitly restrict the domain of the *vector add* function.
2. Using annotated code can improve performance through better optimization. If the constraints can be statically resolved, corresponding code will be removed, and resolved constraints can be propagated to facilitate further optimization.
3. Showing constraints of function, to some extent, can represent software documentation. Since relationships of return values or parameters are given, it helps programmers to better understand code, e.g. generic function definition of matrix multiplication with user defined constraints.

The main contributions of this paper are:

1. a new method called user-defined constraints that explicitly express constraints of functions;
2. an outline of syntax of the innovative method;
3. a discussion about where and how to assert the constraints;
4. a practical case is given to show how this approach works.

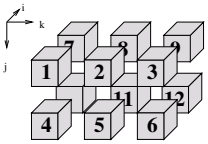
The paper is structured as follows. We start with a brief introduction to the array calculus and the type system of SaC in Section 2. Section 3 presents the motivation of the proposed method. A detailed description and syntax of user-defined constraints is given in Section 4. Section 5 describes code transformation of constraints, and Section 6 discusses where and how to insert constraints into intermediate code. We sketch out some syntactic sugar for shape constraints in Section 7. Related work is discussed in Section 8. Finally, we draw conclusions in Section 9.

2 SAC — Single Assignment C

As the name suggests, SAC is a functional language with a C-like syntax. We interpret sequences of assignment statements as cascading let-expressions while branches and loops are nothing but syntactic sugar for conditional expressions and tail-end recursion, respectively. Details can be found in [1, 2]. The main contribution of SAC, however, is the array support, which we elaborate on in the remainder of this section.

2.1 Array calculus

SAC implements a formal calculus of multidimensional arrays. As illustrated in Fig. 1, an array is represented by a natural number, named the *rank*, a vector of natural numbers, named the *shape vector*, and a vector of whatever data type is stored in the array, named the *data vector*. The rank of an array is another word for the number of dimensions or axes. The elements of the shape vector determine the extent of the array along each of the array's dimensions. Hence, the rank of an array equals the length of that array's shape vector, and the product of the shape vector elements equals the length of the data vector and, thus, the number of elements of an array. The data vector contains the array's elements in a flat contiguous representation along ascending axes and indices. As shown in Fig. 1, the array calculus nicely extends to "scalars" as rank-zero arrays.



rank: 3
shape: [2,2,3]
data: [1,2,3,4,5,6,7,8,9,10,11,12]

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

rank: 2
shape: [3,3]
data: [1,2,3,4,5,6,7,8,9]

[1, 2, 3, 4, 5, 6]

rank: 1
shape: [6]
data: [1,2,3,4,5,6]

42

rank: 0
shape: [ ]
data: [42]

Fig. 1. Calculus of multidimensional arrays

2.2 Array types

The type system of SaC is polymorphic in the structure of arrays, as illustrated in Fig. 2. For each base type (`int` in the example), there is a hierarchy of array types with three levels of varying static information on the shape: on the first level, named AKS, we have complete compile time shape information. On the intermediate AKD level we still know the rank of an array but not its concrete shape. Last not least, the AUD level supports entirely generic arrays for which not even the number of axes is determined at compile time. SaC supports overloading on this subtyping hierarchy, i.e. generic and concrete definitions of the same function may exist side-by-side.

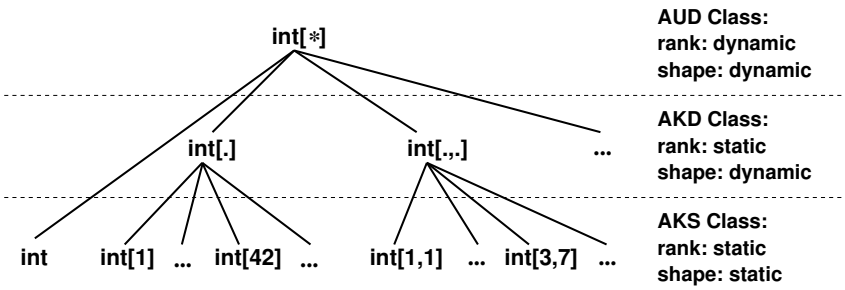


Fig. 2. Type hierarchy of SaC

2.3 Array operations

SaC only provides a small set of built-in array operations, essentially to retrieve the rank (`dim(`*array*`)`) or shape (`shape(`*array*`)`) of an array and to select elements or subarrays (*array*`[`*idxvec*`]`). All aggregate array operations are specified using with-loop expressions, a SaC-specific array comprehension:

```
with {
  ( lower_bound <= idxvec < upper_bound) : expr;
   ...
  ( lower_bound <= idxvec < upper_bound) : expr;
}: genarray( shape, default)
```

This with-loop defines an array of shape ***`shape`*** whose elements are by default set to the value of the ***`default`*** expression. The body consists of multiple (disjoint) *partitions*. Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to induction variables in for-loops. We call the specification of such

an index set a *generator* and associate it with some potentially complex SaC expression that is evaluated for each element of the generator-defined index set.

Based on these with-loops and the support for rank- and shape-invariant programming SaC comes with a comprehensive array library that provides similar functionality as built-in operations in other array languages and beyond. For the SaC programmer these operations are hardly distinguishable from built-ins, but from the language implementor perspective the library approach offers better maintainability and extensibility.

3 Motivation

In this section, we use matrix multiplication ("matmul" for short) as example to show how to define a function on different levels of abstraction according to the hierarchy of array types of SaC. We demonstrate the existence of implicit constraints and motivate the desire for explicit constraints.

3.1 AKS function definition

Fig. 3 shows how to define matrix multiplication for arrays of fixed shapes. We first transpose array B (line 3) before we use a single with-loop for the standard text book definition of matrix multiplication (line 4-7).

```
1 float[3,3] matmul(float[3,5] A, float[5,6] B)
2 {
3   BT = transpose(B);
4   C = with{
5         ([0,0]<=[i,j]<[3,6]):sum(A[i]*BT[j]);
6       }:genarray([3,6],0f);
7   return(C);
8 }
```

Fig. 3. Function definition of matrix multiplication with AKS type

Even though it seems that this function definition reveals the well-known shape constraints on function parameters, it merely does so for the concrete argument shapes, but it does not reveal the underlying general relationship. What's more, this code only works for one pair of array shapes, which very much limits code reuse. Therefore, a more generic solution is needed.

3.2 AKD function definition

As introduced in Section 2, an AKD type, e.g. `float[.,.]`, defines the number of axes, but leaves the extent along each axis open. Matrix multiplication can be written in a shape-generic way, as shown in Fig. 4.

```
1  float[.,.] matmul(float[.,.] A, float[.,.] B)
2  {
3    BT = transpose(B);
4    a0 = shape(A)[0];
5    b1 = shape(B)[1];
6    C = with{
7          ([0,0]<=[i,j]<[a0,b1]):sum(A[i]*BT[j]);
8        }: genarray([a0,b1],0f);
9    return(C);
10 }
```

Fig. 4. Function definition of matrix multiplication with AKD type

In this generic code, we did not supply constraints for function. What happens if we apply the function to wrongly shaped arguments? We give mismatched matrices for this function, for instance, two arrays with shape of [3,5] and [6,3], respectively. Element-wise vector multiplication (e.g. `A[i]*BT[j]`) is defined in the SaC standard library. In the absence of shape constraints it is defined to yield a vector whose length equals the minimum of the lengths of the argument vectors. This way, out-of-bound array indexing can be avoided at the expense of an additional minimum computation and obfuscation of the shape relationships, which is detrimental for further optimization. For our definition of matrix multiplication this means that we avoid a runtime error and make function `matmul` total, but we do this outside the mathematical definition of matrix product.

3.3 AUD function definition

Array programs are composed from general-purpose array operations, and many operations are rank-generic, i.e. they are applicable to arrays with arbitrary number of axes. AUD types like `float[*]` encompass all arrays of a given base type regardless of their structure. Here we generalise *matmul* to *innerproduct*. As shown in Fig. 5, function `innerproduct` takes two arrays `A` and `B`, where the last axis of A has to be as long as the first axis of B.

Values `al` and `bl` are computed and used in line 10-11 for taking specific axis. Line 5-6 get first `al` and last `bl` axis from two shape of parameters respectively by using `drop` function for simple, which comprise shape of final result in line 7. Line 8-13 do inner product computation with with-loop.

Functions `take(sv,a)` and `drop(sv,a)` take and drop as many elements from array `a` as indicated by shape vector `sv`, respectively. Each element of `sv` corresponds to one axis of `a` starting from the leftmost one. For positive components of `sv`, the elements are taken or dropped from the "beginning", i.e., starting with index 0. Otherwise, they are taken or dropped from the "end", i.e., starting from the maximum legal index of the corresponding axis.

```
1  float[*] innerproduct(float[*] A, float[*] B)
2  {
3    al = dim(A)-1;
4    bl = dim(B)-1;
5    a = drop([-1],shape(A));
6    b = drop([1],shape(B));
7    v = a++b;
8    C = with{
9          (0*v<=iv<v) {
10           m = take([al],iv);
11           n = take([-bl],iv);
12           var = sum(take(m,A)*take(n,B));}:var;
13       }:genarray(v,0f);
14   return(C);
15 }
```

Fig. 5. Function definition of inner product with AUD type

3.4 Implicit constraints

Program errors are common in software systems and often hard to detect, especially the implicit constraints discussed above, which may cause runtime errors, e.g. out-of-bound array access. What's more, such constraints are enforced by means of dynamic checks that carry a performance penalty. To avoid this problem, programmers could add additional conditional statements either in the callee or in the caller function, but none of them can express constraints in function level. In the next section, we introduce a user-defined constraints technology, which can express constraints explicitly. Through existing aggressive partial evaluation in SaC, these user-defined constraints may be resolved and removed by compiler to avoid runtime checks.

4 Proposed user-defined constraints

The constraints in program vary, which not only focus on parameter, but also return value in generic functions. In this section, a compiler technology called user-defined constraints is introduced. This technique allows programmers to express constraints of functions explicitly. While still writing code in a generic way, programmers can add their desired annotations between function declaration and function body, which indicates compiler constraints on parameters and/or return value without losing generalization.

According to different usage of constraints, the technology can be divided into two parts:

1. Precondition (user-defined constraints on parameters): programmers can specify the limitations of parameters using dedicated expression;
2. Postcondition (user-defined constraints on return values): programmers can also add constraints on shape or rank information of return value.

We introduce two key words for SaC: `where` is used to express the constraints between function parameters while `assert` is used to indicate shape or rank information of return values.

4.1 User-defined constraints on parameters

In this part, we use `where` to express constraints between parameters. For function in Fig. 4, because the ranks of parameters are known and equal, programmers just need to restrict specific axes of parameters, as demonstrated in Fig. 6. Using this method, the function `matmul` documents the constraint clearly: the second axis of the first parameter has to be as long as the first axis of the second parameter.

```
1 float[.,.] matmul(float[.,.] A, float[.,.] B)
2 where(shape(A)[1]==shape(B)[0])
3 {
4   /*Computation*/
5 }
```

Fig. 6. Constraint on dedicated axis of parameters in AKD type

Fig. 6 shows basic usage of user-defined constraints on specific axes of parameters. There are some other abstract restrictions of function parameters. For example, the generic array addition function in Fig. 7 requires the shape of array A to be equal to the shape of array B. Just using the above method — express axis-wise equality — would be tedious and even impossible for AUD-style functions. Therefore, we directly use the equality of shape vectors of parameters to express this constraint, as shown in Fig. 7.

```
1 int[*] addition(int[*] A, int[*] B)
2 where(shape(A)==shape(B))
3 {
4   /*Computation*/
5 }
```

Fig. 7. Constraint on shape of parameters

In Fig. 6, `==` is used to represent equality of two scalar values, however, here, vectors appear as operands. In SaC, `==` is more general operation which can indicate the equality not only between scalars but also vectors and arrays. How `==` will be transformed will be illustrated in next section. `shape(A)==shape(B)` means shape of parameters A and B are the same. At the same time, it indicates that the ranks of the two array have to be equal as well, which is an implicit constraint inflicted by the user-defined constraint.

Compared with matrix multiplication, our function `innerproduct` in Fig. 5 has a similar restriction: the last axis of the first parameter and the first axis of the second parameter must coincide in length. The built-in function `take` is used to retrieve elements from the shape vector. We can extend builtin functions in `where` domain as well, as illustrated in Fig. 8.

```
1  float[*] innerproduct(float[*] A, float[*] B)
2  where(take([-1],shape(A))==take([1],shape(B)))
3  {
4    /*Computation*/
5  }
```

Fig. 8. Constraints on dedicated axis of parameter in AUD type

Some applications may have other kinds of restrictions among parameters. For instance, function `specialfun` in Fig. 9 has a constraint that from the second element of the shapes of arrays A and B to the sixth, each element must be equal. These constraints can be expressed as in Fig. 9. The function `tile(sv,ov,a)` takes a *tile* of shape `sv` from `a` starting at offset `ov` from array `a`.

```
1  int[*] specialfun(int[*] A, int[*] B)
2  where(tile([2],[4],shape(A))==tile([2],[4],shape(B)))
3  {
4    /*Computation*/
5  }
```

Fig. 9. Constraints on arbitrary shape domain

4.2 User-defined constraints on return values

In some cases, programmers may also want to restrict shapes or ranks of return values or express shapely relationships between multiple return values. Such information can prove indispensable to resolve further constraints later in the code. To express constraints involving return values of functions we need a way to refer to return values in constraint expressions. However, in SaC functions can have multiple return values. As shown in Fig. 10, function `addsub` has a comma-separated list of return types in front of the function name and return-statement in the function definition likewise contains a comma-separated list of expressions. At first glance, we could be tempted to use variables occuring in the return-statement. However, in practice we must be able to expresss constraints without having access to the complete function definition. To solve both issues, we extend the syntax of function definitions to explicitly name return values, just as function parameters.

Constraints among return values For generic functions, like `addsub` in Fig. 10, shape and rank of parameters and return values are all unknown. `assert` expression is used to indicate that the shapes of the two return values should be equal.

```
1  int[*] x, int[*] y addsub(int[*] A, int[*] B)
2  where(shape(A)==shape(B))
3  assert(shape(x)==shape(y))
4  {
5    x = A+B;
6    y = A-B;
7    return(x,y);
8  }
```

Fig. 10. Constraints among return values

Constraints between parameters and return values

In practice, most return values have some shape relationship with parameters. Take matrix multiplication as an example. The shape of return value contains two elements. The first element is equal to the first axis of the first parameter, and the second element is equal to the second axis of the second parameters. This relationship can be represented as demonstrated in Fig. 11.

```
1  float[.,.] x matmul(float[.,.] A, float[.,.] B)
2  where(shape(A)[1]==shape(B)[0])
3  assert(shape(x)[0]==shape(A)[0],
4         shape(x)[1]==shape(B)[1])
5  {
6    /*Computation*/
7  }
```

Fig. 11. Constraints between parameters and return values on dedicated axis

4.3 Syntax of user-defined constraints

We introduce a concrete syntax for user-defined constraints as shown in Fig. 12, `+`, `-`, `*` is extended in SaC to be used in scalar and array operations. Primitive functions `shape`, `dim`, `tile`, `drop`, `take` and `concatenate` are used in this syntax. We simplify the syntax and leave out irrelevant SaC features.

5 Transformation of user-defined constraints

Implicit constraints can be expressed explicitly using user-defined constraints. Before making use of this kind of information, however, we must first transform

$$
\begin{array}{lll}
fundef & \Rightarrow & rets\ funid\ \ (\ args\ )\ \ \lceil\ constraint\ \rceil\ body \\
rets & \Rightarrow & (\ ret\lceil\ ,ret\ \rceil^{*} \mid \lceil\ \textbf{void}\ \rceil) \\
args & \Rightarrow & (\ param\lceil\ ,param\ \rceil^{*} \mid \lceil\ \textbf{void}\ \rceil) \\
ret & \Rightarrow & param \\
 & \mid & type \\
param & \Rightarrow & type\ id \\
constraint & \Rightarrow & \lceil\ \textbf{where}\ \ (\ exprs\ )\ \rceil\lceil\ \textbf{assert}\ \ (\ exprs\ )\ \rceil \\
exprs & \Rightarrow & expr\lceil\ ,expr\ \rceil^{*} \\
expr & \Rightarrow & info\_scalar\ equality\ expr\_scalar \\
 & \mid & info\_vector\ equality\ expr\_vector \\
info\_scalar & \Rightarrow & \textbf{dim}\ \ (\ id\ ) \\
 & \mid & \textbf{shape}\ \ (\ id\ )\ [const] \\
info\_vector & \Rightarrow & \textbf{shape}\ \ (\ id\ ) \\
 & \mid & \textbf{take}\ \ (\ expr\_vector, shape\ (\ id\ )\ ) \\
 & \mid & \textbf{drop}\ \ (\ expr\_vector, shape\ (\ id\ )\ ) \\
 & \mid & \textbf{tile}\ \ (\ expr\_vector, expr\_vector,\ \textbf{shape}\ \ (\ id\ )\ ) \\
expr\_scalar & \Rightarrow & num\ op\ info\_scalar \\
 & \mid & info\_scalar \\
 & \mid & num \\
 & \mid & id \\
expr\_vector & \Rightarrow & num\ op\ info\_vector \\
 & \mid & info\_vector \\
 & \mid & concatenate\ (\ expr\_vector, expr\_vector\ ) \\
 & \mid & array \\
array & \Rightarrow & [expr\_scalar\lceil\ ,expr\_scalar\ \rceil^{*}] \\
equality & \Rightarrow & == \\
op & \Rightarrow & + \\
 & \mid & - \\
 & \mid & *
\end{array}
$$

Fig. 12. Syntax of user-defined constraints of SaC

these abstract representations into more concrete code using primitive functions. Since `where` and `assert` constraints are comma-separated lists of expressions, we can distinguish each sub-constraint expression according to comma, for example, the function in Fig. 10 has an `assert` expression that contains three constraints.

It is convenient to represent user-defined constraints by primitive functions in SaC. At compile time, compiler do code transformation to represent user-defined constraints by primitive function. In the following, the detail of transformation will be given. Some of `where` and `assert` expressions can be transformed into

builtin functions directly. However, some user-defined constraints may introduce new implicit constraints. For example, if programmers use `shape(A)[2]` to query for the third axis of array `A` while `A` actually is a matrix (i.e. 2-dimensional), the index is out of bounds. Therefore, the compiler must represent these implicitly induced constraints as further explicit constraints to ensure user-defined constraints are valid.

In the following, we will take `where` expression as example to show the representation of constraints.

5.1 Equality of dimensionality

Even though in previous function definition, we did not introduce the constraint on rank of array, it may be used in some applications. Lets assume there is a constraint on rank of parameters A and B in a generic function, i.e. `dim(A)==dim(B)`.

Here, `==` is a user-defined overloaded function. SaC compiler resolve overloading of operations into vectors or scalar operations depending on type of arguments. The scalar operation (`eq_SxS`) just evaluates the equality of scalar values, while the vector operation (`eq_VxV`) introduces an implicit constraint, i.e., it only compare elements of vectors without checking the size of vectors, because this tacitly assumes their lengths coincide.

Since the return value of dim(A) is scalar, the constraint can be represented by primitive function `eq_SxS`.

where(dim(A)==dim(B)) $\Longrightarrow$ where(eq_SxS(dim(A),dim(B)))

5.2 Equality of dedicated axis

The constraint (`shape(A)[i]==shape(B)[j]`) can be transformed into the equality of two scalar values as well. To make the example more general, integers i, j, m, n etc. are used to indicate the index of shape of parameters in the user-defined constraints. However, be aware of an implicit restriction underlying the constraint, i.e. the index of `shape(A)` and `shape(B)` have to be valid. The following transformation make the implicit constraints explicitly.

```
where(shape(A)[i]==shape(B)[j]) ==>
  where(gt_SxS(dim(A),i);
        gt_SxS(dim(B),j);
        eq_SxS(shape(A)[1],shape(B)[j]))
```

`gt_SxS` is a builtin function which evaluate to true if its first argument is greater than the second one.

5.3 Equality of shape

Abstract user-defined constraints is more powerful at representing equality of dedicated axis. However, because of its abstract and generic there are some

implicit constraints underlying. To evaluate equality of shape, we should compare the rank of two parameters first, if that hold, we evaluate their shape equality. The primitive function `eq_VxV` evaluates whether two vectors are equal or not.

```
where(shape(A)==shape(B)) ==>
  where(eq_SxS(dim(A),dim(B)),
        eq_VxV(shape(A),shape(B)))
```

5.4 Arbitrary equality of shape axes

Arbitrary equality of shape axes is much more powerful and complicated. The complexity and further implicit constraints make transformation much more intricate. Implicit constraints underlying in primitive function `tile` is shown as following. Here, `ge_SxS` is a builtin function which evaluate to true if its first argument is greater than or equal to the second one.

```
where(tile([i],[j],shape(A))==tile([m],[n],shape(B))) ==>
  where(eq_SxS(i,m),
        ge_SxS(dim(A),i+j),
        ge_SxS(dim(B),m+n),
        eq_VxV(tile([i],[j],shape(A)),
               tile([m],[n],shape(B))))
```

6 Wrap user-defined constraints into program

At some point, user-defined constraints must be inserted into code. There are several points should be considered to make constraints more useful when assert. First, only if `where` constraints evaluate to true, callee function will be executed, otherwise the program should terminate with an error. Second, if `assert` constraints do not hold, program should terminate with an error message. Third, user-defined constraints can be resolved as far as possible. Fourth, constraints should be accessible to compiler optimization, and knowledge gained by evaluating constraints should be propagated as far as possible.

6.1 Where to insert constraints into code

There are several possible strategies to insert and thus evaluate `where` and `assert` constraints. They can be inserted into caller function or callee function or even both.

Precondition Intuitively, it is feasible to insert the where expression into the caller function before applying arguments to callee function. Reasons to do this are as follows:

1. The callee function will not be executed if `where` constraints do not hold, and the program will be terminated.
2. More importantly, putting constraints into the caller function is crucial for the compiler to resolve constraints statically because shape information on arguments can only be acquired in the caller function.

However, we have to consider another aspect: can these constraints be used in callee function? No matter whether `where` expression evaluates to true or not, for callee function, these constraints will be treated as statical knowledge, since once callee function is evaluated, it means the precondition holds. We need to insert these preconditions into the callee function as statically known knowledge, which can benefit further optimizations. Therefore, where constraints should be inserted into both caller and callee functions.

Postcondition The `assert` expressions include constraints among return values and constraints between return values and parameters. For `assert` constraints among return values, it is better to insert constraints into callee function since shape information of return values could be only acquired inside the callee function body, which is a prerequisite for the compiler to resolve any constraint statically. For `assert` constraints between arguments and parameters, it seems a bit complicated, because information of return values and parameters can be acquired from callee and caller function, respectively.

In fact, the situation regarding postconditions is quite similar to that of preconditions. Whenever program execution returns to a caller function, this sheer fact means that the postcondition holds. Thus, we can use the postcondition as static knowledge in the caller function subsequent to the function application itself..

Therefore, we use redundant insertion methods to insert both `where` and `assert` constraints into both callee function and caller function.

6.2 How to insert constraints into code

Insert by conditionals The intuitive way to insert constraints is wrapping user-defined constraints and their implicit constraints into a conditional statement. Using conditional statement, callee function will be evaluated only if parameters satisfy constraints. Since the `where` expressions are inserted into caller function, it is possible to use implicit knowledge of parameters in caller function to resolve constraints as well. However, with respect to fourth point mentioned in the beginning of this section, this method is not optimal, because the result of constraints is only available within the scope of conditional. Optimization or partial evaluation cannot exploit this additional knowledges. Therefore, this kind of insertion is not good enough.

Using explicit evidence To avoid that problem, we use explicit evidence[3] to insert constraints and weave these contracts into the dataflow. We implement this by using a primitive function `guard` [4], which return true if its argument evaluates to true, otherwise it terminates further evaluation and reports the error.

```
1  int[.,.] bar(int x, int y, int z, int w)
2  {
3    ...
4    A = with{([0,0]<=[iv]<[x,y]):1f;}: genarray([x,y]);
5    B = with{([0,0]<=[iv]<[z,w]):2f;}: genarray([z,w]);
6    C = matmul(A,B);
7    ...
8    return C;
9  }
```

Fig. 13. Function definition of bar which call function matmul

As discussed above, the resolved constraints have to become a property of program which can be used in the further evaluation. Therefore, we introduce a new kind of guard function named `guard_hold`. It always evaluates to true and mainly serves to provide additional knowledge for further evaluation. Before code generation `guard_hold` annotations will uniformly be removed.

```
1  int[.,.] bar(int x, int y, int z, int w)
2  {
3    ...
4    A = with{([0,0]<=[iv]<[x,y]):1f;}: genarray([x,y]);
5    B = with{([0,0]<=[iv]<[z,w]):2f;}: genarray([z,w]);
6
7    g1 = guard(gt_SxS(dim(A),0);
8    g2 = guard(gt_SxS(dim(B),1);
9    g3 = guard(eq_SxS(shape(A)[1],shape(B)[0]));
10   A = after_guard(A,g1,g2,g3);
11
12   C = matmul(A,B);
13
14   g4 = guard_hold(gt_SxS(dim(A),0),
15   g5 = guard_hold(gt_SxS(dim(B),1),
16   g6 = guard_hold(gt_SxS(dim(C),1),
17   g7 = guard_hold(eq_SxS(shape(C)[0],shape(A)[0]),
18   g8 = guard_hold(eq_SxS(shape(C)[1],shape(B)[1]));
19   C1 = after_guard(C,g4,g5,g6,g7,g8);
20   ...
21   return C1;
22 }
```

Fig. 14. Function bar after code transformation

Let's assume there is a piece of code in Fig. 13 that call function matmul, as defined in Fig. 11. The caller function in Fig. 14 contains addtional guard

function, i.e. `g1`, `g2` are constraints introduced by `g3`. In Fig. 15, `g1`, `g2`, `g3` become a static knowledge for callee function, which is used for further evaluation. We use `guard_hold` to indicate its argument evaluates to true. `after_guard` takes two or more arguments, and the result of it is the first argument if all consecutive arguments evaluate to true; otherwise, it terminate evaluation.

If the constraints can be resolved at compile time, the guard function may be removed partially by partial evaluation. Here, `guard_hold` will stay and conditional removed after partial evaluation.

```
1  float[.,.] x matmul(float[.,.] A, float[.,.] B)
2  {
3    g1 = guard_hold(gt_SxS(dim(A),0);
4    g2 = guard_hold(gt_SxS(dim(B),1);
5    g3 = guard_hold(eq_SxS(shape(A)[1],shape(B)[0]));
6    A  = after_guard(A,g1,g2,g3);
7
8    BT = transpose(B);
9    a0 = shape(A)[0];
10   b1 = shape(B)[1];
11   C = with {
12     ([0,0]<=[i,j]<[a1,b0]):sum(A[i] * BT[j]);
13     }:genarray([a0,b1],0f);
14
15   g6 = guard(gt_SxS(dim(A),0);
16   g7 = guard(gt_SxS(dim(B),1);
17   g8 = guard(gt_SxS(dim(C),1);
18   g9 = guard(eq_SxS(shape(C)[0],shape(A)[0]),
19   g10 = guard(eq_SxS(shape(C)[1],shape(B)[1]));
20   C1 = after_guard(C,g6,g7,g8,g10);
21
22   return C1;
23 }
```

Fig. 15. Function matmul after code transformation

7 Syntactic sugar for shape constraints

Our shape constraints are a syntactically restricted form of Boolean expressions. While this approach provides a certain degree of generality, phrasing of shape constraints can often be significantly simplified by adding some syntactic sugar to the specification of types.

The matrix multiplication example discussed throughout the paper illustrates the common need to access extents of argument arrays along individual axes. In Fig. 6 we used the term `shape(A)[1]` to express this. Fig. 16 shows the same

example with a little bit of syntactic sugar: by using identifiers instead of dots in AKD types, we can elegantly bind identifiers to the extents of argument arrays along relevant axes.

```
1 float[.,.] matmul(float[.,a1] A, float[b0,.] B)
2 where (a1==b0)
3 {
4   /*Computation*/
5 }
```

Fig. 16. Matrix multiplication of Fig. 6 with syntactic for sugar constraints

We can even go one step further and express equality constraints by repeatedly using the same identifier instead of anonymous dots. Fig. 17 demonstrates this with a complete description of the inherent shape constraints of matrix multiplication, without making any use of any explicit `where` and `assert` constraints.

```
1 float[x,z] matmul(float[x,y] A, float[y,z] B)
2 {
3   /*Computation*/
4 }
```

Fig. 17. Fully sugared expression of matrix multiplication shape constraints

Desugaring of the above examples into explicit constraints is a fairly straightforward preprocessing step.

8 Related work

Programming errors are common in software system and hard to detect. Much research attention has been paid to error detection. One popular approach is *design by contract* proposed by Bertrand Meyer [5, 6], which is widely used both in object-oriented programmings language [7, 8] and functional programming languages [3, 9, 10]. But none of them has concrete shape restrictions on arrays as described in this paper.

Interpreted array programming languages like APL[11], J [12] are dynamically typed. When the interpreter encounters an array operation, it checks whether its arguments are proper, if so, performs computation by invoking native implementation, otherwise, aborts program with error message. Without a priori static analysis makes bugs hard to find, and What's more, this design decision has a considerable runtime impact [13]. Our proposed compiler technology can counter these issues. The SAC compiler tries to statically resolve all user-defined

constraints. If the constraints do not hold, it reports error message. For the constraints that can not be resolved at compile time, we will leave them to dynamic check.

Our work indeed is similar to Qube [14, 15], a programming language that combines the expressiveness of array programming with the power of dependent type. We try to express implicit constraints explicitly and resolve constraints statically. However, our approach is much more programmer friendly and expressiveness, and we eliminate the effort of learning new logic with its theorem prover.

Another related work was carried out by Herhut et al.[4], which focuses on checking domain constraints for built-in functions. In contrast our approach targets general user-defined functions and thus more general constraints.

To resolve these constraints, we may adopt *symbiotic expressions* [16] which is a method for algebraic simplification within a compiler. Even though currently *symbiotic expression* in SaC are compiler-generated expressions, we may be able to reuse some technology to support our user-defined constraints.

9 Conclusion

This paper proposes a compiler technology called user-defined constraints that allows programmers to express shape constraints on parameters (`where`) and return values (`assert`) explicitly in function level. It help compiler to generate more reliable executable code by restrict function definition, improve performance through better optimization and provide a means for software documentation, which helps programmers to better understand code. Using this approach, programmers neither need to add redundant conditional code in caller function, nor add constraints in callee function. User-defined constraints will be transformed into primitive functions in SaC, and then inserted into program properly according to their type.

We mainly focus our presentation on introducing user-defined constraints and its implicit constraints and code transformation in this paper. It remains as future research to investigate how compiler resolves more constraints statically and effectively, and whether this technology brings the properties of our type system close to strongly typed system based on dependent types.

References

1. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427
2. Grelck, C.: Single assignment c (sac): High productivity meets high performance. In Z. Horvath, V.Z., ed.: 4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary. Volume 7241 of Lecture Notes in Computer Science., Springer (2012) to appear.

3. Xu, D.N.: Extended static checking for Haskell. In: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell. Haskell '06, New York, NY, USA, ACM (2006) 48–59
4. Herhut, S., Scholz, S.B., Bernecky, R., Grelck, C., Trojahner, K.: From Contracts Towards Dependent Types: Proofs by Partial Evaluation. In Chitil, O., Horváth, Z., Zsók, V., eds.: 19th International Symposium on Implementation and Application of Functional Languages (IFL'07), Freiburg, Germany, Revised Selected Papers. Volume 5083 of Lecture Notes in Computer Science., Springer (2008) 254–273
5. Meyer, B.: Eiffel: The language and environment. Prentice Hall Press, 300 (1991)
6. Meyer, B.: Applying design by contract. Computer **25** (1992) 40–51
7. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass–java with assertions. Electronic Notes in Theoretical Computer Science **55** (2001) 103–117
8. Plosch, R.: Design by contract for Python. In: Software Engineering Conference, 1997. Asia Pacific... and International Computer Science Conference 1997. APSEC'97 and ICSC'97. Proceedings, IEEE (1997) 213–219
9. Findler, R., Felleisen, M.: Contracts for higher-order functions. ACM SIGPLAN Notices **37** (2002) 48–59
10. Blume, M., McAllester, D.: Sound and complete models of contracts. Journal of Functional Programming **16** (2006) 375–414
11. Iverson, K.: A programming language. In: Proceedings of the May 1-3, 1962, spring joint computer conference, ACM (1962) 345–351
12. Iverson, K.: J Introduction and Dictionary, New York, NY, USA. (1995)
13. Bernecky, R.: Reducing computational complexity with array predicates. In: ACM SIGAPL APL Quote Quad, ACM (1998) 39–43
14. Trojahner, K.: QUBE–Array Programming with Dependent Types. PhD thesis, University of Lübeck, Lübeck, Germany (2011)
15. Trojahner, K., Grelck, C.: Dependently typed array programs don't go wrong. Journal of Logic and Algebraic Programming **78** (2009) 643–664
16. Bernecky, R., Herhut, S., Scholz, S.: Symbiotic expressions. Implementation and Application of Functional Languages (2011) 107–124

Tyre-Check-I:

Low-level Type Inference for Reduceron Code Safety

Marco P. Perez * Colin Runciman

University of York, UK

{mppc500,colin.runciman}@york.ac.uk

Abstract

We present a type-inference system for Reduceron *template code* – untyped compiled code. *No extra annotations* are added to the template code in order to obtain the type information. The purpose of this type-inference system is to guarantee a form of type safety at loading time before executing the template code. We use *mutated* programs to measure the effectiveness of a prototype checker. We use a unification algorithm supporting unification of infinite trees. The infinite trees are represented as set of equations.

1 Introduction

The Reduceron [9] is a graph reduction machine that uses the template instantiation model. The Reduceron evaluates template code, which is compiled from a core language called F-lite. In this paper we propose a type-inference system for template code. We use as baseline the template code described in §3 of [9]. Our goal is to provide a type-inference system *without requiring any extra annotations in the template code.*

This idea of low-level code verification is not new. In the 80's the work of Clutterbuck [2] is one of the precursors in low-level code verification. In the 90's Proof-Carrying Code [10] and Typed Assembly Language TAL [8] are both seminal works under this area. More recently, and more similar to our approach, are the type-inference systems described in [4] and [13].

During the compilation process, F-lite code is transformed to equivalent template code (See §2 [9]). A well established compiler might be trusted to

*Supported by scholarship from Conacyt.

produce safe code. However, template code could be modified at some stage after compilation and before execution, or template code might be supplied from some other source. Faulty code could produce a failure at run-time such as an invalid memory access.

Our idea to is protect against such failures by *examination of the low-level code of the Reduceron.* Taking the ideas of PCC [10] and TAL [8, 1] in which mobile code is verified before its execution, we investigate in this paper how to build a type-inference system to guarantee safety of the template code under an unsafe scenario –eg. code produced by hand, or code intercepted and altered before it reaches the target machine. Unlike [4] and [13], we do not use any extra annotations but infer types directly from template code.

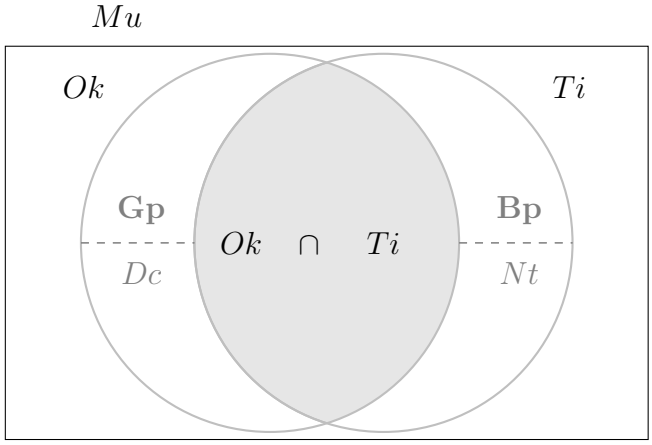


Figure 1: Classification of mutations (Mu): Well-behaved programs (Ok), well-typed programs (Ti), good programs (Gp), bad programs (Bp), programs with dead code (Dc) and, programs with non-termination (Nt).

But how can we verify the effectiveness of low-level type-checking as a means of ensuring safe execution? Milner [7] offered the slogan *well-typed programs don't go wrong.* In the Reduceron context the meaning of "going wrong" is captured by *emulating* the execution of the template code and trapping run-time failures, including a conservative approximation of non-termination. By looking at *mutations* of test programs we measure the effectiveness of type-checking for safety.

In Figure 1, we illustrate this idea. There is a universe of mutated template-code programs Mu. We are interested in the relationship between two sets of programs : Ok and Ti. The set Ok contains well-behaved programs, and the set Ti contains well-typed programs. Ideally, Ti and Ok would be the same sets, but this is hardly attainable. The set Gp contains

well-behaved "good" programs that are rejected by the type-checker. We want Gp to be small. The set Bp contains ill-behaved "bad" programs that are accepted by the type-checker. We also want Bp to be small. Finally, we identify *dead-code* (Dc) and *non-termination* (Nt) by conservative approximations.

§2 describes F-lite and template code. The contributions of this paper are: (1) an approach to type inference by examination of *Reduceron template code*, in §3; (2) an encoding for primitives and case-table types that can support inference of recursive algebraic data types (using graph-terms for types to solve problems related with infinite types), also in §3; (3) preliminary results measuring the effectiveness of checking, in §4. Our measure of effectiveness uses mutation testing. Finally, in §5 we summarise our findings so far and give some ideas to be explored in the future.

2 High-Level Code and Compiled Code

2.1 F-lite : High-Level Code

F-lite is a minimalist-core lazy functional language based on a subset of Haskell [9]. Its syntax is given in Figure 2. F-lite makes use of uniform pattern matching (there is a well known correspondence with case expressions – which we use in this paper). F-lite supports let expressions, to bind expressions to variables, not to patterns. F-lite uses primitive integer operations `((+), (-), (<=), (==))`. F-lite is untyped. But type-inference for F-lite source is possible, and well-typed programs written in F-lite can be executed by Haskell implementations. Bindings in let expressions may be recursive. Every program includes a definition $main = e$ where e is an integer expression.

2.2 Template Code : Compiled Code

F-lite can be compiled to Reduceron template code. Such code could also be generated by other tools or written by hand. The code is defined in Figure 3 and is closely based on [9]. A program p is coded as a series of templates. Each template `t` is a 3-tuple: arity ari , spine application ap, and a list of off-spine applications aps. Each application is a list of atoms $\vec{a}$. There are seven kinds of atoms :

$$
\begin{array}{lcll}
e & ::= & \vec{e} & \text{(Application)} \\
 & | & case \;\; e \;\; of \;\; \vec{a} & \text{(Case Expression)} \\
 & | & let \;\; \left\{\vec{b}\right\} \;\; in \;\; e & \text{(Let Expression)} \\
 & | & n & \text{(Integer)} \\
 & | & x & \text{(Variable)} \\
 & | & p & \text{(Primitive)} \\
 & | & f & \text{(Function)} \\
 & | & C & \text{(Constructor)} \\
 & | & < \vec{f} > & \text{(Case Table)} \\
a & ::= & C\vec{x} \rightarrow e & \text{(Case Alternative)} \\
b & ::= & x = e & \text{(Let Binding)} \\
d & ::= & f\vec{x} = e & \text{(Function Definition)}
\end{array}
$$

Figure 2: F-lite core syntax.

$$
\begin{array}{lcl}
a & ::= & FUN \;\; i \;\; j \;\; | ARG \;\; i \;\; | \;\; PTR \;\; i \;\; | \;\; CON \;\; i \;\; j \\
 & | & INT \;\; i \;\; | \;\; PRI \;\; s \;\; | \;\; TAB \;\; i \;\; j \;\; k \\
ap & ::= & \vec{a} \\
aps & ::= & \vec{ap} \\
t & ::= & \langle ari, ap, aps \rangle \\
p & ::= & \vec{t} \\
ari & ::= & num
\end{array}
$$

Figure 3: A Basic version of Reduceron template code.

FUN i j	Reference to a function of arity i at template address j. The main function is `FUN 0 0`, a function of arity zero at address zero.
ARG i	A reference to the i^{th} argument of the enclosing template function.
PTR i	A pointer to an off-spine application. Where i is the index of the application.
CON i j	A constructor of some (unknown) algebraic data type. It is the j^{th} constructor of the type and has arity i.
INT i	A numeric value i.
PRI s	A primitive s, where $s \in \{$`(+)`, `(-)`, `(<=)`, `(==)`$\}$.
TAB i j k	A table referencing j templates, representing case alternatives, starting at template address i, and k is number of free variables occurring in any alternative.

```
--F-lite
lenNat xs = case xs of {Cons x ys -> Succ (lenNat ys) ;
                        Nil       -> Zero }

--After Scott/Jansen encoding. Eliminate case expressions.
lenNat xs = xs <consCase,nilCase>
consCase x ys t  = Succ (lenNat ys)
nilCase      t  = Zero

--Reduceron template code (Represented as Haskell data).
lenNat=[(1,[ARG 0, TAB 1 2 0],[]),
        (3,[CON 1 0,PTR 0], [[FUN 1 0, ARG 1]]),
        (1,[CON 0 1],[]) ]
```

Figure 4: The `lenNat` function at various stages of compilation.

2.3 From F-lite to Template Code : An example

In Figure 4 the example `lenNat` computes the length of a given list. It returns the result in a data type Nat instead of an Int. If the expression `xs` evaluates to `consCase` then the result is the application of the successor to the application of lenNat recursively. If the case subject evaluates to `nilCase` then it simply gives the natural number `Zero` as result.

The case expression is transformed to a Scott/Jansen encoding [9]. The introduction of a case table `<consCase, nilCase>` is performed to allow the use of only one argument instead of many.

Finally, we have the code actually executed by the Reduceron. The template code is shown as a list of templates. Template zero is the main function. Template one is the function `lenNat`. Templates two and three represent case alternatives. Note the case table `TAB 2 2 0` in template one. There are no free variables after table.

3 Inspecting Template Code

3.1 Finding Types I

One way to derive appropriate types is to look at each single atom in the template code, and try to give a type for each of them. To begin with, we shall consider template code with atoms restricted to `FUN`, `ARG`, `PRI`, `INT`, and `PTR`.

An atom `FUN i j`, refers to a function at address `j` with `i` arguments. Functional atoms prompt us to introduce types `T` $\rightarrow$ `T` as in Figure 5. An

$$
\begin{array}{lcl}
T & ::= & T \to T \mid V \mid Prim\ n \\
Int & ::= & Prim\ n\ \to Prim\ (n-1) \\
 & & where\ n\ >\ 0 \\
Int & \equiv & Prim\ 0
\end{array}
$$

Figure 5: Template Type Model 1 (TTM1).

atom `ARG i` is the i^{th} argument of a function. We give the type variable `V` to the argument for now. An atom `PTR i` is simply an indirection to an address `i`. The two last atoms are related. The atom `PRI "(+)"` has type `Prim 2`, where `2` is the number of arguments expected. If an argument is applied to `PRI "(+)"` then the type is reduced to `Prim 1`; if other argument is applied we get `Prim 0`. `Prim 0 Int` and `Int` are equivalent types. In any other context the type of `INT i` is just an `Int`. But in the presence of primitives, it is a function type denoted by `Prim n → Prim (n-1)`.

3.2 Finding Types II

In this section we explore how to derive types for comparison primitives `PRI "(<=)", PRI "(==)"`, case tables `TAB i j k`, and constructors `CON i j`. These are all related to a table type denoted by $\langle\ \vec{a}\ \rangle$ – a sequence of functions representing case alternatives. We shall need to extend our model to include new types for tables. Under the Scott/Jansen encoding the type of a constructor `CON i j` is a function of $i+1$ arguments. The $i+1^{th}$ argument is a case table. The alternative `j` binds to type $\alpha_1 \to \cdots \to \alpha_i \to \langle\rangle \to \beta$, the notation $\langle\rangle$ represents an arbitrary table. The final encoding for the type of the constructor `CON i j` is :

$$\alpha_1 \to \cdots \to \alpha_i \to \langle\ j \mapsto \alpha_1 \to \cdots \to \alpha_i \to \langle\rangle \to \beta\rangle \to \beta$$

In §3.1, we used type `Prim n` for arithmetic primitives. But now we need to distinguish between the result type of arithmetic and comparison primitives. We extend our model (see 6) from `Prim n` to `Prim n T`, where `T` is the result type. For arithmetic primitives $T = Int$, and for comparison primitives $T = Bool$.

So the type for `PRI "(<=)"` and `PRI "(==)"` is `Prim 2 Bool`. The type `Prim 0 Bool` is equivalent to `Bool`.

3.2.1 The Bool Type

The data type `Bool` in a high-level representation is given by `data Bool = False | True`. However, in template code, constructor atoms take the form

$$
\begin{array}{lcl}
T & ::= & T \to T \mid V \mid Prim\ n\ T \mid \langle\ \vec{\mathbf{a}}\ \rangle \\
Int & ::= & Prim\ n\ T \to Prim\ (n-1)\ T \\
 & & where\ n\ >\ 0 \\
Bool & \equiv & Prim\ 0\ Bool \\
\mathbf{a} & ::= & n \mapsto T
\end{array}
$$

Figure 6: Template Type Model 2 (TTM2).

of `CON a i` where `a` is the arity, and `i` is the index of the constructor in a given data type. The constructor `False` is encoded as `CON 0 0` and `True` as `CON 0 1`.

These constructors are *indistinguishable* from those for any other type whose first two constructors have arity zero. By looking at `CON 0 0` we only know that this is the first constructor of some *unknown* data type of arity zero. We can assign it the type $\langle\ 0 \mapsto \langle\rangle \to \alpha\rangle \to \alpha$. The meaning of this encoding is: a function from a table to some type α; the table has at least one alternative (zero), a function with just a table as argument and the result type α. In a similar way, we can deduce a type for `CON 0 1` which is $\langle\ 1 \mapsto \langle\rangle \to \alpha\rangle \to \alpha$.

Until now, we have found separately the types for `True` and `False`. The type of `Bool` is : $\langle\ 0 \mapsto \langle\rangle \to \alpha\ ;\ 1 \mapsto \langle\rangle \to \alpha\rangle \to \alpha$, is a function from a table of at least two alternatives, each with result type α, to an α result.

3.2.2 The Unification of Tables

Let us consider the problem of unification of two tables denoted by `t1` and `t2`. Suppose `t1` is $\langle\ 0 \mapsto \langle\rangle \to \alpha\rangle$, and `t2` is $\langle\ 1 \mapsto \langle\rangle \to \beta\rangle$. How can we unify those two tables? To unify them, we can treat the tables as extensible records [12], combining the alternatives of two compatible tables. By doing this, it is possible to unify the tables `t1` and `t2` as $\langle\ 0 \mapsto \langle\rangle \to \alpha\ ;\ 1 \mapsto \langle\rangle \to \beta\rangle$.

A more interesting problem is if `t1` is $\langle\ 0 \mapsto \langle\rangle \to T_0\ ;\ 1 \mapsto \langle\rangle \to T_1\rangle$, and `t2` is $\langle\ 1 \mapsto \langle\rangle \to U_1\rangle$. In order to unify `t1` and `t2`, the alternative types T_1 and U_1 must unify. In general *the alternatives with the same constructor index must unify.*

Now, consider when `t1` is $\langle\ 0 \mapsto \langle\rangle \to \alpha\ ;\ 1 \mapsto \alpha \to \beta \to \langle\rangle\rangle$, and `t2` is $\langle\ 1 \mapsto \langle\rangle \to \alpha\rangle$. As $\alpha \to \beta \to \langle\rangle$ and $\langle\rangle \to \alpha$ cannot be unified, unification of `t1` and `t2` fails. The arbitrary table $\langle\rangle$ *always* unifies with any other table.

3.3 Recursive Data Types

3.3.1 Recursive Type in the Argument

One challenge is the inference of recursive algebraic data types. There has been recent research in recursive type reconstruction [11], but it is oriented to a high-level language. Here, we outline how to infer a recursive argument type. To explain our idea we use the function `lenNat` described in Figure 4, a function that computes the length of a given list. Its result has the algebraic type `Nat` instead of the primitive `Int`.

From `template 0` we can see that `lenNat` is a function of one argument (by the arity `1`). So its type is : $a \to b$ for some a and b. From the application of `ARG 0` to `TAB 1 2 0` we infer that the argument is of some algebraic data type with two constructors. From the first alternative, `template 1`, we discover that it has two component arguments – the arity `3` denotes two arguments and one table (`consCase x ys t`). In addition, there is a recursive application of `template 0` to the second of its components represented by the application `[FUN 1 0, ARG 1]`. We assume that recursive argument types are *invariant*. The argument (`ARG 1`) of the function `FUN 1 0` and the argument `ARG 0` in `template 0` have the same type.

We use the pair term and the list of equations representing substitutions. Notice the term c is the recursive term in the expression.

$lenNat :: a \to b$

$$\begin{aligned} &where \\ &c = \langle\ 0 \mapsto e \to c \to \langle\rangle \to d\ ;\ 1 \mapsto \langle\rangle \to d\ \rangle \to d \\ &... \\ &b = h \end{aligned}$$

3.3.2 Recursive Type in the Result

What if the result has a recursive data type? Consider the same function `lenNat`. From `template 0` we see that it is a function of one argument by looking at arity `1`. So its type is of the form $a \to b$. We have already considered the type a. Let us now focus on the type b.

If we look at `template 1`, there is a construction `CON 1 0` applying to one argument. This argument is a function application `[FUN 1 0,ARG 1]` that makes a recursive call to the `template 0`. The application of `CON 1 0` to its recursive argument represented by `[FUN 1 0,ARG 1]` gives us a recursive data type constructor with one argument. So we can see that we have a recursive algebraic data type in the type result.

Then we may conclude that the result type is represented by the following term and the set of equations. Notice the term g, which represents the

recursion in the result type:

$lenNat :: a \to b$

$$
\begin{aligned}
&where \\
&c = \langle\ 0 \mapsto e \to c \to \langle\rangle \to d\ ;\ 1 \mapsto \langle\rangle \to d\ \rangle \to d \\
&g = \langle\ 0 \mapsto g\ \langle\rangle \to h\ ;\ 1 \mapsto \langle\rangle \to h\ \rangle \\
&h = g \to h \\
&b = h \\
&...
\end{aligned}
$$

3.3.3 Infinite Trees

In the example of lenNat we found that there are equations containing recursive references inside the data types. We explore the idea of finding the solution for infinite trees proposed by Colmerauer [3] as unification algorithm. In such representation types are terms *together with* associated recursive type equations. By using term-graphs we will be able to infer recursive types in a more general way. Even if in this example there is only one recursive call (one for the argument and other for the result type), we could have the case where multiple recursive terms in a set of equations. The infinite tree is a tree containing an infinite set of nodes, however Colmerauer's work is focused only in rational trees, which are trees that have a finite set of subtrees. The infinite tree is represented as a set of equations. The method proposed by Colmerauer has five transformations with several properties, two of our interest are :

- They preserve equivalence. Two systems of equations are equivalent if they have the same tree-solution.
- The system obtained as tree-solution is a 'simplified form' of the initial system.

One advantage of using the first property is that we can decouple the part of collection equations and the part of finding a solution. In our current implementation we use the solver when we collect the equations for a template, however we can collect the set of equations for a group of templates and then solve them. By using a simplified tree-solution (the second property) we can have all the type information for a template or a group of templates in a concise form. At this stage we are not concerned about the type error messages. Then, if there is at least one solution that means that the program is well-typed otherwise the program is ill-typed. We have extended the algorithm to allow the unification of tables which are described in §3.2.2, and the unification of primitives descibed in §3.1.

4 Experiments and Results

4.1 The Implementation

For the experiments we use a prototype (Tyre-Check) written in Haskell. Although our current implementation of recursive type inference is not complete, we can infer types when the argument has a recursive algebraic data type and in the result type in functions like length and lengthNat. We infer the types for reduceron in two stages, first we collect the equations or constraints from each template application. Then once we have a complete set of equations we call our solver. This approach of collecting and then solving is similar to the one found in [6]. If the solver has at least one solution then the template is well-typed. The information is stored in the type environment, which contains the list of function definitions along with the pair (term,sys), which represent the the type and the equations representing the substitutions.

The Reduceron machine (Dynamic-Checker) for our experiments is an evaluator written in Haskell, which is based in the operational semantics in §3 of the paper [9]. In the original version of the operational semantics for Reduceron, there is no way to distinguish the different sources of errors. In our implementation we have classified the faults in the operational semantics in order to detect the source of the errors during the execution of template code.

A conservative test detects when a program seems unlikely to terminate. Let k be the number of the steps to evaluate the original program before mutation. After $2 * k$ steps we assume that the mutated program will not terminate. The idea is to compare the set of programs that are *well-behaved* under evaluation(Dynamic-Checking) and the set of programs that are *well-typed* under our static checker (Tyre-Check). In addition we detect *dead code* by using a conservative test: if *mutant* programs don't use all the templates during the execution, the unused templates are assumed to be dead code.

4.2 Mutation

We use a simple test framework. We take a program that terminates and gives a correct result after evaluation. That program is mutated in order to produce similar programs. By mutation we mean :

- increasing or decreasing a numeric value in an atom – though numbers interpreted as naturals cannot fall below zero, or
- swapping two contiguous atoms, or

- deleting an atom.

The idea of mutation is not new, the main research in this field began in the 70's [5]. In our experiments we use mutation to emulate a situation where we have a valid program, and it is *altered* at some point between the compilation and the evaluation or a hand-coder makes a mistake. In contrast to random generation, mutation produces programs close to genuine code, and not arbitrary inventions.

4.3 Results

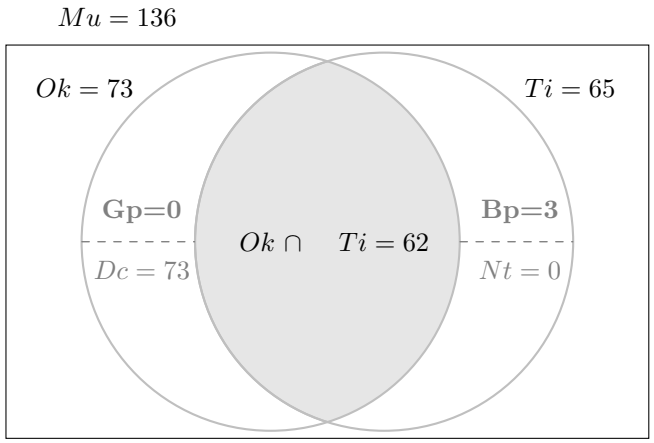


Figure 7: Classification of all *single* atomic mutations (136) of the `lenNat` program.

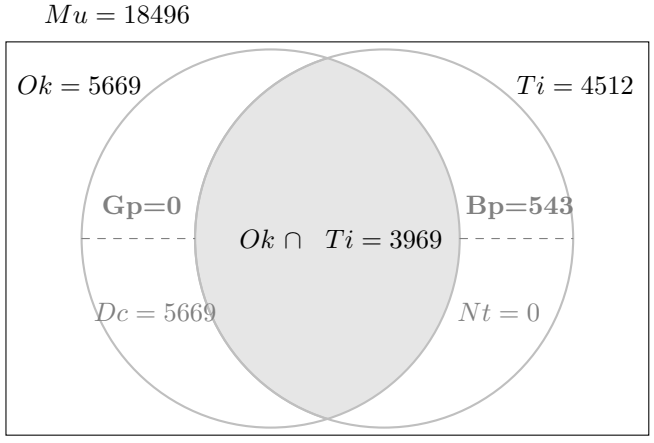


Figure 8: Classification of all *double* atomic mutations (18496) of the `lenNat` program.

Recalling the diagram of Figure 1, we present some actual set sizes in Figure 7. The mutated program is the `lenNat` program described in Figure 4. The type of mutation is single atomic mutation in numeric values.

For the first example we have a universe of 136 mutations represented by `Mu=136`. The set `Ok=73` is the obtained by using a dynamic-checking, and the set `Ti=65` is the result of our type-inference prototype. The number of programs in the intersection set `OK` $\cap$ `Ti` is 62. Zero good programs `Gp=0` wrongly stopped by the type-inference, and only three bad program not detected `Bp=3`. Recalling that our goal is to have small sets `Gp` and `Bp`. We also detect 73 cases of dead code `(Dc=73)` and zero cases of Non-termination `(Nt=0)`.

Figure 8 shows the effect of *double* atomic mutation. The double mutation is a mutation over a previous mutation. The results are similar in proportion with the ones presented in Figure 7. Most of the difference between `Ok` and `Ti` is accounted by dead code.

Most of the problems in the set of mutations `Bp` in both Figures 7 and 8 are because type-checking a mismatch in the arities. We think we can reduce the number of `Gp` and `Bp` mutants by small changes in our approach to type-checking. When the mutations are restricted to increments or decrements of numeric values, the gap between `Ok` and `Ti` is minor. But we intend to experiment more by using the other two type of mutations; deletion and swapping.

5 Conclusions and Future Work

Although Reduceron template code is untyped, we have shown that:

- It is feasible to type-check template code even without extra annotations. Trying to infer the types without extra annotations is quite hard. We have used one syntactic annotation in the atom `TAB`: in the operational semantics, the Reduceron only uses one argument in the `TAB` constructor –the address of the first alternative. For type-inference, we need to know how many alternatives there are in a case table, and how many free variables are involved. Our design decision was to provide two extra arguments in the `TAB` atom to encode such information. This is `TAB i j k` with the extra arguments `j` and `k`. Possibly there are other approaches to avoid this small extra encoding. However, we want static analysis to be simple.

- Mutations can be used to measure empirically the effectiveness checking. This idea of mutation fits well in our experimentation, because it

can emulate hundreds of programs altered by hand. Moreover, we can create mutations of mutations.

- There is a type encoding for Scott-encoded case analysis that can support inference of algebraic data types.

In our future work we plan to:

- Implement a fast and complete type-inference algorithm to work with more complex programs. This implementation will include the inference of recursive data types in result types. To provide a powerful and elegant approach, we will use term-graphs to represent types.

- Support installation at load time of new component functions in template code. This present new challenges, because we need to think for example, which are the allowed addresses to load a new component, and what invariants must be respected in order to maintain the safety of our run-time system.

- Use a minimal amount of extra annotations in the template code to increase the effectiveness of checking and to reduce its cost. One of the main objectives of Proof Carrying Code and TAL is to provide fast checkers in the consumer side. The trusted computing base must be kept as small as possible.

References

[1] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. Semantic Foundations for Typed Assembly Languages. *ACM Trans. Program. Lang. Syst.*, 32:7:1–7:67, 2010.

[2] D. L. Clutterbuck and B. A. Carré. The Verification of Low-level Code. *Softw. Eng. J.*, 3(3):97–111, 1988.

[3] A. Colmerauer. *Prolog and Infinite Trees. Logic Programming*, pages 231–252. Academic Press, 1982.

[4] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent Types for Low-level Programming. In *Proc. 16th European Conference on Programming (ESOP'07)*, pages 520–535, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[6] B.J Heeren, J Hage, and S.D Swierstra. Generalizing Hindley-Milner Type Inference Algorithms. 2002.

[7] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[8] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[9] Matthew Naylor and Colin Runciman. The Reduceron Reconfigured. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*, pages 75–86, 2010.

[10] George C. Necula. Proof-Carrying Code. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, 1997.

[11] Tom Schrijvers and Maurice Bruynooghe. Polymorphic Algebraic Data Type Reconstruction. In *Proc. 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '06)*, pages 85–96. ACM, 2006.

[12] M. Wand. Type Inference for Record Concatenation and Multiple Inheritance. In *Logic in Computer Science, 1989. LICS '89, Proc., Fourth Annual Symposium on*, pages 92 –97, jun 1989.

[13] Simon Winwood and Manuel Chakravarty. Singleton: A General-Purpose Dependently-Typed Assembly Language. In *Proc. 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '11)*, pages 3–14, 2011.

An Embedded Type Debugger

Kanae Tsushima
Ochanomizu University
tsushima.kanae@is.ocha.ac.jp

Kenichi Asai
Ochanomizu University
asai@is.ocha.ac.jp

Abstract

This paper presents how to build a type debugger *without* implementing any dedicated type inferencer. Previous type debuggers required their own type inferencers apart from the compiler's type inferencer. The advantage of our approach is threefold. First, by *not* implementing a type inferencer, it is guaranteed that the debugger's type inference never disagree with the compiler's type inference. Secondly, we can avoid the pointless reproduction of a type inferencer that should work precisely as the compiler's type inferencer. It is cumbersome and error-prone for a large language. Thirdly, our approach is robust to updates of the underlying language. The key observation of our approach is that the interactive type debugging, as proposed by Chitil, does not require a type inference tree but only a tree with a certain simple property. We identify the property and present how to construct a tree that satisfies this property using the compiler's type inferencer. The property guides us how to build a type debugger for various language constructs. We first describe the technique using simply-typed lambda calculus. We then extend it with let polymorphism, weak polymorphism, objects, and modules to see how our technique scales. Finally, we describe the type debugger for OCaml we have implemented that uses OCaml's own type inferencer.

Categories and Subject Descriptors D.2.5 [*Programming Languages*]: Testing and Debugging—Debugging aids; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors—Debuggers

General Terms types, debugging

Keywords type debugging, algorithmic debugging

1. Introduction

To write a well-typed program is not always easy, especially for novice programmers. Although a compiler gives us an error message when a type error occurs, it is not straightforward to infer and understand why the type error arises. Furthermore, the root cause of the type error can be far from the place reported by the compiler as a type error.

Can compiler's error message point out the programmer's intended source of type errors? Let us consider the following example:

```
let rec f g lst = match lst with
  | [] -> []
  | fst :: rest -> (g fst) :: (f g rest) in
(f 1 [2;3;4]) @ [5;6;7]
```

In this program, two boxed parts have a type conflict causing a type error. The first argument `g` of the function `f` is used as a function in `(g fst)`, but an integer is passed as `g` in `(f 1 [2;3;4])`. Because a function type `a -> b` cannot be unified with `int`, a type error occurs. When we compile this program using the OCaml compiler version 3.12.1, we obtain the following error message:

```
(f 1 [2;3;4]) @ [5;6;7]

Error: This expression has type int but
an expression was expected of type 'a -> 'b
```

Since the function `f` as shown above is a `map` function and is typable, the compiler accepts the definition and proceeds to type checking of the last line, where it infers that `1` passed to `f` must be a function. Although the above error message correctly points out a type conflict found in the input program, it is not always the source of the type error. If the programmer who wrote this erroneous program intended that the function `f` was actually a `map` function, the above error message probably points out the source of the type error. However, he could have intended that the function `f` received a number `g` and a list of numbers `lst` and returned a list of numbers whose elements were increased by `g`. In such a case, the source of the type error is not in `1`, but in `(g fst)` in the body of `f`. The following table summarizes the situation:

	intended behavior of `f`	the place to be fixed
(1)	`map` (applies a function to all the elements of a list)	`1 in (f 1 [2;3;4])`
(2)	`add_g` (increases all the elements by a given number)	`(g fst)`

Since the same program has two different fixes according to the programmer's intention, it is impossible for a single error message to point out the source of the type error. We need to somehow obtain programmer's intention. For this reason, type debuggers [1, 16] are designed to receive programmer's intention interactively. Following Chitil's work [1], we have implemented a type debugger for a subset of OCaml [17]. The following session shows how the type debugger locates the source of the type error using programmer's input:[1]

```
A type error occurred in (f 1 [2;3;4])!
Is the type of f ('a -> 'b) -> 'a list -> 'b list ?
> no
Is the type of g in match expression ('a -> 'b) ?
> no
Is the type of [] 'a list ?
```

[1] The actual messages are somewhat different, since we provide an Emacs interface and expressions that the debugger mentions are highlighted.

```
> yes
Error located : (g fst)
```

Using the type debugger, programmers can locate the source of type errors simply by answering a series of questions. We have used this type debugger for introductory OCaml course and found it useful. However, the conventional approach to type debuggers has at least three problems. First, to enable interactive searching of the source of type errors, it employs a different type inference algorithm than the one used in standard compilers (such as Hindley-Milner type inference). Thus, the definition of type errors could be different in a compiler and in a type debugger. Even if we could show their equivalence, there is a second problem that we need to implement the type inferencer by ourselves to construct a type debugger. Although it is not difficult to construct a type inferencer for a small language, it becomes quickly cumbersome and error-prone to support larger languages. In the type debugger we implemented [17], we supported only basic OCaml constructs that were necessary for the OCaml course. Even if we could implement a type inferencer for a large language, there arises a third problem: if the type inferencer of the underlying language is updated, we have to update the type inferencer for the updated language accordingly.

The key observation of our approach is that the interactive type debugging does not require a type inference tree but only a tree with a simple property: decomposition of a well-typed expression must not contain any ill-typed expressions. Once we recognize this property, it is easy to construct a tree that satisfies this property. We use the compiler's type inferencer to obtain the type of each subexpressions. We describe the overview of our type debugger and why the property is required in Section 2.

The property guides us how to build a type debugger for various language constructs. We first describe the technique using simply-typed lambda calculus (Section 3). We then extend it with let polymorphism (Section 4), weak polymorphism (Section 5), objects (Section 6), and modules (Section 7) to see how our technique scales. Finally, we describe the type debugger for OCaml we have implemented that uses OCaml's own type inferencer (Section 8).

The thesis of this paper is that it is possible and also practical to write a type debugger by piggy-backing the built-in type inferencer of an existing compiler. The contributions of this paper that support the thesis are summarized as follows:

- We identify the property required of the type tree to construct an interactive type debugger and show how to construct the tree using the compiler's type inferencer.
- We describe how to handle a number of issues required for the practical use: simple types, let polymorphism, weak polymorphism, objects, and modules.

Related work is in Section 9 and the paper concludes in Section 10.

2. Overview and the necessary property

In this section, we introduce the overall framework of the type debugger and show the property required of a type tree used in the type debugger.

2.1 Locating the source of type errors

In the previous section, we saw that we need programmer's intention to correctly locate the source of type errors. What information precisely do we need to locate it? Let us consider the previous example again. The type of the first argument `g` of `f` is summarized as follows:

	type of `g` (in `(g fst)`)	type of `1` (in `(f 1 [2;3;4])`)
compiler	`('a -> 'b)`	`int`
(1)	`('a -> 'b)`	`('a -> 'b)`
(2)	`int`	`int`

The compiler infers that the type of `g` is `'a -> 'b` and the type of `1` is `int`. In (1), the programmer intends that the function `f` is `map`. In this case, he thinks the type of `g` is `'a -> 'b` as the compiler infers. However, `1` is passed as the first argument `g` of `f` in `f 1 [2;3;4]`. Since `1` is of type `int`, it contradicts with the programmer's intended type `'a -> 'b`. A possible fix to this case would be to replace `1` with an expression of type `'a -> 'b`, such as `fun x -> x + 1`. In (2), the programmer intends that the function `f` adds `g` to each element of its second argument. In this case, he thinks that `1`, which is the first argument of `f`, should be an integer, as the compiler infers. However, in the body of `f`, its first argument `g` is used as a function, which contradicts with the programmer's intention. A possible fix to this case would be to replace `g fst` with `g + fst` treating `g` as an integer. In either case, we observe conflict between the type inferred by the compiler and the programmer's intended type at the source of the type error. In other words, if we can detect conflict between compiler's type and programmer's intended type, we can locate the source of the type error. Thus, the task of an interactive type debugger is to find the conflict of the two types in the input program.

2.2 Standard type inference tree

To locate the source of type errors, we first need a type tree that holds a type inferred by a compiler for each subexpression. Naturally, we want to use the tree constructed in the type inferencer of the compiler. However, we cannot use the standard type inference tree.

To see why, let us consider a simple erroneous program `((fun x -> fun y -> x :: y) 1) ["is"; "int"]`. In this program, we suppose programmer's intended program is `((fun x -> fun y -> x :: y) "1") ["is"; "int"]` which evaluates to `["1"; "is"; "int"]`. We show a type inference tree by the compiler in Figure 1 and programmer's intended type inference tree in Figure 2.

By comparing these two trees, we first find a type conflict at the conclusion of the boxed part. In Figure 1, it has type `int list -> int list`, while the corresponding part in Figure 2 has type `string list -> string list`. Thus, the source of the type error must be in the boxed area. However, we cannot further identify the source of the type error, because a type of an expression in consideration depends on types of its context. For example, consider the subexpression `(fun x -> fun y -> x :: y)` in the boxed part. Since it has type `int -> int list -> int list` which is in conflict with programmer's intended type, we could incorrectly conclude that the source of the type error resides in this expression. However, the most general type for the expression is in fact `'a -> 'a list -> 'a list`, and is consistent with the intended type. Thus, the source of the type error does not exist in this expression.

The type variable `'a` here is instantiated to `int`, because the expression is applied to an integer. In other words, a type of an expression in the standard type inference tree depends not only on the types of its subexpressions but also on its context. This property prohibits us from locating the source of type errors. What we need is a tree that holds the most general type for each subexpression.

2.3 Most general type tree

To construct a tree that holds the most general type for each subexpression, Chitil [1] uses compositional type inference. Figure 3

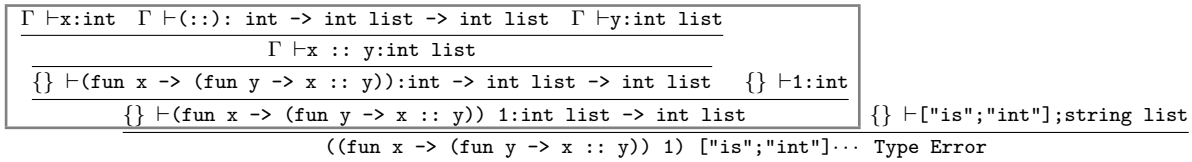


Figure 1. Type inference tree of a compiler (Γ = {x:int; y:int list})

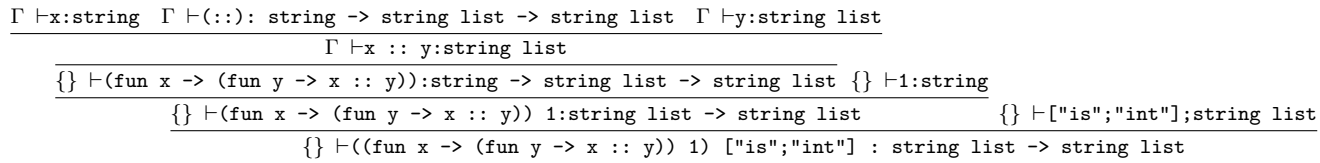


Figure 2. Programmer's intended type inference tree (Γ = {x:string; y:string list})

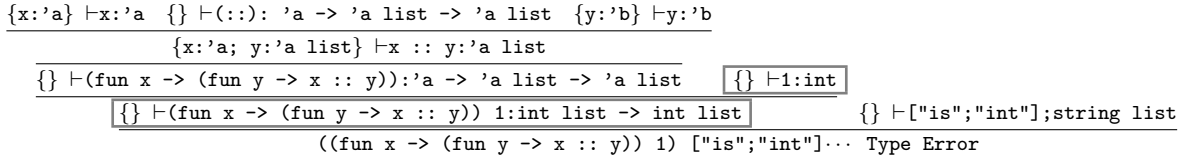


Figure 3. Compositional type inference tree

shows a compositional type inference tree for the example in the previous section.

In the compositional type inference, the type of an expression depends on (or is inferred from) the types of its subexpressions only. For example, like the standard type inference, the type of `(fun x -> fun y -> x :: y) 1` is determined from its two subexpressions. Unlike the standard type inference, the compositional type inference keeps the original types of its subexpressions. In particular, the type of `(fun x -> fun y -> x :: y)` is kept as `'a -> 'a list -> 'a list`. The type variable `'a` is instantiated to `int` *only* to determine the type of the surrounding expression. This way, the compositionality of the subexpression is maintained: even after it is applied to `1`, it keeps its most general type.

With the compositional type inference tree, we can locate the source of the type error. Comparing Figures 2 and 3 from the roots of the trees, we first find the conflict at the expression `(fun x -> fun y -> x :: y) 1` as before. By comparing their children, we find that the type error comes from `1`, because it conflicts with the intended type while the other node `(fun x -> fun y -> x :: y)` does not have a conflict.

The drawback of the compositional type inference tree is that it is not employed in the standard compilers and to use it, *we* have to construct it at all. Chitil employed the compositional type inference tree, partly because it is based on principal typings (rather than principal types) and it is guaranteed that we can infer types of expressions compositionally. However, for the debugging purpose only, we do not actually have to infer types. What we need is a tree that holds the most general type for each subexpression. We call such a tree the *most general type tree*. We will construct such a tree using the compiler's type inferencer based on the decomposition of a program.

2.4 Decomposing a program

Once we realize that what we need is the most general type tree, it is not difficult to construct it using the compiler's type inference. The basic idea is to decompose the erroneous program into subprograms and to infer their types using the compiler's type inference. For example, if a program M is decomposed into subprograms, $M_1, \ldots, M_n$, we construct the following tree, where $\tau, \tau_1, \ldots, \tau_n$ are the types (possibly an error) returned by the compiler's type inferencer by passing $M, M_1, \ldots, M_n$, respectively.

$$\frac{M_1 \quad \ldots \quad M_n}{M} \quad \Rightarrow \quad \frac{M_1 : \tau_1 \quad \ldots \quad M_n : \tau_n}{M : \tau}$$

To cope with bindings properly, we actually maintain a context C of an expression M, treating $C[M]$ as a complete closed program (where $C[M]$ is the expression C whose hole is filled with M possibly capturing free variables of M). We call M in $C[M]$ the *focused* expression.

2.5 Algorithmic debugging

Once the most general type tree is constructed, we use algorithmic debugging to locate the source of type errors, following Chitil [1]. Algorithmic debugging was proposed by Shapiro [13] for locating the source of errors in a Prolog program. The technique is used by many researchers, to locate run-time errors [12], semantic errors [14], etc. Because algorithmic debugging is applicable to any tree structures, it is used widely.

To navigate in a tree, algorithmic debugging receives an oracle and use it to decide which direction in the tree to move. User's input is often used as an oracle.

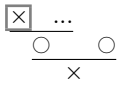
Algorithmic debugging works as follows. Given a tree structure and oracles whether each node contains an error or not, we start from the root of the tree. If the current node contains an error, we examine its child nodes. If all the child nodes are error-free, we conclude that the current node is the source of the error. Otherwise, we pick the erroneous child node and continue recursively.

In our case, we use two kinds of oracles. One is the type error reported by the compiler and the other is whether the type in consideration conflicts with programmer's intention or not. For example, given the tree in Figure 3, we start from the root node. Since

it does not type check, we examine its child nodes. Because both the child nodes type check, we ask the programmer if their types are correct according to his intention. Since the left subexpression is not correct, we proceed to the left subtree. By continuing this process, we identify that `1` is the source of the type error. (When a node has no child nodes, it is reported as the source of the type error.)

2.6 A property of decomposed programs

To correctly locate the source of type errors, we cannot use an arbitrary type trees. We require that once a program is type checked, its subprograms do not contain any type errors. If this property is not satisfied, we could have the following situation:



Suppose that the source of the type error is at the top left $\times$. Even if we use algorithmic debugging to navigate over the tree, we cannot reach the top left $\times$, because it is *protected* by $\bigcirc$. When the middle $\bigcirc$ is encountered, algorithmic debugging judges that there is no error in this subtree.

The property is formally stated as follows:

$$T(C[M]) \Rightarrow \forall C'[M'] \in Dec[\![C[M]]\!], T(C'[M'])$$

Here, T is a predicate stating a given expression is well typed, and Dec decomposes a given program into subprograms. The property says that if a program $C[M]$ is well typed, all of its subprograms are also well typed. Later in the paper, we design various Dec for various languages in a way the property is satisfied.

3. Simply-typed lambda calculus

First, we consider simply-typed lambda calculus. We show the syntax of lambda calculus in Figure 4. It includes constants, variables, abstractions, and applications. We assume that basic primitive operations (such as $+$ that we will use in examples) are predefined as constants. Although the syntax is simple, it is expressive enough to explain the basic behavior of the type debugger. Types include type variables, type constants, and function types.

Let us consider a type inference tree for $\lambda x.x + 1$. Since the only subprogram of $\lambda x.x + 1$ is $x + 1$ and it is further decomposed into three subprograms, x, $+$, and 1, the overall structure of the tree should look like:

$$\dfrac{\dfrac{\Gamma_0 \vdash x \quad \Gamma_0 \vdash (+) \quad \Gamma_0 \vdash 1}{\Gamma_0 \vdash x + 1}}{\Gamma_0 \vdash \lambda x.x + 1}$$

where Γ_0 is the initial environment used by the type inferencer of the underlying compiler and contains all the bindings for the supported constants.

However, the subprograms are not directly typable using the compiler's type inference, because they include free variables (such as x). To make all the subprograms typable, we enclose them with a context that supplies necessary bindings for free variables. The context of this language is defined as either an empty context $\Box$ or a lambda binding $\lambda x.C$ (Figure 4). The most general type tree of $\lambda x.x + 1$ becomes as follows:

$$\dfrac{\dfrac{\begin{array}{l}\Gamma_0 \vdash \lambda x.[x] \quad : \texttt{'a -> ['a]} \\ \Gamma_0 \vdash \lambda x.[(+)] : \texttt{'a -> [int -> int -> int]} \\ \Gamma_0 \vdash \lambda x.[1] \quad : \texttt{'a -> [int]}\end{array}}{\Gamma_0 \vdash \lambda x.[x + 1] : \texttt{int -> [int]}}}{\Gamma_0 \vdash [\lambda x.x + 1] : \texttt{[int -> int]}}$$

Looking at the focused expressions filled in the context, we see that it has the same structure as the previous tree. Thanks to the context, all the subprograms are now typable under Γ_0. The types enclosed by [...] correspond to the types of focused expressions.

Although the most general type tree is similar (modulo notation) to the standard type inference tree for simply-typed lambda calculus:

$$\dfrac{\dfrac{\begin{array}{l}\Gamma_0, x : \texttt{int} \vdash x \quad : \texttt{int} \\ \Gamma_0, x : \texttt{int} \vdash (+) : \texttt{int -> int -> int} \\ \Gamma_0, x : \texttt{int} \vdash 1 \quad : \texttt{int}\end{array}}{\Gamma_0, x : \texttt{int} \vdash x + 1 : \texttt{int}}}{\Gamma_0 \vdash \lambda x.x + 1 : \texttt{int -> int}}$$

they have two important differences. First, the type of x is *not* constrained to `int` at the leaf nodes. Since we treat all the subderivations independently, each judgement depends only on its children. It enables us to locate where the type of x is first forced to `int`. Secondly, the type environment contains only the predefined constants. It enables us to the use compiler's type inferencer to infer the type of each expression. We simply pass it to the compiler's type inferencer and obtain its type. This is in contrast to the standard type inference tree where the environment contains free variables. Since the most type inferencers are designed to accept only an expression to be typed, it is impossible to infer types of open expressions. [2]

The most general type tree is built as follows. A program to be debugged $C[M]$ is first decomposed into subprograms using the decomposition function *Dec* defined in Figure 5. It basically decomposes M and returns a list of its subprograms, but it maintains its context properly so that the resulting subprograms (pairs of a context and a decomposed term) are always closed. When the decomposition of $C[M]$ is $[C_1[M_1]; \ldots; C_n[M_n]]$, all the subprograms become the children of $C[M]$ in the tree.

The type of each subprogram $C[M]$ is determined using the compiler's type inferencer by passing $C[M]$ to it. When the context C is empty $[]$, the returned type is the type of the expression. When the context is not empty, we split the obtained type into two: types for free variables and the type for the focused expression. It is achieved simply by associating free variables to the argument types. For example, if we obtain the type of $\lambda x.[x + 1]$ as int $\rightarrow$ int, we associate the type of x to be int (the argument part of int $\rightarrow$ int) and the type of $x + 1$ to be int (the body part of int $\rightarrow$ int). This is done by the function $Collect$ in Figure 6.

Using Dec and $Collect$, we construct a judgement for $C[M]$ in the tree as shown in Figure 7. First, we construct a closed term M' by plugging M into C. It is then passed to the compiler's type inferencer written as `typing` here. When we obtain a type τ of M', we split it into an environment γ holding types of variables in the context and a type τ' for M. Using them, we can construct a judgement for (possibly open) M (in the context C) as $\Gamma_0, \gamma \vdash M : \tau'$. For $\lambda x.[x+1]$, for example, we have $\Gamma_0, x : \texttt{int} \vdash x+1 : \texttt{int}$.[3]

Once the most general type tree is constructed, we locate the source of type errors using algorithmic debugging as shown in Section 2.5. As an oracle, Chitil's debugger [1] asks if a typing judgement is correct. For example, for the judgement $\Gamma_0, x : \texttt{int} \vdash x + 1 : \texttt{int}$, the following question is asked:

[2] If the compiler's type inferencer was designed to accept an open expression with an environment for its free variables, we could construct a type debugger somewhat easier than the one proposed here. However, it would require deeper understanding of the underlying implementation together with the representation of environments. The method proposed here has an advantage that we can treat the compiler's type inferencer completely as a black box that accepts an expression and returns its type.

[3] In the previous tree, we wrote $\Gamma_0 \vdash \lambda x.[x + 1] : \texttt{int} \rightarrow \texttt{[int]}$ to emphasize that we are using the compiler's type inferencer to infer the type of M in C, but since we are interested in the type of M itself together with the type of its free variables, we also write it using the standard notation $\Gamma_0, x : \texttt{int} \vdash x + 1 : \texttt{int}$.

$$
\begin{array}{rcll}
(M : term) & ::= & c & \text{(constant)} \\
 & | & x & \text{(variable)} \\
 & | & \lambda x.M & \text{(abstraction)} \\
 & | & M_1\ M_2 & \text{(application)} \\
(\tau : typ) & ::= & b & \text{(type variable)} \\
 & | & \text{int, bool, ...} & \text{(type constants)} \\
 & | & \tau_1 \rightarrow \tau_2 & \text{(function type)} \\
(C : context) & ::= & \Box & \text{(empty context)} \\
 & | & \lambda x.C & \text{(lambda context)}
\end{array}
$$

Figure 4. The syntax and types of simply-typed lambda calculus

$$
\begin{array}{rcl}
Dec : context * term & \rightarrow & (context * term)\ list \\
Dec[\![(C, c)]\!] & = & [] \\
Dec[\![(C, x)]\!] & = & [] \\
Dec[\![(C, \lambda x.M)]\!] & = & [(C[\lambda x.\Box], M)] \\
Dec[\![(C, M_1\ M_2)]\!] & = & [(C, M_1); (C, M_2)]
\end{array}
$$

Figure 5. The decomposition function *Dec*

$$
\begin{array}{l}
env = (var * typ)\ list \\
Collect : context \rightarrow typ \rightarrow env \rightarrow (env * typ) \\
\quad Collect_{\Box}[\![\tau]\!]\mu = (\mu, \tau) \\
\quad Collect_{\lambda x.C}[\![\tau_1 \rightarrow \tau_2]\!]\mu = Collect_C[\![\tau_2]\!]\mu[x \rightarrow \tau_1]
\end{array}
$$

Figure 6. The function $Collect$ to obtain types of free variables

$$
\begin{array}{rcl}
Judge[\![(C, M)]\!] & = & \text{let } M' = C[M] \text{ in} \\
 & & \text{let } \tau = \texttt{typing}\ M' \text{ in} \\
 & & \text{let } (\gamma, \tau') = Collect_C[\![\tau]\!] \text{ in} \\
 & & (\gamma, \tau')
\end{array}
$$

Figure 7. The function $Judge$ to obtain typing

`Under the environment where` x `is int,`
`is the type of` $x + 1$ `(an instance of) int?`

As the environment becomes larger, however, the question becomes long and hard to understand especially for novices. To make the question easier to answer, we split the question into two parts [17]: on the environment and on the focused expression. We first ask if x is `int`. If the answer is no, the oracle for this node is immediately no. Only when the answer is yes do we ask if $x + 1$ is `int`. This separation of questions facilitates the reuse of questions which drastically reduces the number of questions [17].

The constructed tree satisfies the property required for the most general type tree. To prove it, we need to show that for each case of the definition of *Dec*, all the subexpressions in the right hand side are well typed if the left hand side is well typed. For constants and variables, it is satisfied vacuously. For abstraction, because the expression in the left hand side $C[\lambda x.M]$ is identical to the expression in the right hand side $C[\lambda x.[M]]$, the property is satisfied. For application, we notice that if $C[M_1 M_2]$ is well typed, $M_1 M_2$ is also well typed in a type environment consistent with C (formally proven by induction on C). Hence, both M_1 and M_2 are well typed in the same environment. Since C has all the necessary bindings for M_1 and M_2 and C simply adds binding to them, both $C[M_1]$ and $C[M_2]$ are well typed as required.

4. Let polymorphism

Next, we consider let polymorphism. We show the syntax in Figure 8. It extends simply-typed lambda calculus with pairs, fixed points, and let expressions. Types are also extended accordingly. Unlike the standard let-polymorphic calculus, we do not introduce type schemes. Type schemes are required only for type inference. Once the type inference is done (in the compiler), all the expressions in the most general type tree are given mono types (possibly containing type variables).

To support let expressions in the type debugger, we first need to define its decomposition. Because a let expression contains two subexpressions, the let-bound expression and the main body, we are tempted to define its decomposition as these two subexpressions. However, straightforward decomposition leads to violation of the required property of Dec (Section 2.6). Consider the following program:

```
let id = fun x -> x in
(id true, id 1)
```

Since `id` in the second subexpression `(id true, id 1)` is free, we need to supply its context. If we naively follow the previous section, however, we end up with the following tree:

$$
\dfrac{\vdash \texttt{[fun x -> x]} \quad : \texttt{'a} \rightarrow \texttt{'a} \qquad \vdash \texttt{(fun id -> [(id true, id 1)])} \cdots \texttt{Type Error}}{\vdash \texttt{[let id = fun x -> x in (true, id 1)]} : \texttt{bool * int}}
$$

Although the expression in the conclusion is well typed, the second subexpression is not well typed. Thus, it does not satisfy the property in Section 2.6.

The reason why the second subexpression is not typable is clear. In the original expression, `id` is used polymorphically, while in the decomposed subexpression, `id` is bound by `fun` and thus monomorphic. From this example, we observe that we need to preserve the *polymorphicness* of let-bound variables, when constructing the most general type tree. For this purpose, we extend the context with a let context (Figure 8). We also extend it with a fix context since it is a (monomorphic) binder. Using the let context, the above tree becomes as follows, satisfying the required property:

$$
\dfrac{\vdash \texttt{let id = fun x -> x in [(id true, id 1)]} : \texttt{bool * int}}{\vdash \texttt{[let id = fun x -> x in (true, id 1)]} : \texttt{bool * int}}
$$

Note that we do not treat `fun x -> x` as the subexpression of the let expression, but as a part of the let context. It leads to better debugging interaction (see below).

The introduction of let contexts has an interesting impact on the interactive debugging. Consider the following example:

```
let fst (a, b) = a in
let snd (a, b) = b in
let lst1 = [("Ann", 18); ("Bob", 23)] in
let lst2 = [(true, "Tom");(false, "John")] in

let rec split lst = match lst with
 | [] -> ([], [])
 | (a, b) :: rest ->
     let (alst, blst) = split rest in
     (a :: alst, a :: blst) in

(fst (split lst1)) @ (snd (split ls2))
```

In this example, the source of type error is the boxed part in the function `split`. The programmer considers `split` receives a list of pairs of type `('a * 'b) list` and returns a pair of two split lists of type `'a list` and `'b list`. Because he has written `a` instead of `b` in the boxed part, however, the compiler infers that the type of

$$
\begin{array}{rcll}
(M : term) & ::= & c \mid x \mid \lambda x.M \mid M_1\ M_2 \mid (M_1, ..., M_n) & \text{(pair)} \\
& \mid & fix\ f\ v_1\ ...\ v_n \rightarrow M & \text{(fixed point)} \\
& \mid & let\ v\ = M_1\ in\ M_2 & \text{(let-expression)} \\
(\tau : typ) & ::= & b \mid \text{int} \mid \text{bool} \mid ... \mid \tau_1 \rightarrow \tau_2 \mid \tau_1\ *\ ...\ *\ \tau_n & \text{(product type)} \\
(C : context) & ::= & \Box \mid \lambda x.C \mid fix\ f\ v_1 ... v_n \rightarrow C & \text{(fix context)} \\
& \mid & let\ x = M\ in\ C & \text{(let context)}
\end{array}
$$

Figure 8. The syntax and types for the let-polymorphic language

$$
\begin{array}{rcl}
Dec : context * term & \rightarrow & (context * term)\ list \\
Dec[\![(C, c)]\!] & = & [] \\
Dec[\![(C, x)]\!] & = & Get[\![(C, x, \Box, \text{None})]\!] \\
Dec[\![(C, \lambda x.M)]\!] & = & [(C[\lambda x.\Box], M)] \\
Dec[\![(C, M_1\ M_2)]\!] & = & [(C, M_1); (C, M_2)] \\
Dec[\![(C, (M_1, ..., M_n))]\!] & = & [(C, M_1); ...; (C, M_n)] \\
Dec[\![(C, fix\ f\ v_1 ... v_n \rightarrow M)]\!] & = & [(C[fix\ f\ v_1 ... v_n \rightarrow \Box], M)] \\
Dec[\![(C, let\ x = M_1\ in\ M_2)]\!] & = & [(C[let\ x = M_1\ in\ \Box], M_2)]
\end{array}
$$

Figure 9. *Dec* for the let-polymorphic language

$$
\begin{array}{rcll}
Get : context * var * context * (context * term)\ option & \rightarrow & (context * term)\ list & \\
Get[\![(\Box, v, C, p)]\!] & = & [] & \text{if } p = \text{None} \\
 & & [(C', M)] & \text{if } p = \text{Some}(C', M) \\
Get[\![(\lambda x.C', v, C, p)]\!] & = & Get[\![(C', v, C[\lambda x.\Box], \text{None})]\!] & \text{if } x = v \\
 & & Get[\![(C', v, C[\lambda x.\Box], p)]\!] & \text{if } x \neq v \\
Get[\![(fix\ f v_1 ... v_n \rightarrow C', v, C, p)]\!] & = & Get[\![(C', v, C[fix\ f\ v_1 ... v_n \rightarrow \Box], \text{None})]\!] & \text{if } v \in \{f, v_1 ... v_n\} \\
 & & Get[\![(C', v, C[fix\ f\ v_1 ... v_n \rightarrow \Box], p)]\!] & \text{if } v \notin \{f, v_1 ... v_n\} \\
Get[\![(let\ x = M\ in\ C', v, C, p)]\!] & = & Get[\![(C', v, C[let\ x = M\ in\ \Box], \text{Some}(C, M))]\!] & \text{if } x = v \\
 & & Get[\![(C', v, C[let\ x = M\ in\ \Box], p)]\!] & \text{if } x \neq v
\end{array}
$$

Figure 10. The function *Get* to search for definition of variables for the let-polymorphic language

$$
\begin{array}{rcl}
env = (var * typ)\ list & & \\
Collect : context \rightarrow typ \rightarrow env & \rightarrow & (env * typ) \\
Collect_{\Box}[\![\tau]\!]\mu & = & (\mu, \tau) \\
Collect_{\lambda x.C}[\![\tau_1 \rightarrow \tau_2]\!]\mu & = & Collect_C[\![\tau_2]\!]\mu[x \rightarrow \tau_1] \\
Collect_{fix\ f\ v_1 ... v_n \rightarrow C}[\![\tau_1 \rightarrow ... \rightarrow (\tau_n \rightarrow \tau)]\!]\mu & = & Collect_C[\![\tau]\!]\mu[f \rightarrow (\tau_1 \rightarrow ... \rightarrow (\tau_n \rightarrow \tau)); v_1 \rightarrow \tau_1; ...; v_n \rightarrow \tau_n] \\
Collect_{let\ x=M\ in\ C}[\![\tau]\!]\mu & = & Collect_C[\![\tau]\!]\mu
\end{array}
$$

Figure 11. *Collect* for the let-polymorphic language

`split` is `('a * 'b) list -> 'a list * 'a list`. This leads to a type error at the last line where the type of `(snd (split lst2))` becomes `bool list` rather than `string list`.

The root part of the most general type tree for this example becomes as follows:

$$
\dfrac{\begin{array}{l}\vdash C\texttt{[(fst (split lst1))]:string list} \\ \vdash C\texttt{[(snd (split lst2))]:bool list}\end{array}}{\vdash C\texttt{[(fst (split lst1)) @ (snd (split lst2))]} \cdots \texttt{Type Error}}
$$

where the context C contains bindings for `fst`, `snd`, `lst1`, `lst2`, and `split`. Because the bindings for these variables are kept in let expressions, `split` can be used polymorphically at different types for `lst1` and `lst2`.

From this tree, the debugging proceeds as follows. Since the type `bool list` of `(snd (split lst2))` is in conflict with the intended type `string list`, we expand the node for `(snd (split lst2))`, which gives us the following subtree:

$$
\dfrac{\begin{array}{l}\vdash C\texttt{[snd]} \quad \texttt{:'a * 'b -> 'b} \\ \vdash C\texttt{[(split lst2)]:bool list * bool list}\end{array}}{\vdash C\texttt{[(snd (split lst2))]:bool list}}
$$

We further expand the conflicting node `split lst2` to obtain:

$$
\dfrac{\begin{array}{l}\vdash C\texttt{[split]:('a * 'b) list -> 'a list * 'a list} \\ \vdash C\texttt{[lst2] :(bool * string) list}\end{array}}{\vdash C\texttt{[(split lst2)]:bool list * bool list}}
$$

At this point, we find that the type of `split` conflicts with the programmer's intention. Since a variable `split` cannot be further decomposed, we could have reported that this variable was the source of the type error. However, this is not very useful. We rather want to inspect why the function `split` has the above unintended type.

To enable inspecting the definition of let-bound variables, we change the decomposition function as shown in Figure 9. The definition is the straightforward extension of the previous definition except for the variable case. When we decompose a variable, we search for its definition using Get defined in Figure 10. When the variable is bound by a let expression, Get returns its definition as the decomposition of the variable. Otherwise, the variable is bound by lambda or fix, so Get returns no decomposition. Using this decomposition function, we can further debug into the definition of `split` to identify the ultimate source of the type error (i.e. the boxed `a`).

Since the context is extended with a let context and a fix context, the definition of $Collect$ is extended accordingly as shown in Figure 11. It collects types for lambda- and fix-bound variables

and discards let-bound variables since they do not appear in the type returned by the compiler. (We assume that the compiler's type inferencer returns $\tau_1 \rightarrow ... \rightarrow \tau_n \rightarrow \tau$ as the type of $fix\ f\ v_1...v_n \rightarrow M$ (and hence of `f`) where τ_1 is the type of v_1, etc., and τ is the type of M.)

As the program to be debugged becomes larger, the number of let-bound variables increases. Since we can debug into the definition of let-bound variables when their types conflict with the programmer's intention, we do not ask the programmer for the type of let-bound variables as an oracle each time. Rather, we only ask for variables in a context that are bound by lambda or fix. This is in consistent with Chitil's approach who maintains an environment for polymorphic variables separately.

We can confirm that the required property for the function *Dec* is still satisfied. The interesting case is for variables. (Other cases are similar to the proof sketch shown for simply-typed lambda calculus.) Assume that $C[x]$ is well typed by the invariant. We first observe that $Get[(C_1, x, C_2, p)]$ maintains an invariant that $C_2[C_1]$ is always the same across the recursive call, because at each recursive call, the topmost frame of C_1 is simply moved to the hole of C_2. This assures us that all the contexts appearing in the definition of Get is well typed (as a context), because the initial context $[C[x]]$ with which Get is called from Dec is well typed. Next, the returned expression $C'[M']$ is collected only from the let case. Because $C[let\ x = M'\ in\ C']$ is well typed, we hence have that $C[M']$ is also well typed as required.

Observe how the required property serves as a guideline what we have to do and what we can do to incorporate let expressions. We have to define the decomposition function so that the let polymorphism is preserved. On the other hand, as long as the property is satisfied, we have the liberty of defining the decomposition in a way the debugging process becomes easier for programmers to understand. By defining the decomposition of let-bound variables as their definition, we achieve jump from the use of variables to their definition.

5. Weak polymorphism

In this section, we introduce references to see the interaction of weak polymorphism in our type debugging. We show the syntax in Figure 12. It includes references, dereferences, and assignments to a reference. Types are extended with a reference type.

Let us consider a typical example where the weak polymorphism arises:

```
let id_ref = ref (fun x -> x) in
(!id_ref 0, !id_ref true)
```

Since the identity function `(fun x -> x)` is put into a cell, `id_ref` is given a weak polymorphic type `('_a -> '_a) ref`. The weak type variable `'_a` can be instantiated only once. Since it becomes `int` at the first application, a type error occurs at the second application where `'_a` needs to become `bool`.

It is not difficult to support references in our type debugger. We simply need to extend *Dec* to handle new constructs (Figure 13). We could then identify the source of the above type error as the second line, because whole the expression is not typable but the two subexpressions together with their context, namely

```
let id_ref = ref (fun x -> x) in [!id_ref 0]
```

and

```
let id_ref = ref (fun x -> x) in [!id_ref true]
```

are both typable. The most general type tree becomes as follows, where C contains a binding for `id_ref`:

$$\frac{\vdash C\texttt{[!id\_ref 0]:int} \quad \vdash C\texttt{[!id\_ref true]:bool}}{\vdash C\texttt{[(!id\_ref 0, !id\_ref true)]} \cdots \texttt{Type Error}}$$

However, the above behavior is sometimes not very informative. Consider the following example:

```
let pair x y = fun f -> f x y in
let fst x y = x in
let snd x y = y in
let p = pair 1 true in
(p snd, p fst)
```

In this program, a pair is Church-encoded using a function. Then, a pair `p` of `1` and `true` is constructed, and its swapped tuple is returned. Because `p` is bound to a non-value, however, it has a weak type `(int -> bool -> '_a) -> '_a`. When `p` is applied to `snd` of type `'a -> 'b -> 'b`, the weak type variable `'_a` is instantiated to `bool`, and a type error occurs when `p` is applied to `fst` of type `'a -> 'b -> 'a`, where `'_a` needs to be instantiated to `int`.

For this program, our type debugger again reports that the expression `(p snd, p fst)` is the source of the type error, because both `p snd` and `p fst` are typable in the current context C (containing four let bindings):

$$\frac{\vdash C\texttt{[p snd]:bool} \quad \vdash C\texttt{[p fst]:int}}{\vdash C\texttt{[(p snd, p fst)]} \cdots \texttt{Type Error}}$$

if both the types are consistent with programmer's intention.

However, if the programmer intends that `p` be fully polymorphic, he would be puzzled why the conclusion is not typed as `bool * int`. In fact, although the type of `p` is constrained to `(int -> bool -> bool) -> bool` at `p snd`, that information is discarded in the most general type tree and a fresh `p` is used to infer the type of `p fst`. Remember that all the types are inferred by passing each expression to the compiler's type inferencer independently. Also, note that our type debugger asks the programmer only for the type of focused expression and the types of its free variables that are *not* bound by `let`. Since `p` is bound by `let` in this case, the type debugger asks only the types of `p snd` and `p fst` (both of which have intended types). Thus, the programmer has no opportunity to say that the type of `p` is too restrictive.

To handle weak polymorphism more properly, we examine the type of weak variables and ask if their instantiation is in conflict with the programmer's intention. In the above case, we construct the following tree:

$$\frac{\vdash C\texttt{[(p, p snd)]:}\tau_1\texttt{ * bool} \quad \vdash C\texttt{[(p, p fst)]:}\tau_2\texttt{ * int}}{\vdash C\texttt{[(p, (p snd, p fst))]} \cdots \texttt{Type Error}}$$

$$\begin{array}{ll} where & \tau_1 = \texttt{(int -> bool -> bool) -> bool} \\ & \tau_2 = \texttt{(int -> bool -> int) -> int} \end{array}$$

Since the definition of `p` is expansive, we pair it with the focused expression and obtain its type from the compiler. We then ask the programmer if the type of `p` is as intended. In our case, since τ_1 (and τ_2) is not polymorphic enough, the programmer can reply no, and the debugger will move to the definition of `p` to find why it is not polymorphic.

To enable this behavior, $ExpansiveVar$ in Figure 14 collects a list of expansive variables, $AttachVar$ in Figure 15 pairs them with the focused expression, and $CollectVar$ in Figure 16 extracts the types of expansive variables. When collecting expansive variables, care must be taken for variables with the same name. For example, in the following context,

`fun x -> let x =` $expansive\ expression$ `in` $\square$

$$
\begin{array}{rcll}
(M : term) & ::= & c \mid x \mid \lambda x.M \mid M_1\ M_2 \mid (M_1, ..., M_n) \mid ref\ M & \text{(reference)} \\
& \mid & !M & \text{(dereference)} \\
& \mid & v := M & \text{(assignment)} \\
& \mid & fix\ f\ v_1\ ...\ v_n \rightarrow M \mid let\ v\ = M_1\ in\ M_2 & \\
(\tau : typ) & ::= & b \mid \text{int} \mid \text{bool} \mid ... \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * ... * \tau_n \mid \tau\ ref & \text{(reference type)} \\
(C : context) & ::= & \Box \mid \lambda x.C \mid fix\ f\ v_1 ... v_n \rightarrow C \mid let\ x = M\ in\ C &
\end{array}
$$

Figure 12. The syntax and types for the language with references

$$
\begin{array}{rcl}
Dec : context * term & \rightarrow & (context * term)\ list \\
Dec[\![(C, ref\ M)]\!] & = & [(C, M)] \\
Dec[\![(C, !M)]\!] & = & [(C, M)] \\
Dec[\![(C, v := M)]\!] & = & [(C, M)]
\end{array}
$$

Figure 13. *Dec* for the language with references (for new constructs only)

$$
\begin{array}{rcl}
ExpansiveVar : context * var\ list & \rightarrow & var\ list \\
ExpansiveVar[\![(\Box, vs)]\!] & = & vs \\
ExpansiveVar[\![(\lambda x.C, vs)]\!] & = & ExpansiveVar[\![(C, vs \setminus \{x\})]\!] \\
ExpansiveVar[\![(fix\ f\ v_1 ... v_n \rightarrow C, vs)]\!] & = & ExpansiveVar[\![(C, vs \setminus \{f, v_1, ..., v_n\})]\!] \\
ExpansiveVar[\![(let\ x = M'\ in\ C, vs)]\!] & = & \text{if } (\texttt{is\_expansive}\ M') \text{ then } ExpansiveVar[\![(C, (x :: vs))]\!] \\
 & & \text{else } ExpansiveVar[\![(C, vs)]\!]
\end{array}
$$

Figure 14. The function $ExpansiveVar$ to collect expansive variables

$$
\begin{array}{rcl}
AttachVar : var\ list * term & \rightarrow & term \\
AttachVar[\![([], M)]\!] & = & M \\
AttachVar[\![(v :: vs, M)]\!] & = & (v, AttachVar[\![(vs, M)]\!])
\end{array}
$$

Figure 15. $AttachVar$ to pair expansive variables with a focused expression

$$
\begin{array}{rcl}
env = (var * typ)\ list & & \\
CollectVar : var\ list \rightarrow typ & \rightarrow & env * typ \\
CollectVar_{[]}[\![\tau]\!] & = & ([], \tau) \\
CollectVar_{v :: vs}[\![\tau_1 * \tau_2]\!] & = & \text{let } (\mu, \tau) = CollectVar_{vs}[\![\tau_2]\!] \text{ in } (\mu[v \rightarrow \tau_1], \tau)
\end{array}
$$

Figure 16. The function $CollectVar$ to obtain types of expansive variables

$$
\begin{array}{rcl}
Judge[\![(C, M)]\!] & = & \text{let } vs = ExpansiveVar[\![(C, [])]\!] \text{ in} \\
 & & \text{let } M' = C[AttachVar[\![(vs, M)]\!]] \text{ in} \\
 & & \text{let } \tau = \texttt{typing}\ M' \text{ in} \\
 & & \text{let } (\gamma, \tau') = Collect_C[\![\tau]\!] \text{ in} \\
 & & \text{let } (\gamma', \tau'') = CollectVar_{vs}[\![\tau']\!] \text{ in} \\
 & & (\gamma @ \gamma', \tau'')
\end{array}
$$

Figure 17. $Judge$ for the language with references

`x` has to be treated as expansive (because `x` in $\Box$ refers to the inner `x`), but not when `fun x ->` appears inside the let expression.

We can easily confirm that the required property holds for this language, because the decomposition function for the new constructs takes simply the subexpression of the original expression and the pairing of expansive variables does not affect the typability of expressions. By modifying the expression to be typed without violating the property, we can design a type debugger that shows more useful information for the programmer.

6. Object

So far, we have seen that interactive debugging is possible for various language constructs by suitably defining *Dec* function that satisfies the required property. In fact, this idea extends to seemingly complex language constructs. In this section, we introduce objects into the language and see how it can be supported using the similar idea. See Figure 18 for the syntax. It models OCaml-style objects where an object is defined using a class (in which single inheritance is allowed) and is created by the *new* construct. Besides the inheritance declaration, an object can contain method and value declarations. Although the class name (to be more precise, the object structure denoted by the class name) is used as a type in OCaml, we introduce only a special type *obj* in our type debugger. It is a type of all the objects.

The decomposition function *Dec* is extended with the new constructs and the *Get* function used in the variable case is extended with the class context. The interesting cases are for *new* and method invocation of *Dec*. In both the cases, we need to identify the object mentioned in the expressions (in case their types contradict with intended types, so that we can debug into the object). For this purpose, the function *SearchObj* in Figure 21 is used. Its behavior is similar to that of *Get*, but differs in that *SearchObj* collects *all* the method declarations in the designated object. In particular, if the object contains inheritance declaration, those method declarations are collected, too (see *SearchObj'*).

We collect all the declarations in an object because types of declared methods in an object are mutually dependent. Thus, we

need to ask the types of all these method declarations to locate the source of type errors. For example, consider the following program:

```
class counter =
object (self)
  val mutable n = 0
  method incr = n <- n+1
  method get = n
end

let t = (new counter) in
t#incr;
("now, counter is " ^ t#get)
```

The last line results in a type error, because `t#get` returns an integer, which is in conflict with the intended type (i.e., `string`). To find the source of this type error, we first look up `t`'s class definition `counter` and search for the definition of the `get` method. However, we find here that the `get` method itself does not force the type of `n` as an integer. It simply returns a value of `n`. Instead, `n` is an integer because it is assigned `0` and `n+1` elsewhere in the class.[4] Thus, we need to examine all the declarations in an object to find the source of type errors.

Since, any method declarations can be the source of type errors, we collect all the method declarations in a class definition, and return them as the decomposition of the object reference. Although this strategy is necessary in general, it could lead to too many number of questions. Its practical impact is our future work.

We can confirm that *Dec* satisfies the required property as follows. First, *Get* will return a list of well-typed subexpressions only, using the similar argument we described in Section 4. For *new* and method invocation, we have to show that *SearchObj* returns a list of well-typed subexpressions. It can be proved by observing that *SearchObj* simply collects subexpressions in an object in a suitable context. The only interesting case is for a class declaration, where we have to properly insert bindings for the arguremnts to the class and the self variable v'. Note that declared values are put into let contexts in *SearchObj*$'$.

7. Module

Similarly to objects, we can introduce modules, too. Figure 22 shows the syntax. We introduce accesses to a value in a module, *open*, and module declarations. A module declaration contains variable and type declarations.

The decomposition function *Dec* is extended to cope with new constructs straightforwardly (Figure 23). *SearchMod* (in Figure 25) is defined similarly to *SearchObj* in the previous section. Since the declarations in a module is ordered (in contrast to method declarations in objects which are mutually recursive), we do not collect all the declarations but maintain the order of declarations in a context and returns a designated definition. The treatment of *open* in *Get* (in Figure 24) is interesting. When we search for the definition of a variable v and the current context is *open X*, we search for the definition of v in the module *X*. It enables us to search for the definition of variable v that is defined in the module *X*, when the type error was located at the variable v.

8. Implementation

We have implemented the type debugger for OCaml version 3.12.1. To minimize the implementation efforts and to avoid going too much detail into the implementation of OCaml itself, we utilize the following components from OCaml as is:

- the abstract syntax tree for structures, expressions, and types (together with the lexer, the parser, and the pretty printer)
- the type inferencer `typing` (that accepts an expression and returns its type, both expressed using the above abstract syntax tree)
- the `is_expansive` function (that accepts an expression and returns a boolean to judge whether the given expression needs to be kept monomorphic or not)

By using exactly the same abstract syntax as OCaml, we can not only avoid reproducing the same abstract syntax but also utilize OCaml's own lexer, parser, and pretty printer. The abstract syntax of OCaml is defined beautifully, and we had no problems using it. In addition to the type inferencer, we utilize the `is_expansive` function. Although OCaml has its own criteria for weak polymorphism [2], we can stay away from it by using OCaml's `is_expansive` function as is. Furthermore, this approach is robust to updates of OCaml: if the syntax and the interface of the two functions are the same, we can use the same debugger.

A slight complication is that OCaml treats a let expression without `in` differently from the one with `in`: the former is a structure, while the latter is an expression. We supported both the styles by splitting the context into two: the structure part and the expression part.

Another complication is the use of patterns in place of a variable declaration. For example, instead of `fun lst ->`, one can write `fun (first :: rest) ->`. In such a case, we have to add the information that `first` and `rest` have types `'a` and `'a list`, respectively, into the most general type tree.

The rest of the language constructs are supported without requiring any special treatment. For each new construct, we define its decomposition and show that it satisfies the required property.

9. Related work

The typical approach to improving type error messages is to design a new type inference. Wand [18] proposes to keep track of the history why type variables are instantiated and shows the conflicting histories when type error arises. Lee and Yi [6] present the algorithm M that finds conflict of types earlier than the algorithm W and thus reports a narrower expression as an error. Heeren and Hage [5] use a constraint-based type inference for improving type error messages. Although these improved type error messages are useful for programmers, it is in general not possible to identify the source of type errors by a single error message.

To locate the source of type errors, Chitil [1] uses compositional type inference and constructs an interactive type debugger for a subset of Haskell. Based on his work, we designed a type debugger for OCaml using the compiler's own type inferencer rather than a tailor-made type inferencer. The use of the compiler's type inferencer enables us to build a type debugger for a larger language easily. Stuckey, Sulzmann, and Wazny [16] find the source of type errors using type inference via CHR solving. They implement a type debugger called Chameleon, which can explain why an inferenced type is derived by searching. Tailor-made type inference is used for this purpose.

As different approaches, Haack and Wells [3] use slicing with respect to types to narrow the erroneous possible parts of programs. By extracting the slice related to type errors, they help the programmer to identify the source of type errors. The advantage of this approach is that the process is automatic and the programmer does not have to answer questions. Lerner et al. [7] propose automatic type-error correction. They replace the erroneous part with various syntactically correct similar expressions, and see if they type check. If they do, they are displayed as the candidates for fixing the

[4] In OCaml, `+` is used exclusively for integer addition.

$(L : classobj)$	::=	$inherit\ x\ M_1 ... M_n$	(inheritance declaration)
	|	$method\ x = M$	(method declaration)
	|	$val\ x = M$	(value declaration)
$(M : term)$	::=	$c \mid x \mid \lambda x.M \mid M_1\ M_2 \mid (M_1, ..., M_n) \mid x_1 \# x_2$	(method invocation)
	|	$fix\ f\ v_1\ ...\ v_n \rightarrow M \mid let\ v = M_1\ in\ M_2 \mid new\ x$	(object creation)
	|	$class\ x\ v_1 ... v_n = object\ (v')\ L_1 ... L_n\ end\ in\ M$	(class definition)
$(\tau : typ)$	::=	$b \mid$ int $\mid$ bool $\mid$ ... $\mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * ... * \tau_n \mid obj$	(object type)
$(C : context)$	::=	$\Box \mid \lambda x.C \mid fix\ f\ v_1 ... v_n \rightarrow C \mid let\ x = M\ in\ C$	
	|	$class\ x\ v_1 ... v_n = object\ (v')\ L_1 ... L_n\ end\ in\ C$	(class context)

Figure 18. The syntax and types for the object language

$$
\begin{array}{rcl}
Dec : context * term & \rightarrow & (context * term)\ list \\
Dec[\![(C, c)]\!] & = & [] \\
Dec[\![(C, x)]\!] & = & Get[\![(C, x, \Box, \text{None})]\!] \\
Dec[\![(C, \lambda x.M)]\!] & = & [(C[\lambda x.\Box], M)] \\
Dec[\![(C, M_1\ M_2)]\!] & = & [(C, M_1); (C, M_2)] \\
Dec[\![(C, (M_1, ..., M_n))]\!] & = & [(C, M_1); ...; (C, M_n)] \\
Dec[\![(C, x_1 \# x_2)]\!] & = & SearchObj[\![(C, x_1, \Box, \text{None})]\!] \\
Dec[\![(C, fix\ f\ v_1 ... v_n \rightarrow M)]\!] & = & [(C[fix\ f\ v_1 ... v_n \rightarrow \Box], M)] \\
Dec[\![(C, let\ x = M_1\ in\ M_2)]\!] & = & [(C[let\ x = M_1\ in\ \Box], M_2)] \\
Dec[\![(C, new\ x)]\!] & = & SearchObj[\![(C, x, \Box, \text{None})]\!] \\
Dec[\![(C, class\ x\ v_1 ... v_n = object\ (v')\ L_1 ... L_n\ end\ in\ M)]\!] & = & [(C[class\ x\ v_1 ... v_n = object\ (v')\ L_1 ... L_n\ end\ in\ \Box], M)]
\end{array}
$$

Figure 19. *Dec* for the object language

$$
\begin{array}{rcl}
Get : context * var * context * (context * term)\ option & \rightarrow & (context * term)\ list \\
Get[\![(\Box, v, C, p)]\!] & = & [] \qquad \text{if } p = \text{None} \\
 & & [(C', M')] \qquad \text{if } p = \text{Some}(C', M') \\
Get[\![(\lambda x.C', v, C, p)]\!] & = & Get[\![(C', v, C[\lambda x.\Box], \text{None})]\!] \qquad \text{if } x = v \\
 & & Get[\![(C', v, C[\lambda x.\Box], p)]\!] \qquad \text{if } x \neq v \\
Get[\![(fix\ f v_1 ... v_n \rightarrow C', v, C, p)]\!] & = & Get[\![(C', v, C[fix\ f\ v_1 ... v_n \rightarrow \Box], \text{None})]\!] \qquad \text{if } v \in \{f, v_1 ... v_n\} \\
 & & Get[\![(C', v, C[fix\ f\ v_1 ... v_n \rightarrow \Box], p)]\!] \qquad \text{if } v \notin \{f, v_1 ... v_n\} \\
Get[\![(let\ x = M\ in\ C', v, C, p)]\!] & = & Get[\![(C', v, C[let\ x = M\ in\ \Box], \text{Some}(C, M))]\!] \qquad \text{if } x = v \\
 & & Get[\![(C', v, C[let\ x = M\ in\ \Box], p)]\!] \qquad \text{if } x \neq v \\
Get[\![(class\ x\ v_1 ... v_n = object\ (v') & = & Get[\![(C', v, C[class\ x\ v_1 ... v_n = object\ (v') \\
L_1 ... L_n\ end\ in\ C', v, C, p)]\!] & & \qquad L_1 ... L_n\ end\ in\ \Box], p)]\!]
\end{array}
$$

Figure 20. *Get* for the object language

$$
\begin{array}{rcl}
SearchObj' : classobj\ list * context & \rightarrow & (context\ *\ term)\ list \\
SearchObj'[\![([], C)]\!] & = & [] \\
SearchObj'[\![((inherit\ x\ M_1 ... M_n) ::\ r, C)]\!] & = & SearchObj[\![(C, x, \Box, \text{None})]\!]@[(C, M_1); ...; (C, M_n)]@SearchObj'[\![(r, C)]\!] \\
SearchObj'[\![((method\ x = M) ::\ r, C)]\!] & = & (C, M) :: SearchObj'[\![(r, C)]\!] \\
SearchObj'[\![((val\ x = M) ::\ r, C)]\!] & = & SearchObj'[\![(r, C[let\ x = M\ in\ \Box])]\!]
\end{array}
$$

$$
\begin{array}{rcl}
SearchObj : context * var * context * & & \\
((context * term)\ list)\ option & \rightarrow & (context\ *\ term)\ list \\
SearchObj[\![(\Box, v, C, p)]\!] & = & [] \qquad \text{if } p = \text{None} \\
 & & P \qquad \text{if } p = \text{Some } P \\
SearchObj[\![(\lambda x.C', v, C, p)]\!] & = & SearchObj[\![(C', v, C[\lambda x.\Box], p)]\!] \\
SearchObj[\![(fix\ f\ v_1 ... v_n \rightarrow C', v, C, p)]\!] & = & SearchObj[\![(C', v, C[fix\ f\ v_1 ... v_n \rightarrow \Box], p]\!] \\
SearchObj[\![(let\ x = M\ in\ C', v, C, p)]\!] & = & SearchObj[\![(C', v, C[let\ x = M\ in\ \Box], p)]\!] \\
SearchObj[\![(class\ x\ v_1 ... v_n = object\ (v') & = & \text{if } x = v \text{ then} \\
L_1 ... L_n\ end\ in\ C', v, C, p)]\!] & & \quad SearchObj[\![(C', v, C[class\ x\ v_1 ... v_n = object(v')\ L_1 ... L_n\ end\ in\ \Box], \\
 & & \qquad \text{Some}(SearchObj'[\![(L_1 ... L_n, C[\lambda v_1 .... \lambda v_n . \lambda v' . \Box])]\!]))]\!] \\
 & & \text{else } SearchObj[\![(C', v, C[class\ x\ v_1 ... v_n = object(v')\ L_1 ... L_n\ end\ in\ \Box], p)]\!]
\end{array}
$$

Figure 21. *SearchObj* to search for the definition of objects

$$
\begin{array}{rcll}
(M : term) & ::= & c \mid x \mid \lambda x.M \mid M_1\ M_2 \mid (M_1, ..., M_n) \mid X.x & \text{(module access)} \\
& \mid & fix\ f\ v_1\ ...\ v_n \rightarrow M \mid let\ v\ = M_1\ in\ M_2 \mid open\ X\ in\ M & \text{(open)} \\
& \mid & type\ x = \tau\ in\ M & \text{(type definition)} \\
& \mid & module\ X = struct\ D_1 ... D_n\ end\ in\ M & \text{(module definition)} \\
(D : definition) & ::= & let\ x = M & \text{(value declaration)} \\
& \mid & type\ x = \tau & \text{(type declaration)} \\
(\tau : typ) & ::= & b \mid \text{int} \mid \text{bool} \mid ... \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * ... * \tau_n & \\
(C : context) & ::= & \square \mid \lambda x.C \mid fix\ f\ v_1 ... v_n \rightarrow C \mid let\ x = M\ in\ C & \\
& \mid & open\ X\ in\ M & \text{(open context)} \\
& \mid & type\ x = \tau\ in\ C & \text{(type context)} \\
& \mid & module\ X = struct\ D_1 ... D_n\ end\ in\ C & \text{(module context)}
\end{array}
$$

Figure 22. The syntax and types for the module language

$$
\begin{array}{rcl}
Dec : context * term & \rightarrow & (context * term)\ list \\
Dec[\![(C, c)]\!] & = & [] \\
Dec[\![(C, x)]\!] & = & Get[\![(C, x, \square, \text{None})]\!] \\
Dec[\![(C, \lambda x.M)]\!] & = & [(C[\lambda x.\square], M)] \\
Dec[\![(C, M_1\ M_2)]\!] & = & [(C, M_1); (C, M_2)] \\
Dec[\![(C, (M_1, ..., M_n))]\!] & = & [(C, M_1); ...; (C, M_n)] \\
Dec[\![(C, X.x)]\!] & = & SearchMod[\![(C, X, x, \square, \text{None})]\!] \\
Dec[\![(C, fix\ f\ v_1 ... v_n \rightarrow M)]\!] & = & [(C[fix\ f\ v_1 ... v_n \rightarrow \square], M)] \\
Dec[\![(C, let\ x = M_1\ in\ M_2)]\!] & = & [(C[let\ x = M_1\ in\ \square], M_2)] \\
Dec[\![(C, open\ X\ in\ M)]\!] & = & [(C[open\ X\ in\ \square], M)] \\
Dec[\![(C, type\ x = \tau\ in\ M)]\!] & = & [(C[type\ x = \tau\ in\ \square], M)] \\
Dec[\![(C, module\ X = struct\ D_1 ... D_n\ end\ in\ M)]\!] & = & [(C[module\ X = struct\ D_1 ... D_n\ end\ in\ \square], M)]
\end{array}
$$

Figure 23. *Dec* for the module language

$$
\begin{array}{rcl}
Get : context * var * context * (context * term)\ option & \rightarrow & (context * term)\ list \\
Get[\![(\square, v, C, p)]\!] & = & [] \quad \text{if } p = \text{None} \\
 & & [(C', M')] \quad \text{if } p = \text{Some}(C', M') \\
Get[\![(\lambda x.C', v, C, p)]\!] & = & Get[\![(C', v, C[\lambda x.\square], \text{None})]\!] \quad \text{if } x = v \\
 & & Get[\![(C', v, C[\lambda x.\square], p)]\!] \quad \text{if } x \neq v \\
Get[\![(fix\ f v_1 ... v_n \rightarrow C', v, C, p)]\!] & = & Get[\![(C', v, C[fix\ f\ v_1 ... v_n \rightarrow \square], \text{None})]\!] \quad \text{if } x \in \{f, v_1 ... v_n\} \\
 & & Get[\![(C', v, C[fix\ f\ v_1 ... v_n \rightarrow \square], p)]\!] \quad \text{if } x \notin \{f, v_1 ... v_n\} \\
Get[\![(let\ x = M\ in\ C', v, C, p)]\!] & = & Get[\![(C', v, C[let\ x = M\ in\ \square], \text{Some}(C, M))]\!] \quad \text{if } x = v \\
 & & Get[\![(C', v, C[let\ x = M\ in\ \square], p)]\!] \quad \text{if } x \neq v \\
Get[\![(open\ X\ in\ C', v, C, p)]\!] & = & \text{let } t = SearchMod[\![(C, X, v, \square, \text{None})]\!] \text{ in} \\
 & & \text{if } t = \text{None then } Get[\![(C', v, C[open\ X\ in\ \square], p)]\!] \\
 & & \text{else } Get[\![(C', v, C[open\ X\ in\ \square], t)]\!] \\
Get[\![(module\ X = struct\ D_1 ... D_n\ end\ in\ C', v, C, p)]\!] & = & Get[\![(C', v, C[module\ X = struct\ D_1 ... D_n\ end\ in\ \square], p)]\!] \\
Get[\![(type\ x = \tau\ in\ C', v, C, p)]\!] & = & Get[\![(C', v, C[type\ x = \tau\ in\ \square], p)]\!]
\end{array}
$$

Figure 24. *Get* for the module language

$$
\begin{array}{rcl}
SearchMod' : definition\ list * var * \\
context * (context * term)\ option & \rightarrow & (context * term)\ option \\
SearchMod'[\![([], v, C, p)]\!] & = & p \\
SearchMod'[\![((let\ x = M) :: r, v, C, p)]\!] & = & \text{if } v = v' \text{ then } SearchMod'[\![(r, v, C[let\ x = M\ in\ \square)], \text{Some}(C, M))]\!] \\
 & & \text{else } SearchMod'[\![(r, v, C[let\ v' = M\ in\ \square], p)]\!] \\
SearchMod'[\![((type\ x = \tau) :: r, v, C, p)]\!] & = & SearchMod'[\![(r, v, C[type\ x = \tau\ in\ \square], p)]\!]
\end{array}
$$

$$
\begin{array}{rcl}
SearchMod : context * var * var * context * \\
(context * term)\ option & \rightarrow & (context * term)\ option \\
SearchMod[\![(\square, V, v, C, p)]\!] & = & p \\
SearchMod[\![(\lambda x.C', V, v, C, p)]\!] & = & SearchMod[\![(C', V, v, C[\lambda x.\square], p)]\!] \\
SearchMod[\![(fix\ f\ v_1 ... v_n \rightarrow C', V, v, C, p)]\!] & = & SearchMod[\![(C', V, v, C[fix\ f\ v_1 ... v_n \rightarrow \square], p)]\!] \\
SearchMod[\![(let\ x = M\ in\ C', V, v, C, p)]\!] & = & SearchMod[\![(C', V, v, C[let\ x = M\ in\ \square], p)]\!] \\
SearchMod[\![(open\ X\ in\ C', V, v, C, p)]\!] & = & SearchMod[\![(C', V, v, C[open\ X\ in\ \square], p)]\!] \\
SearchMod[\![(module\ X = struct\ D_1 ... D_n\ end & = & \text{if } X = V \text{ then let t = } SearchMod'[\![(D_1 ... D_n, v, C, \text{None})]\!] \text{ in} \\
\quad in\ C', V, v, C, p)]\!] & & \quad (\text{if } t = \text{None then} \\
 & & \quad\quad SearchMod[\![(C', V, v, C[module\ X = struct\ D_1 ... D_n\ end\ in\ \square], \text{None})]\!] \\
 & & \quad \text{else } SearchMod[\![(C', V, v, C[module\ X = struct\ D_1 ... D_n\ end\ in\ \square], t)]\!]) \\
 & & \text{else } SearchMod[\![(C', V, v, C[module\ X = struct\ D_1 ... D_n\ end\ in\ \square], p)]\!] \\
SearchMod[\![(type\ x = \tau\ in\ C', V, v, C, p)]\!] & = & SearchMod[\![(C', V, v, C[type\ x = \tau\ in\ \square], p)]\!]
\end{array}
$$

Figure 25. The function *SearchMod* to search for the definition of modules

type error. Since the system automatically shows us possible fixes without our intervention, the system is useful if the programmer's intended fix is shown. Unfortunately, it does not always produce the intended program.

As visualizing tools of types, Simon, Chitil, and Hush [15] show TypeView that allows programmers to browse through the source code and to query the types of each expression. McAdam [11] displays types as graphs and extracts various facts from them that are useful for debugging. Our previous Emacs interface [17] is affected by these work, and we will continue to build such interface.

Marceau, Fisler, and Krishnamurthi [8–10] examine the impact (especially for novices) of error messages and the vocabulary used therein in detail and recommend what we can do for better error messages. Although their research does not directly mention a type debugger, it has large impact on the design of error messages used in our type debugger.

10. Conclusion

In this paper, we have fleshed out our thesis that it is possible and also practical to write a type debugger by piggy-backing the built-in type inferencer of an existing compiler. The key observation is that we only need the most general type tree with a simple property; such a tree can be constructed using the compiler's type inferencer. The property guided the design of our type debugger: we maintained contexts so that the property is satisfied all the time. We have illustrated the thesis with OCaml, and we have described how to handle a number of issues: simple types, let polymorphism, weak polymorphism, objects, and modules. Our design is in use in our laboratory and in our classrooms (four months courses for 40 students).

We plan to continue the present line of work as follows. First, we want to explore how far the idea presented in this paper scales. In particular, we are interested in supporting type classes [4] in Haskell. Treatment of type classes would be a nice first step toward the implementation of a type debugger for Haskell. Secondly, we want to perform thorough user tests. We will first design a useful user interface (like Emacs interface we designed in [17]), and then use the debugger for the introductory OCaml course starting from this April. From the user tests, we will be able to obtain various feedback including usefulness and how to effectively show the type information for novices. For this purpose, Marceau's work [8–10] will be of help. Finally, by collecting logs of the type debugger, we want to analyze erroneous programs and their fixes. Through such analyses, we might be able to suggest possible fixes by searching similar erroneous programs.

Acknowledgments

to be added later

References

[1] Chitil, O. "Compositional Explanation of Types and Algorithmic Debugging of Type Errors," *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming (ICFP'01)*, pp. 193–204 (2001).

[2] Garrigue, J. "Relaxing the value restriction," In Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming (LNCS 2998)*, pp. 196–213 (April 2004).

[3] Haack, C., J. B. Wells. "Type Error Slicing in Implicitly Typed Higher-Order Languages," *Science of Computer Programming - Special issue on 12th European symposium on programming (ESOP 2003)*, Volume 50 Issue 1-3 (2004).

[4] Hall, C., K. Hammond, S. P. Jones and P. Wadler. "Type classes in Haskell," *ACM Transactions on Programming Languages and Systems (TOPLAS)* Volume 18 Issue 2 (1996).

[5] Heeren, B., J. Hage. "Parametric Type Inferencing for Helium," Technical Report UU-CS-2002-035, Utrecht University, 2002.

[6] Lee, O., K. Yi. "Proofs about a Folklore let-polymorphic Type Inference Algorithm," *ACM Transactions on Programming Languages and Systems*, pp. 707-723 (1998).

[7] Lerner, B. S., M. Flower, D. Grossman, C. Chambers. "Searching for Type-Error Messages," *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*, pp. 425–434 (2007).

[8] Marceau, G., K. Fisler, and S. Krishnamurthi "Measuring the Effectiveness of Error Messages Designed for Novice Programmers," *Workshop on Scheme and Functional Programming*, 14 pages, (August 2010).

[9] Marceau, G., K. Fisler, and S. Krishnamurthi "Measuring the Effectiveness of Error Messages Designed for Novice Programmers," *Proceedings of the 42nd ACM technical symposium on Computer science education (SIGCSE '11)*, pp. 499–504 (March 2011).

[10] Marceau, G., K. Fisler, and S. Krishnamurthi "Mind Your Language: On Novices' Interactions with Error Messages," *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software (Onward! 2011)*, pp. 3–17 (October 2011).

[11] McAdam, B. J. "Generalising techniques for type debugging," In Trends in Functional Programming, chapter 6. Intellect, (2000).

[12] Nilsson, H. *Declarative Debugging for Lazy Functional Languages*, PhD thesis, Linköping, Sweden (1998).

[13] Shapiro, E. Y. *Algorithmic Program Debugging*, MIT Press (1983).

[14] Silva, J., and O. Chitil. "Combining Algorithmic Debugging and Program Slicing," *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming (PPDP'06)*, pp. 157–166 (2006).

[15] Simon, A., O. Chitil and F. Huch. "Typeview: A tool for understanding type errors," *Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages,* PP. 63–69 (2000).

[16] Stuckey, P. J., M. Sulzmann, J. Wazny, "Interactive type debugging in Haskell," *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell (Haskell '03)*, pp. 72–83 (2003).

[17] Tsushima, K., and K. Asai. "Report on an OCaml type debugger," *ACM SIGPLAN Workshop on ML*, 3 pages, (2011).

[18] Wand, M. "Finding the Source of Type Errors," *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '86)*, pp. 38–43 (1986).

[19] Yang, J., G. Michaelson., P. Trinder., and J. B. Wells. "Improved Type Error Reporting," *International Workshop on Implementation of Functional Languages*, pp. 71–86 (2000).

The Design of Scalable Distributed Erlang

Natalia Chechina[1], Phil Trinder[1], Amir Ghaffari[1], Rickard Green[2], Kenneth Lundin[2], and Robert Virding[3]

[1] Heriot-Watt University, Edinburgh, EH14 4AS, UK
[2] Ericsson AB, 164 83 Stockholm, Sweden
[3] Erlang Solutions AB, 113 59 Stockholm, Sweden
http://www.release-project.eu

Abstract. The multicore revolution means that the number of cores in commodity machines is growing exponentially. Many expect 100,000 core clouds/platforms to become commonplace, and the best predictions are that core failures on such an architecture will become relatively common, perhaps one hour mean time between core failures. The RELEASE project aims to scale Erlang to build reliable general-purpose software, such as server-based systems, on massively parallel machines.
In this paper we present a design of Scalable Distributed (SD) Erlang – an extension of Distributed Erlang functional programming language for reliable scalability. The design focuses on three aspects of Erlang scalability: 1) scaling the number of Erlang nodes by eliminating transitive connections and introducing scalable groups (s_groups), 2) managing process placement in the scaled networks by introducing semi-explicit process placement, and 3) preserving Erlang reliability model.

Keywords: Erlang, functional programming, scalability, multi-core systems, massive parallelism

1 Introduction

General-purpose software is predominately written in mainstream programming languages, firmly planted in legacy software concepts, and the software industry is struggling for better ways to parallelise these languages. Current shared-memory technologies like OpenMP often work well for small scale problems, but applications are rapidly experiencing the inherent limitations of these technologies. MPI works well for large scale computations with a regular process structure and independent computations, for example classical High-Performance Computing (HPC) problems like computational fluid dynamics. However many general purpose applications have significant data dependencies, or irregular process structures. Simultaneously, increasing the number of cores increases the likelihood of failures: a system with 10^5 cores might experience a core failure every 50 minutes in addition to any other failures [31]. Handling partial failures in massively-parallel applications inevitably introduces dependencies between components and calls for coordination logic that cannot easily be expressed using current programming language technologies.

Erlang [11] is a functional programming language. Its concurrency-oriented programming paradigm is novel in being very high level, predominantly stateless, and having both parallelism and reliability built-in rather than added-on. Building on this success, the user uptake of Erlang is exploding around the world and shifting from its telecom base into other sectors. Currently Erlang/OTP has inherently scalable computation and reliability models, but in practice scalability is constrained by the transitive sharing of connections between all nodes and by explicit process placement. The former implies that the virtual machine maintains data structures quadratic in the number of nodes and the latter makes constructing large dynamic or irregular process structures challenging. Moreover programmers need support to engineer applications at this scale and existing profiling and debugging tools do not scale, primarily due to the volumes of trace data generated.

We target reliable scalable general purpose computing on stock heterogeneous platforms. Our application area is that of general server-side computation, e.g. a web or messaging server. This form of computation is ubiquitous, in contrast to more specialised forms such as traditional high-performance computing. Moreover, this is computation on stock platforms, with standard hardware, operating systems and middleware, rather than on more specialised software stacks on specific hardware.

To extend the Erlang concurrency-oriented paradigm to large-scale reliable parallelism (10^5 cores) we propose an extension to the Erlang language, Scalable Distributed Erlang (SD Erlang), for reliable scalability. Key goals in scaling the computation model are to provide mechanisms for controlling locality and reducing connectivity, and to provide performance portability. The goal in scaling the reliability model is to preserve Erlang's sophisticated and effective reliability mechanisms of first class processes and supervision behaviours in the presence of locality and connectivity controls. The SD Erlang name is used only as a convenient means of identifying the extensions we propose: we expect the extensions to become standard Erlang in the future.

The rest of the paper is organised as follows. We start with an overview of related work and background information (Section 2). For a language to scale in-memory and persistent data structures, together with computation must scale (Section 3). The primary Erlang in-memory data structure Erlang Term Storage (ETS) is implemented inside Erlang Virtual Machine (VM), so any scalability issues will be addressed by the VM team of the RELEASE project. In terms of persistent data structures we believe that such databases as Riak [3] and Cassandra [15] will be able to meet the target scalability requirements. The computation scalability is a language level issue and we address it by eliminating transitive connections of Erlang nodes and introducing a semi-explicit process placement (Sections 4). Finally, we discuss the design validation exemplars (Section 5) and provide conclusion together with the future work (Section 6).

2 Related Work

This section covers related work and provides background information. First, we discuss typical hardware architectures we might expect in the next 4-5 years in Section 2.1. Actor languages and Erlang functional programming language are covered in Sections 2.2 and 2.3 respectively. Finally, the RELEASE project overview is provided in Section 2.4.

2.1 Architecture Trends

To make predictions in computer science even for the next 4-5 years is a hard job as the field is very young and moves forward much faster than any other branch of science. However, to understand limitations of today Erlang we need to get an idea of typical hardware architectures that will use SD Erlang in the next few years. Currently the main factors that shape trends in computer architectures are as follows: memory size/bandwidth, energy consumption, and cooling. Below we provide a brief analysis of these factors and discuss their impact on the development of the hardware architectures.

Memory size/bandwidth. As the number of cores goes up the memory bandwidth goes down, and the larger number of cores shares the same memory the larger memory is required. DRAM-based main memory systems are about to reach the power and cost limit. Currently, the main two candidates to replace DRAM are Flash memory and Phase Change Memory (PCM). Both types are much slower than DRAM, i.e. 2^{11} and 2^{17} processor cycles respectively for a 4GHz processor in comparison with 2^9 processor cycles of DRAM, but the new technologies provide a higher density in comparison with DRAM [26].

Energy consumption and cooling are the main constrains for the high core density. Moreover, the cooling is also a limitation of the silicon technology scal-

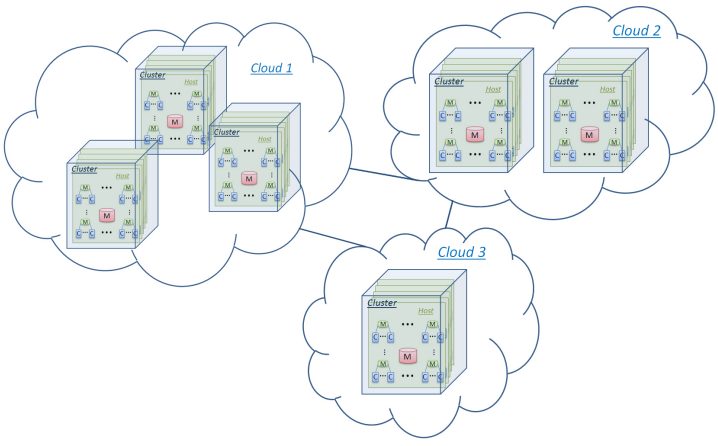


Fig. 1. A Typical Server Architecture

ing [37]. To save energy and maximize compute performance supercomputers exploit small and simple cores apposed to large cores. Many architectures, especially HPC architectures, exploit GPUs [10]. GPUs accelerate regular floating point matrix/vector operations. The high throughput servers that RELEASE targets do not match this pattern of computation, and GPUs are not exploited in the server architectures we target. The air cooling might be replaced by one of the following technologies: 2D and 3D micro-channel cooling [17], phase-change cooling [22], spot cooling [36], or thermal-electric couple cooling [29].

From the above we anticipate the following typical server hardware architecture. A host will contain ∼4–6 SMP core modules where each module will have ∼32–64 cores. Analysis of the Top 500 supercomputers that always lead the computer industry illuminating the next 4–5 year computer architecture trends allows us to assume that ∼100 hosts will be grouped in a cluster, and ∼1–5 clusters will form a cloud. Therefore, the trends are towards a NUMA architecture. The anticipated typical server architecture is presented in Figure 1.

2.2 Actor Languages and Frameworks

An actor model was introduced in 1973. Actors are independent entities that asynchronously exchange messages, and perform actions depending on these messages. There is a number of actor-based languages and frameworks such as Akka [2], Axum [20], E [28], Erlang [11], Kilim [30], Ptolemy [27], SALSA [34], Scala [4], and Smalltalk [16]. In this section we cover only Scala and Akka. Erlang is discussed in Section 2.3.

Scala is a statically typed programming language that combines features of both object-oriented and functional programming languages [4]. Scala has been designed to interact with mainstream platforms such as Java and C#. Currently, Scala is implemented on Java and .NET platforms. Scala has a pure oriented model similar to Smalltalk [16] where values are objects, and operations are messages. Operator names are treated as identifies, and identifies between two expressions are treated as method calls. Scala is also a functional language due to treating functions as values. The language supports such functions as nested, anonymous, curried, and higher order functions. Scala supports parameterisation, abstract members, and classes to model Erlang type actors [24].

Akka is an event driven middleware framework to build reliable distributed applications [2]. Akka is implemented in Scala. A fault tolerance in Akka is implemented using similar to Erlang 'Let it crash' philosophy and supervisor hierarchies [33]. An actor can only have one supervisor which is the parent supervisor but similar to Erlang actors can monitor each other. Due to possibility to create an actor within a different JVM two paths can be used to reach an actor: logical and physical. Logical path follows parental supervision links toward the root. Physical actor path starts at the root of the system where the actual actor object resides, and never spreads over multiple JVMs. Akka uses remote to local approach via optimisation [35]. In Akka multiple shreds can execute actions on shared memory. Like Erlang Akka does not support guaranteed delivery. A cluster support is planned to be introduced in Akka.

2.3 Erlang

Erlang is a functional general purpose concurrent programming language designed in 1986 at Ericsson computer science laboratory [11]. Erlang was influenced by a number of languages such as ML [21], Miranda [32], ADA [19], and Prolog [38]. Erlang was designed to meet requirements of distributed, fault-tolerant, massively concurrent, and soft-real time systems. Erlang is a dynamically typed language. Distributed Erlang was introduced to allow autonomous Erlang Virtual Machines (VMs) to work together when they are situated either on the same or different computers. In Erlang a collection of processes work together to solve a particular problem. The processes are lightweight and communicate with each other by exchanging asynchronous messages [39].

Erlang concurrency differs from the most other programming languages in that concurrency is handled by the language and not by the operating system [8]. Some of the principles of the Erlang philosophy are as follows. *Share nothing* implies that isolated processes do not share memory and variables are not reusable, i.e. once a value is assigned it cannot be changed. *Let it crush* is a non-defensive approach that lets failing processes to crash, and then other processes detect and fix the problem. The approach also provides clear and compact code [7].

2.4 RELEASE Project

The RELEASE project aims to scale the radical concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel machines. Concurrency-oriented programming is distinctive as it is based on highly-scalable lightweight processes that *share nothing*. The trend-setting concurrency-oriented programming model we use is Erlang/OTP. Erlang/OTP provides high-level coordination with concurrency and robustness built-in: it can readily support 10,000 processes per core, and transparent distribution of processes across multiple machines, using message passing for communication. Moreover, the robustness of the Erlang distribution model is provided by hierarchies of supervision processes which manage recovery from software or hardware errors.

Currently Erlang/OTP has inherently scalable computation and reliability models, but in practice scalability is constrained by the transitive sharing of connections between all nodes and by explicit process placement. Moreover programmers need support to engineer applications at this scale and existing profiling and debugging tools do not scale, primarily due to the volumes of trace data generated. In the RELEASE project we tackle these challenges working at three levels: evolving the Erlang virtual machine, evolving the language to Scalable Distributed (SD) Erlang, developing a scalable Erlang infrastructure.

3 SD Erlang Design Overview

In this section we provide our vision of requirements for Erlang scalability and principles behind design decisions taken in Section 4. We believe that for SD

Erlang to scale in-memory and persistent data structures together with computation must be taken into account.

Scalable in-memory and persistent data structures. When an application scales it requires support from in-memory and persistent data storages to handle a large number of processes and data. Therefore, in-memory and persistent data structures need to be able to scale to the same magnitude as the application that implores them (Section 3.1).

Scalable computation. From the analysis of the typical Erlang exemplars in Section 5 we have identified two main scalability issues. These are a fully connected network together with transitive connections and explicit placement. A fully connected network and transitive connections prevent a network of Erlang nodes to scale because it becomes not feasible to maintain a good performance in a network of more than a hundred of nodes. Whereas an explicit placement requires a programmer to be aware of all Erlang nodes in the network for every process which is again not feasible (Section 4).

To design SD Erlang we came up with two types of principles: general and reliable scalability. The general principles include our view on the aspects of the language that we want to preserve and implementation level of the modifications. The general principles are as follows.

- *Preserving the Erlang philosophy and programming idioms.*
- *Minimal language changes*, i.e. minimizing the number of new constructs but rather reusing of existing constructs.
- *Working at Erlang level* rather than VM level as far as possible.

The reliable scalability principles include concepts that we want to either preserve or avoid when scaling Erlang. They are as follows.

- *Avoiding global sharing*, i.e. global names, bottlenecks, and using groups instead of fully connected networks.
- *Introducing an abstract notion of communication architecture*, e.g. locality/affinity and sending disjoint work to remote hosts.
- *Avoiding explicit prescription*, e.g. replacing spawning on named node with spawning on group of nodes, and automating load management.
- *Keeping the Erlang reliability model* unchanged as far as possible, i.e. linking, monitoring, supervision.

3.1 Scalable Data Structures

Scalable In-Memory Data Structures. The primary Erlang in-memory data structure is Erlang Term Storage (ETS). ETS is a data structure to associate keys with values, and is a collection of Erlang tuples, i.e. tuples are inserted and extracted from an ETS table based on the key. An ETS is memory resident and provides large key-value lookup tables. Data stored in the tables is transient. ETS tables are implemented in the underline runtime system as a BIF inside the Erlang VM, and are not garbage collected. ETS tables are stored in a separate storage area not associated with normal Erlang process memory. The size of ETS

tables depends on the size of a RAM. An ETS table is owned by the process that has created it and is deleted when a process terminates. The process can transfer the table ownership to another local process. The table can have the following read/write access [12]: 1) *private*, i.e. only the owner can read and write the table, 2) *public*, i.e. any process can read and write the table, 3) *protected*, i.e. any process can read the table but only the owner can write it.

ETS tables provide limited support for concurrent updates [11]. That is inserting a new element may cause a rehash of elements within the table; when a number of processes write or delete elements concurrently from the table the following outcomes are possible: a runtime error, `bad arg` error, or undefined behaviour, i.e. any element may be returned. As the number of SMP cores increases the number of Erlang nodes and processes also increase. This can lead to either a bottleneck if the table is private/protected or undefined outcomes if the table is public. ETS tables are implemented inside the Erlang VM, and hence any scalability issues will be addressed by the VM team of the RELEASE project.

Scalable Persistent Data Structures. We have analysed a number of DataBase Mangament Systems (DBMSs) for Erlang such as Mnesia [23], Riak [3], CoachDB [5], and Cassandra [15]. Below we have summarised the main principles and desirable features required for highly available and scalable databases. We believe that such DBMSs as Riak and Cassandra will be able to meet the target scalability requirements [25].

Fragmenting data across distributed nodes. 1) Decentralized approaches are preferable as they show a better throughput by spreading the load over a large number of servers and increase availability by removing a single point of failure; 2) The placement of replicas should be handled systematically and automatically, i.e. location transparency. 3) A node departure or arrival should only affect the node immediate neighbours whereas other nodes remain unaffected.

Replicating data across distributed nodes. 1) A decentralized model such as P2P is desirable. 2) An asynchronous replication where consistency is sacrificed to achieve higher availability, i.e. from the CAP theorem [18] a database cannot simultaneously guarantee consistency, availability, and partition-tolerance.

Partition tolerance, i.e. a system continues to operate despite of connection loss between some nodes. We anticipate the target architecture to be loosely coupled; therefore, partition failures are highly expected. Again from the CAP theorem by putting stress on availability we must sacrifice strong consistency to achieve partition-tolerance and availability.

4 Scalable Actor Computation

We have identified two main Distributed Erlang issues that prevent scalability, these are transitive connections and explicit placement. Transitive connections are the reason of inability to scale beyond a hundred of nodes, whereas explicit placement is a restriction that prevents a programmer to easily manipulate the available nodes.

In this section we introduce extension of the Distributed Erlang – Scalable Distributed (SD) Erlang – to effectively operate when the number of hosts, cores, nodes, and processes scales. We start with a discussion of scalability limitations of Distributed Erlang in Section 4.1. To scale a network of Erlang nodes we introduce scalable groups (s_groups) in Section 4.2. S_groups aim to eliminate transitive connections, i.e. nodes have transitive connections with nodes of the same s_group and non-transitive connections with other nodes. To manage process placement we introduce semi-explicit placement and `choose_node/1` function in Section 4.3. So that a process can be spawned, for example, to an s_group or a particular communication distance.

4.1 Distributed Erlang Scalability Limitations

Figure 2 illustrates Erlang's support for concurrency, multicores and distribution. A blue rectangle represents a host with an IP address, and a red arc represents a connection between nodes. Multiple Erlang processes may execute in a node, and a node can exploit multiple processors, each having multiple cores. Erlang supports single core concurrency as a core may support as many as 10^8 lightweight processes [1]. In the Erlang distribution model a node may be on a remote host, and this is almost entirely transparent to the processes. Hosts need not be identical, nor do they need to run the same operating system.

Erlang currently delivers reliable medium scale concurrency, supporting up to 10^2 cores, or 10^2 distributed memory processors. However, the scalability of a distributed system in Erlang is constrained by the transitive sharing of connections between all nodes and by explicit process placement. The transitive sharing of connections between nodes means that the underlying implementation needs to maintain data structures that are quadratic in the number of processes, rather than considering the communication locality of the processes. While it is possible to explicitly place large numbers of processes in a regular, static way (as for conventional HPC computations), explicitly placing the irregular or dynamic processes required by many servers and other general purpose applications is far more challenging. Erlang/OTP has a world leading language level reliability. The challenge is to maintain this reliability at massive scale.

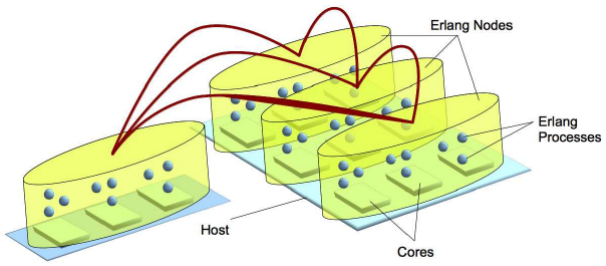


Fig. 2. Connections in s_groups

4.2 Network Scalability

To allow scalability of networks of nodes the existing scheme of transitive connection sharing in Distributed Erlang should be changed as it is not feasible for a node to maintain connections to tens of thousands of nodes, i.e. the larger the network of Erlang nodes the more 'expensive' it becomes on each node to keep up-to-date replications of global names and global states, and periodic checking of connected nodes. Instead we propose to use overlapping scalable groups (s_groups) where nodes would have transitive connections within their s_group and non-transitive connections with nodes of other s_groups. The idea of s_groups is *similar* to the existing in Distributed Erlang hidden global_groups in the following: 1) each s_group has its own name space; 2) transitive connections are only with nodes of the same s_group. The *differences* with hidden global_groups are in that 1) a node can belong to an unlimited number of s_groups, and 2) information about s_groups and nodes is not globally collected and shared.

In SD Erlang nodes with no asserted s_group membership belong to a notional group *G0* that follows Distributed Erlang rules and allows backward compatibility with Distributed Erlang. By the backward compatibility we mean that when nodes run the same VM version they may use or not use s_groups and still be able to communicate with each other. Therefore, s_groups are not compulsory but rather a tool a programmer may use to scale a network of nodes.

Types of s_groups. To allow programmers flexibility and provide an assistance in grouping nodes we propose s_groups to be of different types, i.e. when an s_group is created a programmer may specify parameters against which a new s_group member candidate can be checked. If a new node satisfies an s_group restrictions then the node becomes the s_group member, otherwise the membership is refused. The following parameters can be taken into account: communication distance, security, available code, specific hardware and software requirements. A programmer may also introduce his/her own s_group types on the basis of some personal preferences. The information about specific resources can be collected by introducing node self awareness, i.e. a node is aware of its execution environment and publishes this information to other nodes.

We also propose the following features: *a)* a node can establish a direct connection with any other node, *b)* nodes can have short lived connections, and *c)* a host can have an unlimited number of nodes. We do not consider any language constructs to provide programmers control over cores, i.e. the lowest level a programmer may control in terms of where a process can be spawned is a node.

s_group Functions. We propose a number of functions to support s_group employment, some of them are listed below. The functions may be changed during the development. The final implementation will be decided during actual SD Erlang code writing and will depend on the functions that programmers find useful.

1. Creating a new s_group, e.g.
   `new_s_group(S_GroupName, [Node]) -> ok | {error, ErrorMsg}`

2. Adding new nodes to an existing s_group, e.g.
   `add_node_s_group(S_GroupName, [Node]) -> ok | {error, ErrorMsg}`
3. Monitoring all nodes of an s_group, e.g.
   `monitor_s_group(S_GroupName) -> ok | {error, ErrorMsg}`
4. Listing nodes of a particular s_group, e.g.
   `s_group_nodes(S_GroupName) -> [Node] | {error, ErrorMsg}`
5. Connecting to all nodes of a particular s_group, e.g.
   `connect_s_group(S_GroupName) -> [boolean() | ignored]`
6. Disconnecting from all nodes of a particular s_group, e.g.
   `disconnect_s_group(S_GroupName) -> boolean() | ignored`

Example. Assume we start six nodes A, B, C, D, E, F, and initially the nodes belong to no s_group. Therefore, all these nodes belong to notional group $G0$ (Figure 3(a)). Fist, on node A we create a new s_group $G1$ that consists of nodes A, B, and C, i.e. `new_s_group(G1, [A, B, C])`. Note that a node belongs to group $G0$ only when this node does not belong to any s_group. When nodes A, B, and C become members of an s_group they may still keep connections with nodes D, E, F but now connections with these nodes are non-transitive. If connections between nodes of s_group $G1$ and group $G0$ are time limited then the non-transitive connections will be lost over some time (Figure 3(b)). Then on node C we create s_group $G2$ that consists of nodes C, D, and E. Nodes D, and E that now have non-transitive connections with node F may disconnect from the node using function `disconnect_s_group(G0)`. Figure 3(c) shows that node C does not share information about nodes A and B with nodes D and E. Similarly, when nodes B and E establish a connection they do not share connection information with each other (Figure 3(d)).

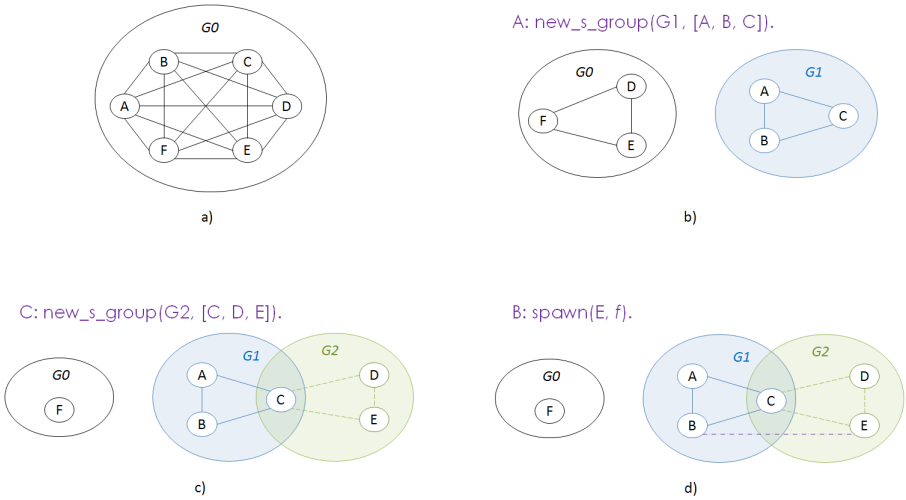


Fig. 3. Connections in s_groups

4.3 Semi-Explicit Placement

For some problems, like matrix manipulations, optimal performance can be obtained on a specific architecture by explicitly placing threads within the architecture. However, many problems do not exhibit this regularity. Moreover, explicit placement prevents performance portability: the program must be rewritten for a new architecture, a crucial deficiency in the presence of fast-evolving architectures. We propose a dynamic semi-explicit and architecture aware process placement mechanism. The mechanism does not support the migration of processes between Erlang nodes. The semi-explicit placement is influenced by Sim-Diasca process placement [14] and architecture aware models [9].

In Sim-Diasca a computing load is induced by a simulation and needs to be balanced over a set of nodes. By default, model instances employed by Erlang processes are dispatched to computing nodes using a round-robin policy. The policy proved to be sufficient for most basic uses, i.e. a large number of instances allows an even distribution over nodes. However, due to bandwidth and latency for some specifically coupled groups of instances it is preferable for a message exchange to occur inside the same VM rather than between distant nodes. In this case, a developer may specify a placement hint when requesting a creation of instances. The placement guarantees that all model instances created with the same placement hint are placed on the same node. This allows the following: a) to co-allocate groups of model instances that are known to be tightly coupled, b) to preserve an overall balancing, and c) to avoid model level knowledge of the computing architecture. In [9] to limit the communication costs for small computations, or to preserve data locality, the authors proposes to introduce communication levels and specify the maximum distance in the communication hierarchy that the computation may be located. Thus, sending a process to level 0 means the computation may not leave the core, level 1 means a process may be located within the shared memory node, level 2 means that process may be located to another node in a Beowulf cluster, and level 3 means that a process may be located freely to any core in the machine.

For the SD Erlang we propose that a process could be spawned either to an s_group, to s_groups of a particular type, or to nodes on a given distance. From a range of nodes the target node can be picked either randomly or on the basis of load information.

Load Management. When a node is picked on the basis of load an important design decision is the interaction between two main load management components, i.e. information collection and decision making. The components can be either merged together and implemented as one element or implemented independently from each other. We propose to implement information collection and decision making as one element, i.e. a load server. Its responsibility will be collecting information from the connected nodes and deciding where a process can be spawned when a corresponding request arrives. It seems that one load server per node is an appropriate number of load servers per node for SD Erlang. In this case the decisions are made within the node (in comparison with one load server per s_group, a host, and a group of hosts) and load information

redundancy level is not too high (in comparison with one per group of processes and a multiple number of load servers per node).

chose_node. We propose to introduce a new function `chose_node/1`. The function will return a node ID where the process should be spawned. The node will be picked on the basis of restrictions identified by the programmer, e.g. s_groups, s_group types, minimum/maximum/ideal communication distances. The function can be written in SD Erlang as follows.

```
chose_node(Restrictions) -> node()
Restrictions = [Restriction]
Restriction = {s_group_name, S_GroupName}
| {s_group_type, S_GroupType}
| {min_dist, MinDist :: integer() >= 0}
| {max_dist, MaxDist :: integer() >= 0}
| {ideal_dist, IdealDist :: integer() >= 0}
```

We deliberately introduce `Restrictions` as a list of tuples to allow the list of restrictions to be extended in the future. A process spawning may look as follows:

```
start() ->
TargetNode = chose_node([{s_group, S_Group}, {ideal_dist, IdealDist}]),
spawn(TargetNode, fun() -> loop() end).
```

4.4 Summary.

To enable scalability of network of nodes we propose a *new s_group library* for Erlang. 1) Grouping nodes in s_groups where s_groups can be of different types, and nodes can belong to many s_groups. 2) Transitive connections between nodes of the same s_group and non-transitive connections with all other nodes. Direct non-transitive connections are optionally short lived, e.g. time limited.

To enable *semi-explicit placement* and load management we propose the following constructs. 1) Function `chose_node(Restrictions) -> node()` where the choice of a node can be restricted by a number of parameters, such as s_groups, s_group types, and communication distances. 2) The nodes may be picked randomly or on the basis of load. We assume that when a process is spawned using semi-explicit placement it is a programmer responsibility to ensure that the prospective target node has the required code. If the code is missing an error is returned.

5 Design Validation Exemplars

To validate SD Erlang design presented in Section 4 we have done a theoretical validation of five Erlang applications: Sim-Diasca [14], Orbit, Mandelbrot set, Moebious, and Riak [3]. Here, we only discuss Moebious. The description of the remaining applications is presented in [25].

Moebius is a continuous integration system recently developed by Erlang Solutions. Moebius aims to provide users an automated access to various cloud

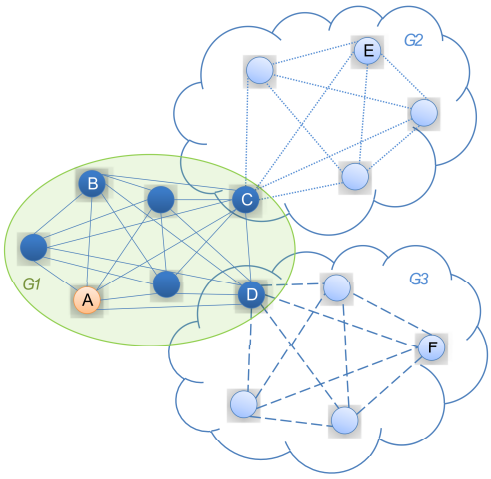


Fig. 4. Mandelbrot Set Node Grouping

providers such as Amazon EC2. The system has two types of nodes: master node and moebius agents. The master node collects global information and makes decisions to start and stop nodes. The moebius agents are located on the utilised nodes and periodically send state information to the master node. Currently, moebius agents are only connected to the master node via HTTP but in the future there are plans to move Moebius to SD Erlang and build a hierarchical master structure.

A top level Moebius algorithm is as follows. A user is asked to indicate the requirements, e.g. hardware configuration, software configuration, and a description on how the initial master node should start the remaining nodes in the cloud (Figure 4). Thus, a hierarchical organization of nodes can be easily set from top to bottom. Additional s_groups from nodes of different levels can also be formed if needed. An SD Erlang Moebius may require the following. 1) The s_groups may be grouped on the basis of different factor such as communication locality, security, and availability of a particular hardware or software; therefore, custom s_groups types are required. 2) Nodes and s_groups will dynamically appear and disappear depending on the user current requirements. 3) Moebius master nodes most probably will be organised in a hierarchical manner, so nodes will not need to directly communicate with each other. 4) The number of s_groups most probably will be much less than the number of nodes.

Table 1 provides a summary of the exemplar requirements for scalable implementations. Thus, s_groups may be either static, i.e. once created nodes rarely leave and join their s_groups, or dynamic, i.e. nodes and s_groups are constantly created and deleted from the network. S_groups may be formed on the basis of locality (Sim-Diasca and Mandelbrot set), Hash table (Orbit), Preference list (Riak), or programmers' and users' preferences (Moebius). *'Yes/No'* indi-

No	Property	Sim-Diasca	Orbit	Mandel-brot set	Moebius	Riak
			s_groups			
1	**Static/Dynamic**	Static	Static	Static	Dynamic	Dynamic
2	**Grouping**	Locality	Hash table	Locality	Multiple	Preference list
3	**Custom types**	Yes	No	No	Yes	No
			General			
4	**Number of nodes and s_groups**	$N_g << N_n$	$N_g << N_n$	$N_g << N_n$	$N_g << N_n$	$N_g >= N_n$
5	**Short lived connections**	Yes	Yes	No	No	Yes
6	**Semi-explicit placement**	Yes	No	Yes	No	No

Table 1. Exemplar Summary

cates whether an application requires a particular feature. For instance, some scalable exemplars require custom s_group types, short lived connections, and semi-explicit placement. Such applications like Riak may have the number of s_groups compatible with the number of nodes.

6 Conclusion and Future Work

This paper presents the design of Scalable Distributed (SD) Erlang: a set of language-level changes that aims to enable Distributed Erlang to scale for server applications on commodity hardware with at most 10^5 cores. The core elements of the design are to provide scalable in-memory data structures, scalable persistent data structures, and a scalable computation model. The scalable computation model has two main parts: scaling networks of Erlang nodes and managing process placement on large numbers of nodes. To tackle the first issue we have introduced s_groups that have transitive connections with nodes of the same s_group and non-transitive connections with nodes of other s_groups. To resolve the second issue we have introduced semi-explicit placement and `choose_node/1` function. Unlike explicit placement a programmer may spawn a process to a node from a range of nodes that satisfy predefined parameters, such as s_group, s_group type, or communication distance.

Erlang follows a functional programming idiom of having a few primitives and building powerful abstractions over them. Examples of such abstractions are algorithm skeletons [13] that abstract common patterns of parallelism, and behaviour abstractions [6] that abstract common patterns of distribution. We plan to develop SD Erlang behaviour abstractions over primitives presented in Sections 4.2 and 4.3. We expect the behaviours to become apparent during the work on the case studies and scalable infrastructure.

Acknowledgements. We would like to thank all our colleagues who work on the RELEASE project. This work has been supported by the European Union grant RII3-CT-2005-026133 'SCIEnce: Symbolic Computing Infrastructure in Europe', IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software', and by the UK's Engineering and Physical

Sciences Research Council grant EP/G055181/1 'HPC-GAP: High Performance Computational Algebra and Discrete Mathematics'.

References

1. *Erlang/OTP Efficiency Guide, System Limits*, 2011. http://erlang.org/doc/efficiency_guide/advanced.html#id67011.
2. Akka: Event-driven middleware for Java and Scala, July 2012. http://www.typesafe.com/technology/akka.
3. Basho documentation, July 2012. http://wiki.basho.com.
4. The Scala programming language, July 2012. http://www.scala-lang.org.
5. J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, 1st edition, 2010.
6. J. Armstrong. *Making Reliable Distributed Systems in the Presence of Sofware Errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.
7. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
8. J. Armstrong. Erlang. *Commun. ACM*, 53:68–75, 2010.
9. M. Aswad, P. Trinder, and H.-W. Loidl. Architecture aware parallel programming in Glasgow parallel Haskel (GpH).
10. S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.
11. F. Cesarini and T. Simon. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media Inc., 1st edition, 2009.
12. M. Christakis and K. Sagonas. Static detection of race conditions in Erlang. In M. Carro and R. Peña, editors, *Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 119–133. Springer Berlin / Heidelberg, 2010.
13. M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
14. EDF. The sim-diasca simulation engine, July 2012. http://sim-diasca.com/.
15. T. A. S. Foundation. Apache Cassandra, 2012. http://cassandra.apache.org/.
16. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
17. W. H., M. Stan, S. Gurumurthi, R. Ribando, and K. Skadron. Interaction of scaling trends in processor architecture and cooling. In *SEMI-THERM '10*, pages 198–204. IEEE Computer Society, 2010.
18. D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon highly available key-value store. In *SIGOPS '07*, pages 205–220, New York, NY, USA, 2007. ACM Press.
19. J. D. Ichbiah, B. Krieg-Brueckner, B. A. Wichmann, J. G. P. Barnes, O. Roubine, and J.-C. Heliard. Rationale for the design of the Ada programming language. *SIGPLAN Not.*, 14(6b):1–261, 1979.
20. Microsoft. *Axum Programmer's Guide*, 2009. http://www.microsoft.com.
21. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.
22. S. Murthy, Y. Joshi, and W. Nakayama. Orientation independent two-phase heat spreaders for space constrained applications. *Microelectronics Journal*, 34(12):1187–1193, 2003.

23. H. Nilsson, C. Wikström, and E. T. Ab. Mnesia - an industrial DBMS with transactions, distribution and a logical query language. In *CDSAA'96*, Kyoto, Japan, 1996.
24. M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the Scala programming language. Technical Report LAMP-REPORT-2006-001, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, 2004.
25. R. Project. Scalable reliable sd erlang design, July 2012. http://release-project.eu/documents/D3.1_main.pdf.
26. M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *SIGARCH Comput. Archit. News*, 37:24–33, 2009.
27. H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08*, pages 155–179, Berlin, Heidelberg, 2008. Springer-Verlag.
28. J. E. Richardson, M. J. Carey, and D. T. Schuh. The design of the E programming language. *ACM Trans. Program. Lang. Syst.*, 15(3):494–534, 1993.
29. G. Snyder, M. Soto, R. Alley, D. Koester, and B. Conner. Hot spot cooling using embedded thermoelectric coolers. In *SEMI-THERM '06*, pages 135–143. IEEE Computer Society, 2006.
30. S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP '08*, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.
31. A. Trew. Parallelism and the exascale challenge. Distinguished Lecture. St Andrews University, 2010.
32. D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *FPCA '85*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
33. Typesafe Inc. *Akka Documentation: Release 2.1 - Snapshot*, July 2012. http://www.akka.io/docs/akka/snapshot/.
34. C. A. Varela, G. Agha, W.-J. Wang, T. Desell, K. E. Maghraoui, J. LaPorte, and A. Stephens. *The SALSA Programming Language: 1.1.2 Release Tutorial.* Rensselaer Polytechnic Institute, Troy, New York, 2007. http://www.cs.rpi.edu/research/groups/wwc/salsa/tutorial/main.pdf.
35. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer Berlin / Heidelberg, 1997.
36. E. Wang, L. Zhang, L. Jiang, J. Koo, J. Maveety, E. Sanchez, K. Goodson, and T. Kenny. Micromachined jets for liquid impingement cooling of VLSI chips. *Microelectromechanical Systems*, 13(5):833–842, 2004.
37. J. Warnock. Circuit design challenges at the 14nm technology node. In *DAC '11*, pages 464–467, New York, NY, USA, 2011. ACM.
38. D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog – the language and its implementation compared with Lisp. *SIGPLAN Not.*, 12(8):109–115, 1977.
39. C. Wikström. Distributed programming in Erlang. In *PASCO'94*, pages 412–421, 1994.

A Concurrent Persistent Functional Language Towards Practical Functional Databases

Lesley Wevers[1], Marieke Huisman[1], and Ander de Keijzer[2]

[1] University of Twente, Netherlands
[2] Windesheim University of Applied Sciences, Netherlands

Abstract. Functional languages are close to the declarative model of database query languages, making functional languages well suited to the querying and manipulation of databases, and thereby providing a good basis for the implementation of transaction processing systems. This paper describes a prototype implementation of a persistent functional language for transaction processing, where transactions can be executed concurrently through lazy evaluation of states, and storing states in persistent memory is handled transparently. Our prototype includes a language for the specification of transactions, as well as a new method for load balancing in graph reduction based on sharing results between threads and randomization of their reduction order. We also discuss a new method for storing the state to persistent memory. Furthermore, we found that lazy evaluation of states has some problems in practice, for which we provide a theoretical solution. Finally, we have evaluated our prototype implementation through some practical experiments.

1 Introduction

Functional languages provide an interesting basis for the implementation, querying and manipulation of databases [16]. Queries in a purely functional language can be executed in parallel without explicit management of hardware resources, while lazy evaluation may reduce I/O and provide a basis for concurrency between transactions. Persistent functional languages provide a natural way for implementing and working with such functional databases, and allow the construction of functional transaction processing systems.

The traditional approach to the construction of transaction processing systems (TPS) is to use a database management system (DBMS) for storage, together with a (usually imperative) programming language for domain logic. A problem with this approach is that the mapping between the data model of the program and the data model of the DBMS is complex and unnatural, a problem known as the *impedance mismatch* [9]. Moreover, the sequential nature of the interface of the DBMS limits parallel execution of transactions, as well as limiting concurrency between transactions. Also, the connection between the DBMS and the program may be interrupted while there are unfinished transactions, which must be correctly handled both by the program and the DBMS. In addition,

in modern databases, transaction commits, rollbacks and security have to be managed explicitly, making correctness of a TPS difficult to establish.

In contrast, in our approach a database is implemented in a persistent functional language, and transactions are programs written in this same language and executed on the same system as where the database resides. The database and the programs use the same data model and computational model, thus avoiding the impedance mismatch. A transaction contains the complete program to be executed, and because of this, all queries are known when the transaction starts, allowing the execution of multiple queries in parallel. Furthermore, transactions are guaranteed to commit, avoiding the possibility of aborted transactions. This allows a higher level of concurrency than is possible in traditional DBMS's, as the system does not have to prevent access to modifications made by transactions that have not been fully evaluated. Furthermore, because concurrency is achieved through lazy evaluation of states, there is no risk of deadlock or starvation in transactions that update the state.

In the literature, a practical implementation of a persistent functional language and its application to functional databases have been given by Nikhil in 1990 [13], and McNally in 1993 [11]. However, these do not allow concurrent execution of transactions. This paper presents a prototype implementation of a concurrent persistent functional language, and we show how this language can be used to construct a transaction processing system. In addition, development of the prototype also led to further fine-tuning of the theoretical model of persistent functional languages. Concretely, this paper describes the following contributions:

- a language for the definition of functional transactions;
- a prototype implementation of our language that supports concurrent execution of transactions and storing states in persistent memory;
- a new method for load balancing in parallel graph reduction, based on sharing results between threads, and randomizing their reduction order;
- a discussion of different methods to store functional states in persistent memory, allowing both the storage of suspended computations, as well as larger than main-memory states;
- an improvement of the theoretical model of persistent functional languages, solving problems that arise in practise from the theoretical model due to lazy reduction of states; and
- an experimental validation of our results.

The remainder of this paper is organised as follows. First, Section 2 gives some relevant background on transaction processing systems and databases, as well as functional transaction processing systems, then Section 3 present a simple language for the definition of functional transactions. Next, Section 4 discusses how evaluation of transactions in this language is implemented, Section 5 presents our graph reduction method, Section 6 discusses several methods to store states in persistent memory, Section 7 discusses some issues in lazy evaluation of states, as well as a solution, and Section 8 presents our experimental results. Finally, Sections 9 and 10 present related work and conclusions.

2 Transaction Processing Systems and Databases

This section gives a quick overview of transaction processing systems, and functional transaction processing systems in particular. First, we discuss transaction processing systems in general, its desired properties and implementation challenges. After that, we discuss how functional languages can be used for transaction processing.

2.1 Transaction Processing Systems

A *transaction processing system* (TPS) is a system that manages concurrent operations by multiple users on a single *state* by means of *transactions* [7]. A *transaction* is a collection of operations on the state, that provides guarantees about its execution as a whole. For many transaction processing systems the *ACID* properties are guaranteed:

Atomicity: Either all operations in a transaction are executed, or none at all.
Consistency: Transactions preserve consistency of the state.
Isolation: The result of transactions executing in parallel is the same as the result for some sequential executions of the transaction.
Durability: Once a transaction has been committed, its effects must persist even in the case of system failure.

Typical examples of transaction processing systems are database management systems, transactional file systems, version control systems and transactional memory. This paper focuses in particular on database management systems as application area. However it should be stressed that our approach is also applicable to other kinds of TPSs.

Formally, atomicity and isolation are defined in terms of serializability and recoverability. A concurrent execution of a set of transactions is *serializable* if some sequential execution produces the same final state. A transaction is terminated by either committing the transaction, making its effects persistent, or aborting it and rolling back all changes. *Recoverability* states that, as long as a transaction t has not been comitted, all transactions that have read changes by t can not commit until t commits. If t chooses to abort, all transactions that have read changes by t must also abort.

Ensuring that the ACID properties hold results in several challenges for TPS implementations. First, one has to ensure that transactions can be correctly executed in parallel, avoiding inconsistent states that could be caused by interactions between transactions. Second, the TPS has to ensure correct behavior in case of system failure, as writes to persistent storage may only be partially complete at the time of failure.

Database Management Systems As mentioned above, a database management system (DBMS) is a particular TPS with the following additional characteristics: it has a data model, its state can be larger than main memory, and it has an

appropriate query language. A *data model* defines how information is structured in a database, together with the basic operations that can be performed on this data. A common example of a data model is the relational model, implemented in a relational database, where data is structured in the form of relations, and with basic operations such as fetch, insert, selection, projection, and join. To query and manipulate databases, a DBMS usually provides a *query language*, which is explicitly designed to work with the data model provided by the DBMS. For example, SQL is typically used as query language for relational databases.

2.2 Functional Transaction Processing

We now discuss how functional languages can be used in the construction of a TPS, by reviewing the work of Trinder [16]. A TPS can be considered a *transaction manager* function of type $State \times [Transaction] \to [Result]$ that takes an initial state and a stream of transactions, and produces a stream of results. A transaction is a function $State \to State \times Result$ that takes a state, and produces a new state together with an observable result. The transaction manager can be made available to many users by merging the transactions from each user into a single stream of transactions.

It is easy to guarantee the ACID properties in this model. Serializability is trivially satisfied, as transactions are executed serially. Recoverability can be satisfied by requiring that all transactions are *total* functions, i.e. they always produce a result. Consistency of a state `s` can be guaranteed by wrapping a transaction `f` by a function of the form `if g(f(s)) then f(s) else s`, where `g` validates that consistence holds in `f(s)`. Durability can be guaranteed by journalling transactions before executing them, as will be discussed in more detail later in this paper.

Transactions can be executed concurrently by reducing states lazily, forced by the reduction of the result expressions of the transactions. For this to have any effect, states must have a tree structure, such that large parts of the new state can be returned early for the next transaction to start. Concurrency control is essentially done through data dependency. This system can be implemented efficiently using graph reduction to share large parts of a new state with the previous state.

Finally, in order to support databases in this model, data can be stored in *bulk data structures* such as associative maps implemented by a binary tree. Queries that read and modify the database can be built from operations on these data structures.

3 A Transactional Functional Language

This section describes a language for the notation of functional transactions. We first discuss a model for a transactional functional language, followed by the description of our language, and we conclude with some examples. In the sections after this we discuss our prototype implementation of this language.

```
<transaction> ::= <definition>*
  <definition> ::= <variable> ( '(' <variable>* ')' )? '=' <expression>
 <expression> ::= LITERAL
                | ( <variable> | <constructor> ) ( '(' <expression>* ')' )?
                | 'match' <expression> '{' ( <pattern> '->' <expression> )+ '}'
                | 'let' ( <variable> '=' <expression> )* '{' <expression> '}'
    <pattern> ::= <constructor> ( '(' (<variable> | '_' )* ')' )?
    <variable> ::= 'a'..'z' ('a'..'z' | 'A'..'Z')* ''?
<constructor> ::= 'A'..'Z' ('a'..'z' | 'A'..'Z')*
```

Fig. 1. Grammar of prototype language.

We follow the model as described by Nikhil [12] to extend regular functional languages with constructs for transaction processing. The state of the system is a mapping from identifiers to closed expressions. Where expressions may contain functions as well as data. A transaction consists of an expression that is evaluated in the current state, and an environment that replaces the current state, which is a mapping from identifiers to expressions, where the expressions may contain free variables.

A transaction in our language describes only the updates to the state, instead of the complete new state. We distinguish two kinds of variables: *current state variables* and *next state variables*. To distinguish them syntactically, next state variables are primed. To specify an update to variable x, we assign an expression to variable x', where this expression can refer to both the current and the next state. We can also assign to current state variables: this means that the binding is local to the transaction, and will not be available in successive transactions. Finally, there is a special variable *result* which contains the observable result of the transaction.

Figure 1 presents the EBNF syntax of our language. A transaction consists of a list of definitions that map variables to expressions. Expressions are built from literals, function applications, case distinctions on data, and let expressions. In our prototype, literals can be integers, strings and doubles. Let expressions can be used to explicitly introduce sharing in expressions. Furthermore, our language also implements some built-in functions that are not part of the grammar, such as arithmetic functions and comparison functions on primitive data types.

We will now illustrate how our language can be used to set up and use a simple database of user names. This transaction updates the state to include a variable *users*, which is initialised to the empty list, and a function *length* that can be used to compute the length of a list:

```
1 users' = Nil
2 length'(list) = match list {
3   Nil -> 0
4   Cons(x xs) -> add(1 length'(xs))
5 }
```

Note that in the definition of *length′* we refer to *length′* to create a recursive function. If we would refer to *length* instead, we would refer to the value of *length* in the current state. The next transaction inserts a user into the database, and requests the size of the resulting database:

```
1 users' = Cons("bob" users)
2 result = length(users')
```

Note that in the assigned expression of *users′* we refer to *users* in the current state; thus inserting a user into the existing database. The observable result of the transaction is defined as the number of users in the database, including our newly inserted user 'bob'. Finally, the following transaction queries the database with a locally defined function *contains* that exists only for the duration of the transaction:

```
1 contains(value list) = match list {
2   Nil -> False
3   Cons(x xs) -> match equals(x value) {
4     True -> True
5     False -> contains(value xs) } }
6 result = contains("bob" users)
```

4 Implementation of the Prototype

This section discusses the implementation of a framework to support persistent functional languages. The prototype language discussed in the previous section is implemented on top of this framework; however the framework is set up in such a way that it can also support other persistent functional languages. Our prototype is implemented in Java, but code is presented as psuedo code.

Overview In order to support our language model, we need to modify the structure as discussed in section 2.2 a bit. In our language, a transaction is not a function, but an environment of updates together with a result expression. In order to execute a transaction, we *bind* a transaction to the state using a function of type $State \times Transaction \rightarrow State \times Expression$ that takes the transaction and the current state, and produces a new state together with a result expression. Also, in our implementation we combine the transaction manager with functionality to non-deterministically sequence transactions, instead of a separate merger process as described by Trinder.

The execution of transactions is done as follows. At the highest level, our system receives a transaction as a sentence in our language. First, we parse the sentence, to obtain a data structure representing the transaction. We then bind the transaction to the state, as is discussed in the next paragraph, to obtain a result expression. Finally, we reduce the result expression to normal form, and send the reduced result back to the sender of the request.

Data Structures First, we define data structures to represent states, and transactions.

```
1 data State       = environment : Map Identifier Node
2 data Transaction = updates : Map Identifier Node
3                  = locals : Map Identifier Node
```

A state is simply a mapping from identifiers to nodes. A transaction has two mappings: a mapping *updates* that stores the modifications to the state (where removal of a binding is encoded using as a null pointer expression), and a mapping *locals* that stores expressions that do not go into the new state, *i.e.*, local function definition, and functions obtained through λ-lifting. The mapping *locals* may also contains the special name *result* for the result expression.

Binding In order to execute transactions we need a binding function $State \times Transaction \rightarrow State \times Expression$ that takes a transaction and the current state, and produces a new state together with a result expression. We implement this function by binding free variables in the expressions of the transaction to the expressions in the state, and updating the state with the changes as specified by the transaction. A simple implementation of binding looks like this:

```
1 procedure bind(state, transaction) {
2   transaction' :=
3     bindFreeVariables(state.environment, transaction);
4   state.environment :=
5     bindUpdates(state.environment, transaction'.updates);
6   return transaction'.locals.get("result");
7 }
```

The `bindUpdates` procedure produces a new state according to the update environment of the transaction. The `bindFreeVariable` procedure, of type $Environment \times Transaction \rightarrow Transaction$ binds the free variables in the expressions in `transaction` to the values in the environment. That is, all pointers to free variables nodes in the expression graphs are replaced by pointers to the appropriate node. In our implementation, binding of free variables is done non-destructively to implement *stored transactions*, which is outside the scope of this paper. A complicating factor is that the variables have to be bound to already bound versions of expressions in the transaction, while the bound expressions themselves are still being constructed. To facilitate this, copies of the root nodes of the expressions are created before actually binding the expressions. As some of these copied root nodes may be free variables, we first have to resolve these before binding the other expressions. After this is done, the copied root nodes are updated to bind the free variables in their child nodes. For the full algorithm we refer to the Masters Thesis by Wevers [17].

Handling Concurrent Requests In order to handle concurrent requests, we create a new request handler for every request, and execute them in parallel using threads. All requests handlers share the same state, which they update through the bind procedure. We have to ensure that updates to the state do not interfere.

The simplest method to ensure correct updating of the state is to make access to the state mutually exclusive using a lock. However, as read transactions do not modify the state, these may be bound concurrently, so we could use a readers / writers lock. We can do even better by making updates to the state non-destructive, allowing lockless reads, while we can synchronize writes to make updates to the state mutually exclusive. The following algorithm shows how we implemented this.

```
1  procedure bind(state, transaction) {
2    if(transaction.updates.isEmpty()) {
3      transaction' :=
4        bindFreeVariables(state.environment, transaction);
5    } else {
6      synchronize(state) {
7        transaction' :=
8          bindFreeVariables(state.environment, transaction);
9        state.environment :=
10         bindUpdates(state.environment, transaction'.updates);
11     }
12   }
13   return transaction'.locals.get("result");
14 }
```

Updating the state is now an atomic operation that creates a new environment for every update, this ensures that read transactions see a consistent state. Below, in Section 8 we evaluate the throughput performance of our implementation.

5 Parallel Graph Reduction

Aside from the transaction manager as described in the previous section, our implementation also need a parallel graph reducer. In this section we describe our method for parallel graph reduction. The main challenge in parallel graph reduction is load balancing of tasks among a fixed set of execution threads, while not using too many resources for performing the load balancing [15]. We show a method for load balancing based on result sharing between execution threads, and randomization of execution paths of threads.

Standard Graph Reduction Our graph reducer is based on *template instantiation* [14]. We have defined data structures to represent different kinds of nodes as shown in Figure 2. We have also defined a special *sharing node*, which we discuss in the next paragraph. To implement lazy evaluation, we define a procedure `whnf` that reduces a graph to weak head normal form (WHNF) by repeatedly reducing the outermost reducible node until either a primitive data node, a data node or a supercombinator node is obtained. This reduction is done non-destructively, with the exception of sharing nodes as is discussed below. For some nodes, `whnf`

```
1 data Node = SupercombinatorNode(template : Node)
2           | SupercombinatorApplication(arguments : [Node],
                supercombinator : Node)
3           | BoundVariableNode(index : Integer)
4           | DataNode(constructor : int, children : [Node])
5           | CaseNode(scrutinize : Node, cases : [Node])
6           | IntNode(value : int)
7           | AddNode(left : Node, right : Node)
8           | SharingNode(shared : Node)
9           | ...
```

Fig. 2. Data structures for graph reduction.

may need to call itself tail-recursively to obtain a WHNF of an intermediate result. We define a procedure `reduce` that performs a minimal reduction step towards WHNF, which is obtained from taking the implementation of `whnf` and not performing a tail-recursive call to `whnf`. Furthermore, we define a procedure `isWhnf` that tests if a certain node is in weak head normal form, by checking if it is a primitive data node, a data node, or a supercombinator.

Parallel Graph Reduction Our approach to parallel graph reduction is based on sharing results of work between threads, and randomizing their reduction order. The idea of randomization and result sharing is not new, and has already been applied successfully in the context of model checking for the parallel exploration of a state-space [6], however to our knowledge this method has not yet been applied to graph reduction.

The main idea in our approach is that we consider a thread as taking a walk through the graph while reducing nodes on the way. Instead of having all threads taking the same path, we try to make each thread take a different path by taking different decisions on which child node to reduce first. Work is shared between threads through special result sharing nodes. If a thread encounters such a node, it may find that another thread has already performed the work, and it can use that result. Below we present our result sharing algorithm, as well as a method for randomization of execution paths.

Sharing Results The first part of our load distribution algorithm is the sharing of results between threads. We share results through a special result sharing node that has a pointer to either an unreduced expression, or its reduced form. When a thread computes a reduced form of a shared node, it may update the sharing node to point to the new result as to allow another thread to use that result as well. The `whnf` procedure may need to tail-recursively call itself for some nodes in order to obtain a WHNF, in order to share results in such a case, we instead use our `reduce` procedure. Furthermore, in order for sharing to be done correctly, we have to ensure that each thread uses the same unique result, as to avoid loss of sharing due to data races. The following algorithm allows the reduction of a sharing node node:

```
1 procedure whnf(node : SharingNode) {
2   Node current := node.shared;
3   while(not isWhnf(current)) {
4     Node result := reduce(current);
5     if(compareAndSet(node.shared, current, result)) {
6       current := result;
7     } else {
8       current := node.shared; }
9   return current; }
```

This procedure first fetches the current shared node from the sharing node. If it is in weak head normal form, it returns immediately. If the shared node is not in weak head normal form it is reduced using `reduce`, which result is then stored in the sharing node, except if the pointer in the sharing node has been modified by another thread. If the sharing node has not been modified, this will succeed, and the algorithm continues until a weak head normal form is obtained. If the sharing node has been modified, another thread has already reduced the shared node, in this case we continue with the result of the other thread and discard our own result. The uniqueness of the result is thus guaranteed by this algorithm. Also, this algorithm is guaranteed to terminate if reduction of the shared node terminates, because there must always be one thread that succeeds in reducing the shared node by one step, after which all other threads will fail to write their result, and continue with the new result.

Randomizing Execution Paths The second part of our load balancing method is the randomization of the execution paths of each thread. The path a thread takes is determined by the order in which it reduces the child nodes of some node. In standard graph reduction, child nodes are usually reduced in a fixed order. In order to randomize the execution paths of threads we have to choose a different reduction order of child nodes for each thread.

In our implementation, we do not actually randomize the order of threads, but we alternate orders. To implement this, we maintain a boolean in primitive functions that require eager reduction of their arguments, indicating whether to reduce the arguments from left to right, or right to left. Every time a thread wants to reduce this node, the boolean is flipped, so that threads alternate between reducing child nodes from left to right, or from right to left. As an example, consider the evaluation of the addition node:

```
1 Node whnf(AddNode add) {
2   add.left_to_right = !add.left_to_right;
3   if(add.left_to_right) {
4     l := whnf(add.left); r = whnf(add.right);
5   } else {
6     r := whnf(add.right); l = whnf(Add.left); }
7   return Int(l.value + r.value); }
```

Here, we modified the data structure of `add` to include a field `left_to_right`. Every time a thread starts reducing this node, the `left_to_right` field is negated. The first thread will now reduce from left to right, the second from right to left, the third from left to right again, etc. Below, in Section 8 we discuss some experimental results that show that this approach to load balancing in parallel graph reduction indeed results in a significant speedup.

6 Maintaining Persistent State

The implementation so far stores the whole state in main memory. In this section we investigate different ways to efficiently store the state in a persistent heap, with the goal of guaranteeing durability in case of a system failure, storing results of reduction, as well as to support states which do not fit in main-memory. First we discuss journalling to support durability. We then discuss snapshotting as a method for persisting evaluated states. Next, we discuss an alternative approach to persisting states, based on log-structured storage. Finally, we describe a way to combine both of these approaches to combine their strenghts.

Journalling for Durability A standard method in databases for guaranteeing durability is journalling [7]. Using journalling, before a transaction is executed, it is first written to a log in persistent memory. If the system crashes and starts up again, it recovers by re-executing all transactions in the log to obtain the same state as prior to the crash. To guarantee durability to the user, the system must ensure that a transaction log is actually stored in persistent memory before confirming the execution of the transaction to the client.

In theory, having an initial state and a journal starting in this initial state is enough to reconstruct the state at any point in time. However, in practice this is not really sufficient: the log grows beyond bounds as entries can never be removed, and moreover it is extremely inefficient to re-execute all transactions when a journal grows large. For this reason, we want to store reduced forms of states as a *checkpoint*, so that the system only has to recover from the last checkpoint. We now discuss some methods for creating checkpoints.

Snapshotting A simple method for storing reduction results is *snapshotting*, where we serialize the state and write this to persistent memory. However, a complication here is that snapshotting a state containing suspended computations, while concurrently reducing the state, can lead to a loss of sharing in computation as well as data in the snapshot. A simple solution to solve this problem is to reduce the state to normal form before snapshotting it, which is also the approach that we have implemented in our prototype. However, the drawback of this approach is that snapshotting could be delayed for a long time while a large update is being performed (and thus the journal can grow very large, leading to long recovery times if the system crashes before the snapshot is finished).

We now want to make sure that a snapshot is consistent: *i.e.*, to make it look as if the snapshot is created while no reduction was going on. For this we observe

that the only nodes reachable from the root that change during reduction are the sharing nodes. Therefore, we remember the original node referenced by a sharing node when it is reduced during the creation of a snapshot, and we snapshot the original node instead of its reduced version. During snapshotting we can clean up the references to the original nodes. One complication here is that using this scheme, there is no garbage collection during snapshotting, to solve this we only remember the original reference for sharing nodes that have been constructed before snapshotting starts.

Advantages of this approach are that snapshots can contain suspended computations, allowing snapshotting of long running computations, and that sharing is maintained in snapshots.

The biggest drawback of snapshotting is that we can not support states that are larger than main-memory. Another drawback is that snapshotting can take considerable time to complete. Moreover, large parts of the state might not have changed between the snapshots. If there is a high load on the system, we might have to snapshot quite often to keep the journals small enough for quick recovery times. As the size of the state grows, taking snapshots takes a longer time, and the journals grow larger between every subsequent snapshot.

Log-Structured Storage An alternative approach to store data is the use of *log-structured storage* [8], also known as append-only storage. The main idea is that the nodes in the state are stored as records in a log file, and records can point to other records by their position in the file to encode graph edges. After execution of a transaction, all new nodes in the state are *appended* to the log, followed by a special root record that encodes the state environment after the transaction. This model fits the functional model well, as it is inherently non-destructive.

Advantage of this approach is that it does not overwrite any data, thus if the system crashes while appending records to a log, there is no risk of data loss. State recovery is simply a matter of finding the last correctly written root record from the end of the log file. One can consider this approach as a continuous snapshot of the system state in a single file. This has the advantage that we always have an up-to-date snapshot of the system by appending only the changes since the last snapshot, instead of writing the whole state every time. Further, states that are larger than main-memory can be supported by dynamically loading data from persistent memory, and only caching data in main memory.

A major complication of this approach is that the log only grows, and never shrinks in size. Further, the layout of the log becomes sub-optimal for reading on persistent media with high access latency, due to many I/O operations needed. Garbage collection can be used to periodically clean up the log, and ensure locality of reference to improve read performance.

Further, log-structured storage is inefficient for storing suspended computations. If we store a suspension, and later compute its result, we have to somehow overwrite the suspension with the result such that the result is stored. We can not actually overwrite the result in the log destructively, because if the system crashes while updating a record, the log may be corrupted due to incomplete written records. Instead, we have to write every path from the root to the result.

There might be many paths to the result, and as we cannot see locally which expressions reference the suspended computation, it can be expensive to find all these paths. Furthermore, writing all these paths can be quite expensive in terms of both I/O operations, as well as space requirements.

Mixed Approach We plan to further investigate a combined approach, taking advantage of the strength both approaches, while minimizing their drawbacks. The idea is that we split the heap into an *active heap* which may contain reducible expressions, and a *passive heap* which must be in normal form. For the active heap we use snapshotting, while for the passive heap we use log-structured storage. However, instead of using special root records in the passive heap, the active heap contains the root of the graph, and the passive heap only contains data.

7 Forcing Evaluation of Transactions

During experimentation with our prototype, we found some problems with the theoretical model in practice. One problem is that we may get long chains of lazily evaluated *thunks* (suspended computations) in states as a result of a chain of transactions that update the state, but that do not read the state. In practise, this can lead to a stack overflow when reading after many updates. Another problem is that thunk leaks may build up in the state. An example of the latter is that a `map` function applied to a binary tree may leave thunks applied to `Leaf` nodes, of which result may never be demanded. Another related problem is that sharing nodes are created in the state during evaluation of result expressions, but these are never garbage collected.

Our solution to these problems is to force the reduction of states to normal form. This forces the reduction of all thunks, as well as to clean up sharing nodes. Also, we limit the number of active transactions to avoid long chains of thunks. In order to implement this, we need to keep track of which parts of the state have already been reduced to normal form, in order to avoid duplicate reduction of the state every time its reduction is forced. This is implemented in our prototype by maintaining a flag on `Data` nodes that indicates whether it is in normal form.

We found that simply forcing the reduction of states to normal form does not work well for limiting the number of active transactions in a concurrent setting. The problem is that if a small transaction is preceded by a very large transaction, and we force the evaluation of the state created by the small transaction, this also forces the reduction of the state produced by the large transaction. Because of that, we can not know accurately when the reduction of the small transaction is done, which is needed to count the number of active transactions.

A theoretical solution to this problem is to reduce only those parts of the state that a transaction modifies. To implement this, for all reducible nodes, we need to keep track of which transaction it belongs to. In order to force the evaluation of a transaction, we need to force the reduction of all reducible nodes that belong

to that transaction. We plan to further investigate the implementation of this solution.

8 Evaluation

Using our prototype implementation, we have performed several experiments to asses the performance of our parallel graph reduction method, and the performance of the prototype implementation for transaction processing. Given the limitations of our current implementation, we have performed the experiments with artificial, and relatively small benchmarks[1]. All experiments have been run on a database which resides fully in main memory, so that there is no overhead from I/O operations.

Parallel Graph Reduction To evaluate our load balancing method for parallel graph reduction, we measure the relative speedup that we can obtain for two parallel algorithms: *nfib*, which naively computes the nth Fibonacci number, and *treesize*, which computes the size of a binary tree. The latter is heavily data dependent, whereas *nfib* is not. For both algorithm, we have implemented two variants: one in our prototype language, and one as a native function implemented in Java to simulate performance of a compiled language.

We have run two different sets of benchmarks. First, we measured the relative speedup by executing the algorithm with 1 up to 48 threads, and dividing the measured execution time with the execution time when using a single thread. Second, we measured the overhead of our parallel graph reducer compared to a serial graph reducer (where the serial graph reducer is obtained from our parallel graph reducer by disabling the randomizer, as well as replacing concurrent result sharing by a non-concurrent variant). We noticed increasing amounts of variation in execution times as we used more threads, the results shown here are the median of several measurements.

Figure 3 shows the relative speedup that we have measured. The dashed line shows the ideal speedup that can be achieved, assuming a linear speedup as the number of threads increases. The vertical lines show the boundaries of the NUMA nodes of our testing system. We see that the relative speedup for both algorithms is nearly ideal when all threads are able to run on a single NUMA node. When more than one NUMA node is used, the increase in speedup suddenly drops for the treesize benchmark. We suspect that this happens because the threads have to access memory on another NUMA node when scaling beyond one node, which takes longer than accessing memory locally.

Figure 4 shows concrete execution times obtained from our serial graph reducer and our parallel graph reducer, and the overhead of parallel graph reduction compared to serial graph reduction.

[1] All experiments are run on quad AMD Opteron 6168 system with 48 cores divided over 4 processors, using scientific linux, running Oracle HotSpot JVM version 1.7.0 build 147.

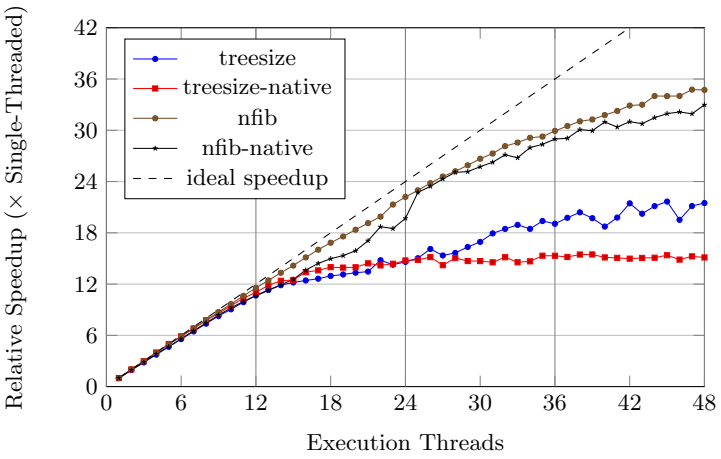


Fig. 3. Relative speedup of our parallel graph reducer.

	treesize	treesize-native	nfib	nfib-native
Serial	2666 ms	819 ms	3294 ms	626 ms
Parallel	3243 ms	1291 ms	4162 ms	819 ms
Overhead	21.6%	57.6%	26.4%	30.1%

Fig. 4. Running time of parallel reduction compared to serial reduction.

Concurrency In the next set of benchmarks, we evaluate the behaviour of our prototype when executing transactions concurrently. First, we show that transactions are actually executed concurrently, next we investigate the effect of concurrent transactions on transaction throughput, and finally we investigate the effect of concurrent transactions on memory usage. All benchmarks are run on a state that is initialized with a map, implemented as a binary search tree, from keys $0, \ldots, 100,000$ to the value 0. Using only a single thread, we update this binary tree to increment all values by one. We then read individual values at random from the map. The new state is constructed lazily, meaning that only those parts of the state are evaluated that are actually read.

First, we show that transactions are actually executed concurrently. We measured that updating the state took 89 μs, reading a single value from this new state took 97 μs, and then forcing the full evaluation of the state took 1405 ms. This effectively shows that the read transaction must have been executed concurrently with the update transaction.

Figure 5 shows the effect of a number of updates on the throughput of read transactions. It can be seen that throughput drops significantly in the first moments after an update, but recovers as a larger part of the state has been evaluated. In this same benchmark, we found that memory usage increased during

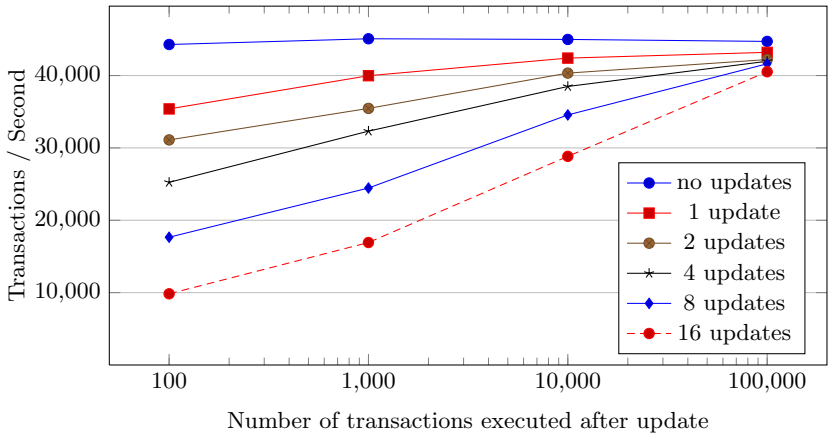


Fig. 5. Influence of updates on transaction throughput for a map of size 10,000.

the first 10,000 transactions, after which memory usage remained constant. Inspection of the heap showed that the `map` functions are pushed into the leafs of the tree and these are never evaluated, also sharing nodes build up in the state during evaluation, as described in Section sec:forced.

Transaction Throughput In our final set of benchmarks, we investigate the transaction throughput of our prototype. To do this, we use the same setup as for the concurrency benchmarks, but we use multiple threads to issue transactions against the system.

We found that reads scaled very well, from 36,000 transactions per second with one thread up to 1,109,000 transactions per second with 48 threads. Updates scaled much worse, from 15,857 transactions per second using one thread, to 93,139 transactions per seconds with 8 threads, and back to 49,714 transactions per second using 48 threads. We found that synchronization is the bottleneck here due to multiple NUMA nodes contending for a single lock, leading to a lot of communication overhead. This has been validated by running the benchmark with a limited number of cores, where no slowdown is observed after the peak performance has been reached, and where the sustained throughput is higher than when using all cores. A NUMA aware lock might solve this problem, but we could not test this, as we did not have an implementation of this available in Java.

9 Related Work

Asside from the work by Trinder [16], McNally [11] and Nikhil [12] on which we build, other work has been done that is closely related to our work. In this section we compare our work to this work, and discuss how our approaches differ.

Persistent languages The first efforts to integrate transparent persistence with programming languages was the language PS-Algol [3]. More recent efforts to integrate programming with persistence include Persistent Haskell [5] and Persistent Java [10]. In these approaches, objects can be marked as persistent roots, and the system automatically ensures that any object reachable from these roots is stored in persistent memory. One of the main differences from our model is that this model is not inheritly transactional.

Approaches such as ACID State for Haskell [1] and Prevayler for Java [2] extend this model of persistence by introducing transactions. The main difference from our model is that in these approaches the interface to the data is part of the program, while in our model it is part of the system. This makes interoperation with other systems in these approaches more flexible, but it does not allow ad-hoc queries which makes changing the schema of stored data difficult.

Persisting states in persistent memory Earlier work on persistent functional languages used a generic persistent object-store [4] to manage storage of states in persistent memory, since the creation of a persistent store is a complex task in itself. We have instead created our own methods for storing states in persistent memory, with a specific focus on the functional data model.

Parallel Graph Reduction A common approach for load balancing in parallel graph reduction is *work stealing* [15]. In this approach, if an execution thread does not have any more work to perform, it may 'steal' a task from the work pool of another execution thread. The main problem in work stealing is that bookkeeping of tasks is relatively expensive, and if too many tasks are sparked, parallel performance may be worse than sequential performance. To solve this problem, functional programs are annotated to explicitly define where to spark tasks. However, it requires a lot of effort from the user to find the optimal places in the program to maximize performance. Interestingly, in our example programs using our load balancing method for parallel graph reduction, we did not have to provide any annotations to obtain good relative speedup results. However, we do not yet know if this also holds for more complex examples.

10 Conclusions

This paper discusses the prototype implementation of a transactional functional language, that can be used for constructing a transaction processing system based on functional databases.

We have designed a language for describing transactions in our prototype, as well as allowing the definition of stored transactions. We have described how this language can be implemented through a process called binding, and we have shown a method for executing transactions concurrently in our framework.

Further, the implementation contains a new method for load balancing in parallel graph reduction, based on sharing results between threads, and randomizing their reduction order. Also, we discussed different approaches to store

functional states in persistent memory, allowing both the storage of suspended computations, as well as supporting states that are larger than main-memory.

Besides the prototype implementation, we also discussed some problems with the theoretical model that arise in practise. We have outlined a solution to solve this problem, by forcing the evaluation of transactions.

The experimental validation of our results shows that our load balancing method for parallel graph reduction resulted in a good relative speedups, and may provide an interesting alternative to the work-stealing approach of load balancing. We also investigated the behaviour of our prototype for transaction processing applications, where we investigated both its behaviour under concurrent evaluation of transactions as well as their throughput.

Future Work The work described in this paper is only the first step of a much larger project. There are many ways in which this work can (and will) be continued.

First, we will further investigate our prototype persistent functional language. Currently it is untyped, and we will explore how it can be typed correctly and efficiently. It would also be interesting to explore if this system can be integrated with programs written in existing programming languages, such that the state can be queried in an ad-hoc manner using our language, while flexible interfaces can be created using traditional programming languages.

When developing examples, we realised that algorithms and data structures that provide transactional concurrency under lazy evaluation have not yet been investigated in the literature. In particular, a concurrent functional balanced search tree is required for the efficient implementation of indices in functional databases. We plan to define a theory of concurrency in functional transactions, and investigate the development of algorithms in this setting, as well as the possibility of automatically optimizing expressions to maximize concurrent evaluation.

Concerning the implementation, we will further investigate our method for load balancing in parallel graph rewriting, and compare and combine it with work stealing approaches. We will also investigate other methods for the randomization of execution paths. We also plan to further investigate the forced evaluation of transactions. We have a theoretical solution (as explained above) that we will implement, and evaluate its usability in practice. We will also further investigate the different storage methods, and implement the mixed approach as outlined at the end of Section 6.

In the long term, we also plan to investigate the distribution of persistent functional languages among multiple computer systems for fault tolerance and to increase performance.

References

1. Acid state manual: http://happstack.com/docs/crashcourse/acidstate.html.
2. Prevayler website: http://prevayler.org/.

3. M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. PS-algol: A Language for Persistent Programming. In *10th Australian National Computer Conference, Melbourne, Australia*, pages 70–79, 1983.
4. A.L. Brown. Persistent object stores. Technical report, Software Engineering Journal, 1988.
5. T. Davie, K. Hammond, and J. Quintela. Efficient persistent Haskell, 1998.
6. M.B. Dwyer, S. Elbaum, S. Person, and R. Pur. Parallel randomized state-space search. In *in International Conference on Software Engineering*, pages 3–12.
7. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.
8. D. Hulse and A. Dearle. A log-structured persistent store. In *In Proceedings of the 19th Australasian Computer Science Conference*, pages 563–572, 1996.
9. C. Ireland, D. Bowers, M. Newton, and K. Waugh. A classification of object-relational impedance mismatch. In *Proceedings of the 2009 First International Conference on Advances in Databases, Knowledge, and Data Applications*, DBKDA '09, pages 36–43. IEEE Computer Society, 2009.
10. A. Marquez, S.M. Blackburn, G. Mercer, and J. Zigman. Implementing orthogonally persistent Java. In *In Persistent Object Systems*, pages 218–232, 2000.
11. D. McNally. *Models for Persistence in Lazy Functional Programming Systems*. PhD thesis, University of St Andrews, 1993.
12. R.S. Nikhil. Functional databases, functional languages. In *Data Types and Persistence, Proc. of the First Workshop on Persistent Objects*, pages 299–313, 1985.
13. R.S. Nikhil. The semantics of update in a functional database programming language. In François Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages, Papers from DBPL-1, September 1987, Roscoff, France*, pages 403–421. ACM Press / Addison-Wesley, 1987.
14. S. Peyton Jones and D. Lester. *Implementing functional languages: a tutorial*. Prentice Hall, 1992.
15. M.A. Rainey. *Effective scheduling techniques for high-level parallel programming languages*. PhD thesis, 2010.
16. P. Trinder. A functional database. Technical report, 1989.
17. L. Wevers. A persistent functional language for concurrent transaction processing. Master's thesis, University of Twente. Expected August 2012.

Detecting Process Relationships in Erlang Programs

Melinda Tóth, István Bozó

Eötvös Loránd University, Budapest, Hungary
{tothmelinda,bozoistvan}@caesar.elte.hu

Abstract. Static software analyser tools can help in program comprehension by detecting relations among program parts. Detecting relations among the concurrent program parts, e.g. relations between processes, is not straightforward. In case of dynamic languages only a (good) approximation of the real dependency can calculated. In this paper we present algorithms to built a process relation graph for Erlang programs. The graph contains direct relation through message passing and hidden relations represented by the ETS tables.

1 Introduction

Erlang [7] is a dynamically typed concurrent programming language. Erlang was designed to develop highly concurrent, distributed, fault tolerant systems with soft real-time characteristics. The dynamic and concurrent features of the language makes static analysis hard, however the statically calculated information and the built abstract representations from the source code can help the developers in different phases of the software development lifecycle. The static analysis techniques can help in debugging and maintenance tasks, or in program comprehension.

The main goal of the RefactorErl project [4] is to support program comprehension for Erlang developers in numerous ways. It gives a semantic query language to query information about the source. The tool can generate call graphs with dynamic call information, and calculates side-effect analysis based on it. RefactorErl provides a platform for module and function restructuring, the clustering. It can generate function or module dependency graphs. The tool has several interfaces, e.g it built into Emacs, it can be used from the Erlang shell, and it provides a web interface for multi user usage.

Our goal is to extend the current functionality of the tool and implement process relation analysis. We represent the processes as graph nodes and add the relation as edges to the graph. In this paper we describe relations trough message passing and through ETS tables. The latter one represent hidden relations among processes, and the algorithms presented in this paper can be adopted to other types of hidden relations (such as relation through files, and database usage).

The paper is structured as follows. In Section 2, we introduce the tool RefactorErl. In Section 3, we present the basic concurrent language construct of Erlang. In Section 4, we introduce a small client-server example. In Section 5, we

describe a representation of process relations and give algorithms to detect them. Section 6 applies the presented algorithms to the motivating example Section 7 discusses related work and Section 8 concludes the paper.

2 Background – RefactorErl

RefactorErl [9, 5] is a static source code analyser and transformer tool for Erlang. The tool represents the source code in a Semantic Program Graph (SPG). This graph stores lexical, syntactic and semantic information about the source code. RefactorErl has as asynchronous semantic analyser framework and implements thorough static semantic analysis based on this framework: variable, function, module, record analysis, call graph and dynamic call analysis, side-effect analysis, dependency analysis, data-flow analysis, etc.

Since RefactorErl provides a great platform for further analysis, we build our process relation analysis based on it. Both syntactic (such as collecting message passing expressions) and semantic (such the possible values of an expression by data-flow reaching) information can be gathered efficiently from the SPG using its query language. Therefore, the necessary syntactic information collection part of the algorithm can be transformed to queries during our analysis.

3 Examined Language Constructs

In this section we describe the subset of language constructs, that are relevant to describe our analysis methods. As we described in the introduction, our analysis is focused on communication of parallel processes. We cover only the language constructs and expression used in our paper. The reader can find more detailed description on this topic in the documentation of the language [8].

3.1 Process creation

Every process in Erlang is identified with a process identifier (`pid`). The process identifier is unique for every process. The function `self/0` returns the process identifier of the executing process.

In Erlang processes can be created with any of the functions `spawn, spawn_link, spawn_monitor, spawn_opt` etc. These spawning functions have similar behaviour, thus in this paper we describe only the basic functions `spawn/3` and `spawn_link/3`.

Function spawn/3 The function `spawn(Mod, Fun, Args)` creates a new process running the given function application `Mod:Fun(Arg1, Arg2,..., ArgN)` and returns the `pid` of the created process (where $Args = [Arg1, Arg2, ..., ArgN]$). The newly created process is placed in the system scheduler queue.

If the given function does not exist, a `pid` is returned and the error is reported.

Function spawn_link/3 The function `spawn_link(Mod, Fun, Args)` behaves similarly as the function `spawn/3`, it creates a new process and returns its `pid`. The difference is in linking the calling process and the created process.

Two processes can be linked with the function `link/1` and unlinked with the `unlink/1`. Terminating process emits the exit signal to every linked process. The default behaviour is that if the exit signal is other than normal, it causes the linked processes to terminate. If the `process_flag(trap_exit, true)` option is enabled in the process, it will receive the exit signals from linked processes as standard messages.

The function `spawn_link/3` spawns and links the created process to caller in one atomic operation.

3.2 Processes Registration

Processes can be registered with a given name (atom expression) with the built in function `register(Name, Pid)`. After registration the process can be addressed with its `pid` or with its registered name. The process can be unregistered with the built in function `unregister(Name)`. The registered process is automatically unregistered when the process terminates.

3.3 Communication

Processes communicate by message sending and receiving. There are different language constructs for sending and receiving messages.

Sending Messages Messages can be send with message sending operator (`!`) or with functions `erlang:send/2, erlang:send/3, erlang:send_after/3` etc.

The expression `Expr1 ! Expr2` sends a message `Expr2` to `Expr1` asynchronously, where `Expr1` must evaluate to a `pid` or an atom (registered name).

If `Expr1` evaluates to:

- `pid` – the message sending always succeed, even if the process does not exist;
- `atom` – the message sending succeeds only if a registered process exists with the given name, otherwise a run-time error is raised.

Receiving Messages Messages can be received with the `receive` construct.

The skeleton of the receive construct is showed in Fig. 1. The expressions enclosed in brackets (`[ ]`) are optional.

The receive expression suspends the execution of the executing process until a message is received. The received messages are matched sequentially against the given patterns. The body of the first matching branch where the guard expression evaluates to `true` is executed. The return value of the receive expression will be the result of the evaluated body. If none of the patterns match, or none of the guard expressions evaluate to `true`, a new message is extracted from the message queue. The after branch is evaluated if none of the received messages mach the given patterns (and/or guards) in the given time interval `MilliSec` (milliseconds).

```
receive
    Pattern1 [when Guard1] -> Body1;
    ...
    PatternN [when GuardN] -> BodyN
[after
    MilliSec -> AfterBody]
end
```

Fig. 1. Receive construct

3.4 Erlang Term Storage (ETS)

The ETS tables provide possibility to store large amount of data in the Erlang run-time system and constant access time to this data. The data is stored in dynamic tables as tuples. The table is created by a process, when the process terminates the created ETS table is destroyed.

Creating New Tables A new ETS table can be created with the function `ets:new(Name, Options)`. The function returns a table identifier, that can be used in accessing the table. The first argument is the name of the table (an atom), the second argument is a list of the options. The used options in our example are:

- `named_table` – The table can be accessed with either its name and table identifier;
- `public` – The table can be accessible for every process, either for reading or writing.

The ETS table can be initialised as a `set`, `ordered_set`, `bag` or `dupliacate_bag`. The default value is the `set` option.

Writing the Table New entities can be inserted to the given table with the function `ets:insert(Tab, Data)`. The `Tab` must be evaluated to a table identifier or an atom if the table is named. The `Data` is either a tuple or a list of tuple expressions.

Reading the Table In our paper we use two different table reading methods. The simplest is the `ets:match(Table, Pattern)` that matches the elements in the table against the given pattern and returns the list of matching objects. A more sophisticated function for data extraction is the `ets:select(Table, MatchSpec)`. In the MatchSpec we provide a list of match functions. Match functions are three tuples, where the first element is a pattern, the second is a list of guards and the third element is the description of elements of the result list.

4 Motivating Example – Job Server

We will use a simple client-server example to illustrate our model. The source code of the server and client can be found in Fig. 2 and 3.

The server module provide interface functions for starting (`start/0`) and stoping (`stop/0`) the server process.

The server module also provides interface functions for the client application:

- `connect/1` – Connects the given client (`Cli`) to the server;
- `disconnect/1` – Disconnects the given client (`Cli`) from the server;
- `do/3` – Asks the server to execute the given function (`Fun`) from module (`Mod`) on the given table (`Tab`).

The server exports the callback functions that initiates the server process (`init/1`) and the iterating function (`loop/1`) that receives messages and performs the asked tasks. The function `loop(State)` stores the connected clients in the server state variable (`State`). If receives the message `stop` then terminates. If receives the {`connect, Cli`} message, then updates the server state with adding the new client to the list. If it receives the {`disconnect, Cli`} message, then updates the server state with removing the client from the list. If it receives the message {`do, Mod, Fun, Tab`}, it extracts the necessary data from the provided table, executes the given function and writes the result to the table and calls itself recursively.

The client module provides an interface function (`start/1`) to start the client application. The function connects to the server process and creates a named, public `ets` table (`data`). In the next step it spawns a new input reader process and starts to execute the function `loop/2`. The function `input/1` reads commands iteratively from the input and sends these commands to the parent process. If it reads the atom `quit`, it stops reading input.

The function `loop/2` receives messages from the function `input/1` and forwards the jobs to the server. If it receives the atom `quit` from the input it disconnects from the server and prints the results.

There are several processes in our example. There is a client process to communicate with the server, a client process for input reading, a server process, and processes to start and stop the server. We note here that the last two operations can be performed from the same process. Our goal is to present a model to represent these processes and the relations among them.

```
1  -module(server).
2
3  -export([connect/1, do/3, disconnect/1]). %% Client interface
4  -export([start/0, stop/0]). %% Server interface
5  -export([init/0, loop/1]). %% Server callbacks
6
7  -define(Name, job_server).
8
9  %%%%%%%% Client interface %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
10 connect(Cli) ->
11     ?Name ! {connect, Cli}.
12
13 disconnect(Cli) ->
14     ?Name ! {disconnect, Cli}.
15
16 do(Mod, Fun, Tab)->
17     ?Name ! {do, Mod, Fun, Tab}.
18
19 %%%%%%%% Server interface %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
20 start() ->
21     register(?Name, spawn_link(?MODULE, init, [])).
22
23 stop() ->
24     ?Name ! stop.
25
26 %%%%%%%% Server implementation %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
27 init()->
28     process_flag(trap_exit, true),
29     ?MODULE:loop([]).
30
31 loop(State)->
32     receive
33         stop ->
34             ok;
35         {connect, Cli} ->
36             ?MODULE:loop([Cli|State]);
37         {disconnect, Cli} ->
38             ?MODULE:loop(lists:filter(fun(A) ->
39                                           A /= Cli
40                                       end, State));
41         {do, Mod, Fun, Tab} ->
42             handle_job(Mod, Fun, Tab),
43             ?MODULE:loop(State)
44     end.
45
46 handle_job(Mod, Fun, Tab) ->
47     Data = ets:select(Tab, [{{'$1','$2'},
48                              [{'/=', '$1', result}],
49                              ['$$']}]),
50     Result = Mod:Fun(Data),
51     ets:insert(Tab, {result, Result}).
```

Fig. 2. Job server skeleton code

```
1  -module(client).
2  -export([start/1, input/1]).
3
4  start(Client) ->
5      server:connect(Client),
6      ets:new(data, [named_table, public]),
7      spawn(?MODULE, input, [self()]),
8      loop(data, Client).
9
10 loop(Tab, Name) ->
11     receive
12         quit ->
13             server:disconnect(Name),
14             io:format("~p~n", [ets:match(Tab, {result, '$1'})]);
15         {job, {Mod, Fun}} ->
16             server:do(Mod, Fun, Tab),
17             loop(Tab, Name)
18     end.
19
20 input(Loop) ->
21     case read_input() of
22         quit ->
23             Loop ! quit,
24             ok;
25         Job ->
26             Loop ! {job, Job},
27             input(Loop)
28     end.
29
30 read_input() ->
31     [ets:insert(data, Data) || Data <- init_data()],
32     returns_the_job_to_be_executed().
```

Fig. 3. Client skeleton code

5 Representing Process Relationships

Based on our case study we will illustrate how we detect and build the communication model of Erlang programs to represent the relationships among processes. In this paper we focus on two type of relationships: relationships through message passing and hidden dependencies through ETS tables.

We represent the process relationships in a labelled graph ($G = (V, E)$) that describes the communication of Erlang processes. The vertexes ($v \in V$) of the graph are the processes. We use the `ModuleName:FunctionName/Arity` triple to identify the process p, and if p is registered we also use its name. The labelled edges of the graph ($e \in E$) represent:

- process creation ($\{spawn, spawn\_link\}$),
- process name registration ($register$),
- message passing (labelled with a tuple containing the sent message, $\{send, Message\}$),
- ETS table creation ($create$),
- reading from an ETS table (labelled with a tuple containing the selection pattern, $\{read, Pattern\}$),
- writing into an ETS table (labelled with a tuple containing the inserted data, $\{write, Data\}$).

5.1 Identifying Processes

The dynamic nature of Erlang make static process detection hard, therefore we use data-flow reaching [11, 10] to calculate the values of expressions which hide the necessary information for process detection.

We identify different types of process nodes for functions which take part in communication (*Identification Algorithm*):

1. A process node p_i is created in the graph for each `spawn*` call.
2. A process node is present in the graph for each function (f) which takes part in communication (when f sends or receives messages or spawns a new process). In this case we have to identify whether the function f already belongs to a process from the first group. Therefore, we calculate the backward call chain of the function f. If the backward call chain contains a spawned function, then the function f belongs to the process of the spawned function p_i. Thus, the communication edges generated by f are linked to p_i.
3. When a function g takes part in communication, but its backward call chain does not contain a spawned function we create a new process node p_j. This process is identified with the module, the name and the arity of g if there is no communicating function in the backward call chain. Otherwise we select the last communicating function h in the call chain and we identify the created p_j process with module, name and arity of h.
4. There is a "super process" (SP) in the graph which represent the runtime environment. It represents the fact that communicating functions can be called from the currently running process, for example from the Erlang shell.

We create the listed process nodes in a predefined order:

1. At first we collect the spawn expressions from the source code and add them to the set S.
2. We create a process node p_s for each $s \in S$ spawned process and add it to the set P_s
3. We collect the communicating functions C and create its process node (using the second and third step of the identification algorithm).
4. We link every created p_s ($s \in S$) process to its parent process with a $spawn*$ edge.
5. We select each register expression from the source code and add the appropriate *register* link to the graph.
6. Each process node p_j that is not a spawned process ($p_j \notin P_s$) is linked to the node SP.

To identify a spawned process we have to calculate the possible values of the actual parameters of the function call `spawn*(ModName, FunName, Args)`. We calculate the values based on the result of the data-flow analysis [11], while it is a static analysis it is always an approximation of the real dynamic information.

5.2 Message Passing

The next step is to add the message passing edges to the graph. We calculate the message passing edges based on the data-flow information presented in [10]. That analysis links the sent and received messages with a $flow$ edge in the Semantic Program Graph of RefactorErl (Section 2). We use the following algorithm to calculate the communication edges:

1. We select the message sending expressions from the source code and add it to the set M.
2. For each $m \in M$ we calculate the receive expression r_m which receives the sent message.
3. We calculate the containing process node p_m for each $m \in M$ expression and the containing process node p_r for each r_m, and add the $\{send, Message\}$ link from p_m to p_r (where $Message$ is the sent message from the expression m).

5.3 Hidden Communication – ETS tables

ETS tables can be considered as a form of shared memory in Erlang: one process can write some data in it and share it with other processes. Therefore, ETS tables represent hidden communication among processes, thus we add them as a special process relation to our model. Every created ETS table is added to our graph as a special process node, and read and write operation added as special message passing edges:

1. The first step is to select the created ETS tables and add them to the set E.

2. For each $e \in E$ table we create a process node p_e and link it to the parent process. The parent process is the process of the function which calls the function `ets:new/2`.
3. The next step is to detect whether the found table can be referred using its name. We analyse the option list (the second parameter of the call `ets:new/2`) and calculate its possible values by data-flow reaching. If the `named_table` atom is one of them, then we have to calculate the possible names of ETS table by data-flow reaching, and add the name of the ETS table as an attribute to the process node.
4. Each ETS table manipulation (e.g. `insert*, delete*`) added as write operation to the graph between the ETS table node and the process of the expression calling the `ets` functions.
5. Each query operation (e.g. `match*, select*`) added as read operation to the graph between the ETS table node and the process of the expression calling the `ets` functions.

Calculating write edges. The relation $\overset{\mathbf{1f}}{\rightsquigarrow}$ denotes the first order data flow reaching [11]. $n_1 \overset{\mathbf{1f}}{\rightsquigarrow} n_2$ means that the value of node n_2 can be a copy of the value of node n_1. We use this relation to calculate the ETS *write* edges in the following steps:

1. Collect the function calls which refer to an ETS table and change it. Add it to the set W. For example, `ets:insert(Tab, Data)`.
2. For each $w \in W$ call calculate the referred ETS table with data-flow reaching. At first the possible values of w_1 (denoted with E_w) have to be calculated: $e \in E_w$ and $e \overset{\mathbf{1f}}{\rightsquigarrow} w_1$ (where w_1 is the first parameter of the call expression w, e is an expression which value can flow to w_1). If there is an expression $e \in E_w$ which is an atom and its value is *some_name*, then we select the process node (p_e) referring to the named ETS table *some_name*. Otherwise we should find a table reference in E_w which creates the ETS table (a call to `ets:new/2`), and select the process node p_e of the created ETS table.
3. Determine the process node where the call `ets:insert/2` belongs to: p_w. To identify this process we use a similar algorithm that was presented in the second and third step of the *Identification algorithm*. We determine the function f which contains w, and calculate the process of f.
4. Connect the process node p_w and the found ETS table node p_e.

Calculating the *read* edges works similarly to *write* edges, but we have to analyse the query functions of the `ets` module.

The function `ets:rename/2` also has to be considered, because it changes the name of the ETS table, and only the new name can be used. To handle this we will refine our analysis using the Control-Flow Graphs of Erlang programs [10].

6 Motivating Example – Resulted Model

We will illustrate the presented algorithms step by step using the client-server example from Section 4.

Process Identification.

1. There are two spawn expressions in our example at the 7^{th} line in the client module and in the 21^{th} line of the server module, we add their expression nodes to the set S.
2. We create to process node for them: `server:init/0` and `client:input/1`: P_s.
3. We collect the communicating functions and add them to C: `client:start/1, client:loop/1, client:input/1, client:read_input/1, server:do/3, server:connect/1, server:disconnect/1, server:start/1, server:stop/0, server:loop/1, server:handle_job/3`. We create the process node for them using the identification algorithm: `client:start/1, server:start/1, server:stop/0`. Some example:
   - We create the process node `client:start/1`, because there is no spawned process in its backward call chain, and the last function in its backward call chain is itself.
   - We do not create a process node for `client:loop/1`, because it contains `client:start/1` in its backward call chain and it already has a process node.
   - We do not create a process node for `server:loop/1`, because there is a function in its backward call chain which is spawned, thus it already has a process node: `server:init/0`.
4. All spawned process is linked to its parent process: `client:start/1` $\overset{\textbf{spawn}}{\rightarrow}$ `client:input/1` and `server:start/0` $\overset{\textbf{spawn\_link}}{\rightarrow}$ `server:init/0`
5. There is a register expression in the 21^{th} line of the server code, thus we add a register link to the graph: `server:start/0` $\overset{\textbf{register}}{\rightarrow}$ `server:init/0`
6. Processes `client:start/1, server:start/1, server:stop/0` are linked to the SP process.

The resulted graph is shown in Figure 4.

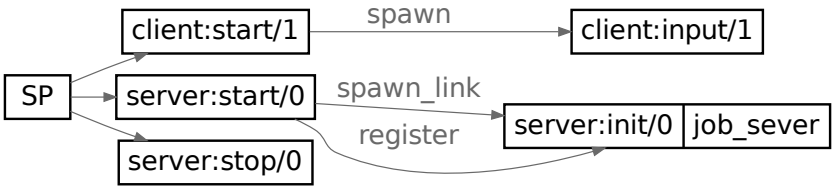


Fig. 4. Process Relation

Message Passing Information.

1. The message passing expression set M contains six expressions from the following lines: the 11^{th}, the 14^{th}, the 17^{th} and the 24^{th} lines of the server code, and the 23^{th} and the 26^{th} line of the client code.
2. The process identifier part of the message sending expressions from the server code is the macro `?Name`, thus the receive expressions for the server code must be in the spawned and register process `job_server`. This is the receive expression from the body of the function `server:loop/1`.
   The process identifier part in the client code is a variable, so we use data-flow reaching to calculate its value, which is the result of the function `self()` in the body of the function `client:start/1`. Therefore, the receive expression must be in the process of `client:start/1`: it is the receive expression form `client:loop/2`.
3. We have to identify the process of the function `server:loop/1` which is `server:init/1` and link the messages from the processes of the message sending expressions (`server:stop/1, client:start/1`) to it.
   Similarly, we link the messages from the process `client:input/1` to the process `client:start/1`.

The resulted graph is shown in Figure 5.

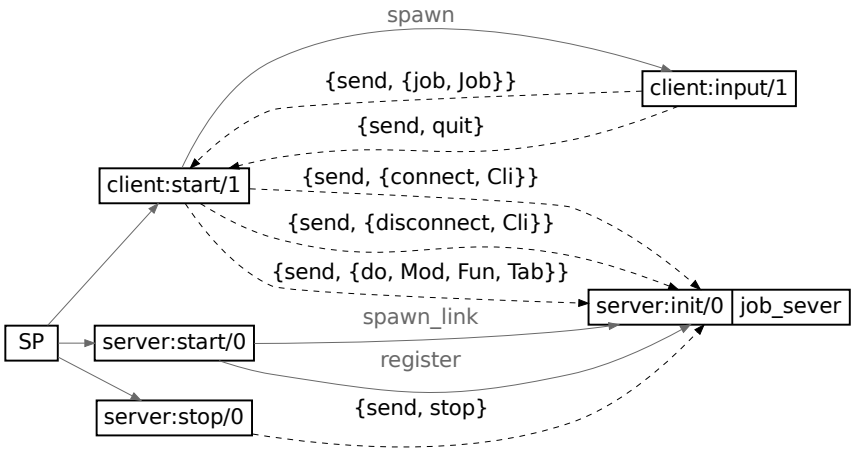


Fig. 5. Communication Model

ETS Usage Information.

1. The set E contains only one expression from the 6^{th} line of the client code which creates a table.

2. We create the process node `ets:new/2` in the graph and link it to its parent process `client:start/1`.
3. The second parameter of the `ets:new/2` call contains the `named_table` atom, therefore we add the value of the first parameter of the call (*data*) as a property of the process node. In this example the parameter is an atom, thus the result of the data-flow reaching is only the atom expression.
4. We select the write operations from the source code: there is a call `ets:insert` in the 51^{th} line of the server code and in the 31^{th} line of the client code. We link the process node of the manipulated ETS table to the process node of the caller function. For example in the server code:
   - We add the w = `ets:insert(Tab, {result, Result})` expression to the set W.
   - We calculate E_w with data-floe reaching. E_w contains the `Tab` variables from the functions `server:handle_job/3, server:loop/2, client:loop/2` and the first actual parameter of the function call `loop(data, Client)` from the body of `client:start/1`. Therefore the value of w can be the atom `data`, so it can refer to the ETS table *data*. Thus p_e is the process node `ets:new/2, data`.
   - The process of the expression w is the process of `handle_job/3`: p_w = `server:init/0`.
   - We create a link from the process `server:init/0` to `ets:new/2, data`.
5. We select the read operations from the source code: there is a call `ets:select` in the 47^{th} line of the server code and a call `ets:match` in the 14^{th} line of the client code. We can select the affected process as it was presented in case of the call `ets:insert`.

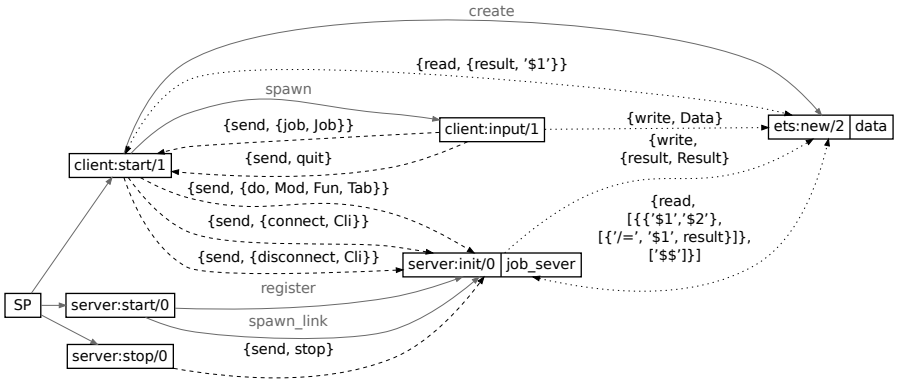


Fig. 6. Communication Model (extended with ETS usage)

The graph, extended with the ETS usage, is shown in Figure 6:

– The grey edges represent the process creation relations.
– The dashed edges stand for the message passing.
– The dotted edges denote the ETS manipulations.

7 Related work

Analysing parallel and distributed Erlang software are key research topics. For example, one goal of the ParaPhrase project [1] is to analyse Erlang programs, statically detect parallel patterns and transform the source code to adjust the advantages of manycore architectures.

The RELEASE project [2] aims to help in developing well designed scalable distributed Erlang software by defining the SD Erlang (Scalable Distributed Erlang). They define language primitives to create process groups. The frequently communicating processes should be placed to the same group and to the same Erlang node. However, the explicit process placement based on the communication flow, is not straightforward for the programmers. Therefore SD Erlang aims to design automatic process placement based on connectivity distance metrics.

The goal of the static analyser tool, Dialyzer [3], is to identify software discrepancies and defects, such as type mismatches, race condition defects, etc. The tool detects message passing by analysing the Core Erlang code [6], and can report concurrent programming defects.

8 Conclusions and Future Work

In this paper we presented a model to represent the communication among Erlang processes. The model contains the most important relations between processes, i.e. process hierarchy, communication through message passing, and hidden relations generated by ETS usage.

We want to make the presented algorithms more efficient and more precise based on the result of the control-flow analysis.

References

1. ParaPhrase project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. http://paraphrase-ict.eu/.
2. RELEASE project: A High-Level Paradigm for Reliable Large-Scale Server Software. http://www.release-project.eu/.
3. *The DIALYZER: a DIscrepancy AnaLYZer for ERlang programs.* http://www.it.uu.se/research/group/hipe/dialyzer.
4. RefactorErl Home Page, 2011. http://plc.inf.elte.hu/erlang/.
5. I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, M. Tejfel, and M. Tóth. Refactorerl – source code analysis and refactoring in erlang. In *Proceeding of the 12th Symposium on Programming Languages and Software Tools, Tallin, Estonia*, 2011.

6. R. Carlsson. An introduction to core erlang. In *In Proceedings of the PLI01 Erlang Workshop*, 2001.
7. F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009.
8. Ericsson AB. *Erlang Reference Manual*. Latest version available online at http://www.erlang.org/doc/reference_manual/part_frame.html.
9. Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, M. Tóth, I. Bozó, and R. Király. Modeling semantic knowledge in Erlang for refactoring. In *International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Selected papers, Presa Universitara Clujeana, ISSN: 2067-1180*, pages 38–53, Cluj-Napoca, Romania, Jul 2009.
10. M. Tóth and I. Bozó. Static analysis of complex software systems implemented in erlang. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *Central European Functional Programming School*, volume 7241 of *Lecture Notes in Computer Science*, pages 440–498. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-32096-5_9.
11. M. Tóth, I. Bozó, Z. Horváth, and M. Tejfel. First order flow analysis for Erlang. In *Proceedings of the 8th Joint Conference on Mathematics and Computer Science, MACS 2010*, 2010.

Skel: A Streaming Process-based Skeleton Library for Erlang (Early Draft!)

Archibald Elliott[1], Christopher Brown[1], Marco Danelutto[2], and Kevin Hammond[1]

[1] School of Computer Science, University of St Andrews, Scotland, UK.
[2] Dept. Computer Science, University of Pisa, Pisa, Italy.

Emails: ashe@st-andrews.ac.uk, {chrisb, kh}@cs.st-andrews.ac.uk, marcod@di.unipi.it

Abstract. With the increasing shift towards parallel programming, parallel programs are still notoriously difficult to implement. Indeed, parallelism is usually made even more difficult by programmers typically using sequential or small-scale parallel programming techniques. Functional languages such as Erlang are increasingly starting to dominate the parallelism scene with their typical first-class light-weight parallelism models.
In this paper we introduce a new process-based skeleton framework for Erlang based on streaming models. In particular, we show a number of core "primitive" skeletons that can be used as a basis of more complex parallel systems. We give details of the skeletons, in Erlang, and in our full paper we promise to show promising and scalable speedups on a manycore machine on a number of examplars.

1 Introduction

The single-core processor, which has dominated for more than half a century is now obsolete. Machines with dual-, quad- and even hexa-core CPUs are already common place in desktop machines and CPUs with 50 cores as standard have already been announced [3]. There has been a *seismic* shift between sequential and parallel hardware, but programming models have been very slow to keep pace. Indeed, many programmers still use outdated sequential models for programming parallel systems, where parallel concepts have effectively been *bolted-on* to the language, rather than high-level parallel constructs being *first-class*. What is needed is an effective solution to help programmers *think parallel*. In the context of parallel programming, parallel design patterns represent a natural language description of a recurring problem and of the associated solution techniques that the parallel programmer may use to solve that problem.

[3] Intel's Many Integrated Core Family

An *algorithmic skeleton*, is a computational, abstract entity, typically described by a concurrent activity graph, modelling and embedding a frequently recurring parallelism exploitation pattern, provided to the application programmer as a new abstraction in the programming framework at hand; a parallel application is therefore developed as a composition of skeletons, which may be specialised by providing (suitably wrapped) sequential portions of code implementing the business logic of the application.

In this paper we introduce `skel`: a process-based streaming skeleton library for Erlang. `skel` aims to model the most common set of "primitive" skeletons that are typically used to make up more complex systems.

In particular, the contributions of our paper are:

1. we describe a new parallel skeleton library for Erlang. To our knowledge, this is the first time parallel skeletons have been exploited this way in Erlang; and,
2. we demonstrate the effectiveness of our skeletons on a set of synthetic benchmarks, therefore demonstrating the parallel capabilities of Erlang;

2 Erlang

Erlang is a strict, impure, functional programming language with support for *first-class* concurrency. This concurrency model allows the programmer to be explicit about processes and communication, but implicit about placement and synchronisation. Erlang supports a *lightweight* threading model, where processes model small units of computation (tasks) that are executed on a capability. The scheduling of processes is handled automatically by the Erlang Virtual Machine, which also provides basic load balancing mechanisms. Erlang typically has three primitives for handling concurrency:

- `spawn()`, allowing new functions to execute in a lightweight Erlang process;
- `!`, allow messages to be explicitly sent from one Erlang process to another; and,
- `receive`, to allow messages to be received in another process queue.

Furthermore, Erlang also supports fault tolerance, by allowing groups of processes to be *supervised*, and new instances of processes can be spawned in the case failure. Although Erlang supports concurrency, there has been little research into how Erlang can be used to effectively support *deterministic* parallelism.

3 Skeletons in Erlang

The design of the `skel` library has been based upon the design of FastFlow [1], a parallel programming framework for multi-core platforms written in C++, but with significant changes to take advantage of features provided by the Erlang language and VM.

3.1 Skeletons

So far we have only implemented seven core "primitive" skeletons so far, however they can be combined to make more complex skeletons. They are:

Seq a skeleton to encapsulate an indivisible portion of sequential code.
Pipe the functional composition of multiple skeletons. In `skel`, these are implicit.
Farm a skeleton that schedules inputs onto replicas of a pipeline, and then collects the results back into a single stream.
Map a skeleton that can decompose each item, put each decomposed part through its own replica of a pipeline, and then recompose the results back into a single item again.
Reduce a skeleton that applies a treefold, in parallel, over each decomposed item.
Feedback a skeleton that can send independent items back through a skeleton.
Ord a skeleton to restore order to a stream of items[4].

The skeletons we have implemented here are a set of foundation skeletons, encapsulating common functional patterns. By investigating foundational skeletons, we can explore the parallel capabilities of Erlang while providing a strong framework that allows for more complex skeletons to be implemented in the future.

Most of these skeletons need to have one or more skeletons nested inside them to work correctly, for instance a **Farm** skeleton requires at least a single **Seq** skeleton to be nested within it or there would be no point in having the **Farm** skeleton in your pipeline. When more than one skeleton is specified to be nested, those skeletons will be assembled into a pipeline that is nested inside the other skeleton. The only two skeletons that cannot have other skeletons nested inside them are **Seq**, which is designed to encapsulate sequential code, and **Reduce**, which applies a binary function to each item.

The `skel` API that is presented to programmers is a single function:

```
skel:run(Pipeline, ItemSource).
```

- `Pipeline` an ordered list of skeletons that each item on the stream is to be processed through.
- `ItemSource` either a list of stream items, or is a module with functions that can supply stream items.

The process will then receive a message in response with the contents `{sink_results, ResultItems}` where `ResultItems` is a list of the results of sending the items from `ItemSource` through `Pipeline`. In the code examples, this is denoted by `% -> {sink_results, ItemSource}` (`sink_results` just tells the receiver that this message contains the results from the sink).

[4] The implementation of this skeleton is not described, though it is present for applications where the order of items is important.

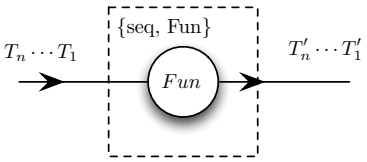


Fig. 1. The **Seq** Skeleton

In the following schematics, circles should be read as explicit Erlang processes – if they are white then they have some user-defined behaviour in them, as specified by the identifier inside them, and if they are black they only have some system-level logic happening in them. Rectangles with solid edges denote an internal pipeline, and the outer dotted rectangle denotes which processes are contained in that particular skeleton. Data messages travel along the lines with arrows, though as mentioned, they are not explicit channel objects.

Pipe Pipe is the only skeleton that does not have any of its own processes, and does not explicitly exist. The basic building block of our library is a pipeline, so skeletons are defined in terms of pipelines with 1 or more stages. We then use algorithms (termed "assembly algorithms") to turn this skeleton declaration into a system that can run computations on a stream of items. Our implementation currently contains two assembly algorithms: a parallel one (the default) and a sequential one. Functionally, both assemblers will give you exactly the same results, however the assembled process structures (and hence the parallelism degrees) are completely different.

The parallel assembly algorithm is one that maps over the list of skeleton declarations, using the details in each declaration to create a list of what we term a "maker functions". We then do a right fold over this list, so that we start each one in reverse pipeline order, starting with the process id of a sink (the end of the pipeline). The "maker functions" take the process id of the receiving part of the next skeleton, start up that skeleton's requisite processes, and then returns the process id of the receiving part of that skeleton, hence the fold right.

The sequential assembly algorithm makes each skeleton declaration into a single function, then composes them together into an entirely sequential version of the pipeline. On the face of it, this may not seem sensible, however it is useful for benchmarks and for working out, at a functional level, what each skeleton does. All future focus is on the parallel versions of each skeleton.

Seq Seq is the most basic of all the skeletons. It consists of a single process that applies a function, `Fun`, to any data messages it receives, before sending the results on to the next skeleton. Should that process receive an end-of-stream message it will exit immediately. A schematic of this skeleton is shown in Figure 1, and an example of its operation can be seen in Figure 2.

```
skel:run([{seq, fun (X) -> X+1 end}],
         [1,2,3,4,5,6]).
% -> {sink_results,[2,3,4,5,6]}
```

Fig. 2. An example of the **Seq** Skeleton

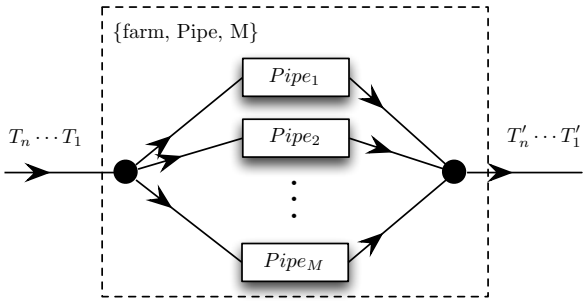


Fig. 3. The **Farm** Skeleton

Farm In a Farm skeleton, an emitter forwards inputs into one of `M` replicas of the pipeline, `Pipe`, which then forwards results onto a collector and then the next skeleton. A schematic of this skeleton is shown in Figure 3, and an example of its operation is shown in Figure 4.

The emitter process is very simple. When it receives a data message, it forwards that message on to any single one of the pipeline replicas. When it receives an end-of-stream message, it forwards this message to every single one of the pipeline replicas. At the moment the scheduling algorithm is round-robin, but we expect to add other algorithms in the future.

The collector process is also simple. When it receives a data message (from any of the pipeline replicas), it forwards the message onto the next skeleton. The collector waits for `M` end-of-stream messages before exiting, to make sure that it has received all messages from each of the pipeline replicas.

Map In a Map skeleton, each item is decomposed by the `Decomp` function into a number, m_i, of parts, then each of these is forwarded into one of m_i replicas of the pipeline, `Pipe`. After the pipes, each part is then forwarded to a recomposer, which combines all the parts back into a single item using the `Recomp` function. A schematic is shown in Figure 5, and an example of its operation is shown in Figure 6.

The `Decomp` function is for disassembling a collection-like item into a list of its constituent parts, and the `Recomp` function is for turning the list of constituent parts back into a collection-like item. If you're dealing with lists, you

```
skel:run([{farm, [{seq, fun(X)-> X+1 end}],
                  3}],
         [1,2,3,4,5,6]).
% -> {sink_results,[2,5,3,6,4,7]}
```

Fig. 4. An example of the **Farm** Skeleton

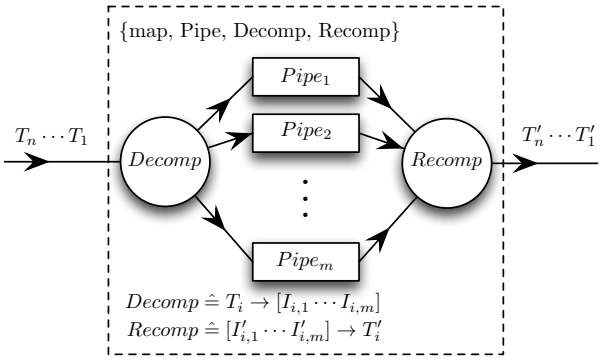


Fig. 5. The **Map** Skeleton

can just use an identity function, and helpfully Erlang has `tuple_to_list/1` and `list_to_tuple/1` if you're moving around tuples, like we are in Figure 6.

The decompose process is less trivial than a normal emitter. It waits for items and then splits them into many parts. After each data message is split, each part is labelled with a unique tag that relates to the input item, the index of that part in the collection of parts, and the total count of parts for that item (this is the reason we have the stack in each data message). Each part is then sent through a replica of the pipeline.

As we cannot make any assertions about the number of parts that the `Decomp` function will produce, the decompose process also has the ability to start more replicas of the pipeline if it does not have enough for all the parts of an input.

The recompose process is also quite complicated. It waits for each part, then puts it into a store (keyed by item unique tag and part index), along with the data about how many parts it has so far received, and how many it is expecting. When it has received as many as it is expecting, it calls the `Recomp` function with a list of the inputs that it has received (therefore preserving the order the parts were in when they came out the `Decomp` function), which produces a single item again, which it then forwards to the next skeleton. The process also stores the highest number of parts that it has received for any input, in order to know how many end-of-stream messages to wait for before exiting.

```
skel:run([{map, [{seq, fun(X)-> X+1 end}],
                fun erlang:tuple_to_list/1,
                fun erlang:list_to_tuple/1}],
         [{1,2},{3,4}]).
% -> {sink_results,[{2,3},{4,5}]}
```

Fig. 6. An example of the **Farm** Skeleton

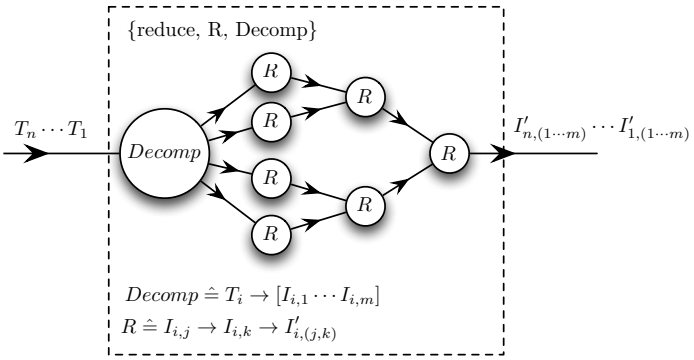


Fig. 7. The **Reduce** Skeleton

Reduce The Reduce skeleton executes a parallel treefold over the decomposed parts of each input. Inputs are first decomposed into several parts, and then they're submitted into a binary tree of reduce processes. The reduce processes compute their result with both parts of the item that they receive, then forward the result on to the next reduce process. A schematic is shown in Figure 7, and an example of its operation is shown in Figure 8.

The decompose process waits for inputs, then splits them into m_i parts. After each data message is split, each part is labelled with a unique tag that relates to the input item, and a number that indicates how many reduce steps it will go through (this is calculated as $\lceil log_2(m_i) \rceil$). The parts are distributed to the correct level of the reduce process tree corresponding to m_i parts. Two parts are given to each reducer, and then any reducers still waiting for a part are given a unit input. The unit input is a system-level message that would act as the unit value for the computation (this is explained further, later). When the decomposer receives an end-of-stream message, it forwards 2 end-of-stream messages to each reducer at the widest level of the tree.

Again, we cannot make any assertions about the number of parts the `Decomp` function produces, so process instantiation is dynamic.

The reduce processes wait for parts or unit values. For the first part (or unit value) of an item that they get, they store it, and then wait for another. When they receive the second part (or unit value), they use a fairly simple algorithm to

```
skel:run([{reduce, fun(X,Y) -> X + Y end,
                   fun erlang:tuple_to_list/1}],
         [{1,2,3,4,5,6},{7,8,9,10,11,12}]).
% -> {sink_results,[21,57]}
```

Fig. 8. An example of the **Reduce** Skeleton

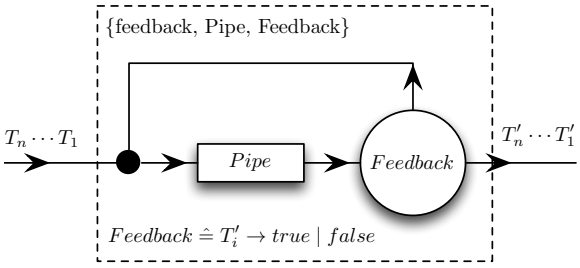


Fig. 9. The **Feedback** Skeleton

work out what to forward. If they received two unit values, they forward the unit value. If they received a part and a unit value, they forward the part, and only if they received two parts do they combine them using the reduce function, before forwarding on the result of the computation. Before forwarding an input, the reducer decrements the count of steps that the part has to go through. When this count gets to zero, the reduce label on the data message is taken off, to preserve the invariant that the label stack is the same on leaving a skeleton as it is when it entered it. When a reducer receives two end-of-stream messages, it forwards a single end-of-stream message to the next reducer.

Feedback The Feedback skeleton sends items through a pipeline, `Pipe`, and then checks a predicate, `Feedback`, to find out whether it should send that input through the pipeline again or forward it to the next skeleton. A schematic is shown in Figure 9, and an example is shown in Figure 10.

The only complication in this skeleton comes in the race condition between an end-of-stream message coming in, and items that are going through the feedback loop. Because we cannot prioritise one "stream" over another in Erlang (which would let us prioritise the feedback queue, and just stop receiving from the previous skeleton), we instead maintain two counters of items in two parts of the skeleton. The counters are stored in a separate process, due to the lack of any shared state in Erlang, which also allows us to make any operations on them atomic, and allows the receiver to subscribe to updates anyone makes to the counters. The first counter keeps a count of items in the pipeline, and the second keeps a count of items in the pipeline. Once an end-of-stream message is received, the receiver process (the black circle in Figure 9) continues receiving,

```
skel:run([{feedback, [{seq, fun(X) -> X+1 end}],
                     fun(X) -> X < 5 end}],
         [1,2,3,4,5,6,7,8,9,10]).
% -> {sink_results,[5,6,7,8,9,10,11,5,5,5]}
```

Fig. 10. An example of the **Feedback** Skeleton

but also waits to be told when both counters get to zero. When both counters reach zero, the end-of-stream message can be forwarded without fear of race conditions.

4 Related Work

Since the nineties, the "skeletons" research community has been working on high-level languages and methods for parallel programming [4, 5, 3, 6, 2, 9]. Skeleton programming requires the programmer to write a program using well-defined abstractions (called skeletons) derived from higher-order functions that can be parameterized to execute problem-specific code. Skeletons do not expose to the programmer the complexity of concurrent code, for example synchronization, mutual exclusion and communication. They instead specify abstractly common patterns of parallelism – typically in the form of parametric orchestration patterns – which can be used as program building blocks, and can be composed or nested like constructs of a programming language. A typical skeleton set includes the pipeline, the task farm, map and reduction.

Early proposals of pattern-based parallel programming frameworks have been mainly focused on distributed memory platforms, such as clusters of workstations and grids [13, 11]. All these skeleton frameworks provide several parallel patterns covering mostly task and data parallelism. These patterns can usually nested to model more complex parallelism exploitation patterns according to the constraints imposed by the specific programming framework. Recently skeletons gained renewed popularity with the arrival of multi-core platforms, the consequent diffusion of parallel programming frameworks, and their adoption in some programming frameworks, such as FastFlow [1], Intel Threading Building Block (TBB) [10] and to a limited extent the Microsoft Task Parallel Library [12]. Google MapReduce [7] brings to the mainstream of out-of-core data processing the map-reduce paradigm. The main features of these frameworks, as well as many other experimental ones, are surveyed in [8].

5 Conclusions and Future Work

Acknowledgements

This work has been supported by the European Union Framework 7 grant IST-2011-288570 "ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems", `http://www.paraphrase-ict.eu`.

References

1. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: High-level and Efficient Streaming on Multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, 2012.
2. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High Level Programming Language and its Structured Support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.
3. G. H. Botorog and H. Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Proc. of the 5th International Symposium on High Performance Distributed Computing (HPDC'96)*, pages 243–252. IEEE Computer Society Press, 1996.
4. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.
5. M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.
6. J. Darlington, Y. Guo, Y. Jing, and H. W. To. Skeletons for Structured Parallel Composition. In *Proc. of the 15th Symposium on Principles and Practice of Parallel Programming*, 1995.
7. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1):107–113, 2008.
8. H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.
9. M. Hamdan, P. King, and G. Michaelson. A Scheme for Nesting Algorithmic Skeletons. In K. Hammond, T. Davie, and C. Clack, editors, *Proc. of the 10th International Workshop on the Implementation of Functional Languages (IFL'98)*, pages 195–211. Department of Computer Science, University College London, 1998.
10. Intel Corp. *Threading Building Blocks*, 2011.
11. H. Kuchen. A Skeleton Library. In B. Monien and R. Feldman, editors, *Proc. of 8th Euro-Par 2002 Parallel Processing*, volume 2400 of *LNCS*, pages 620–629, Paderborn, Germany, Aug. 2002. Springer.
12. D. Leijen and J. Hall. Optimize Managed Code for Multi-Core Machines. *MSDN Magazine*, Oct. 2007.
13. M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.

Pure and Lazy Lambda Mining

Nicolas Wu[1], José Pedro Magalhães[1], Jeroen Bransen[2], and Wouter Swierstra[2]

[1] {nicolas.wu,jose.pedro.magalhaes}@cs.ox.ac.uk
Department of Computer Science, University of Oxford
[2] {j.bransen,w.s.swierstra}@uu.nl
Department of Computer Science, Utrecht University

Abstract. The ICFP programming contest has been run every year since 1998. This year, participants were invited to program a virtual mining robot to collect resources called 'lambdas' while avoiding falling rocks, getting trapped, or drowning. The overall score of a route was determined by the number of lambdas collected and the number of moves required to collect those lambdas. The problem specification was extended four times over the course of the competition, which demanded efficient and correct code to be produced under tight deadlines. As such, this has provided an excellent means of substantiating the claim that functional programming languages help to produce code that is both modular and reusable (Hughes, 1989).
In this paper we describe a solution based on our submission to this year's problem. This solution uses many features of Haskell: pure immutable data structures, laziness, higher-order functions, concurrency, and exception handling. Each of these features plays an essential part in our overall solution. We present our work as a tutorial, where we demonstrate how these key elements can be composed together. In this exposition, we stress the importance of how the code was structured in such a way that made safely refactoring and extending the model a relatively easy task, and how Haskell's strong type system made it possible for our team to remain agile under changing specifications.

1 Introduction

This paper describes a solution to the 2012 ICFP programming contest.[3] This programming contest allows participants to write solutions in any language, or combination of languages, in a time frame of 72 hours. Our solution was entirely implemented in Haskell (Peyton Jones, 2003), a lazy, purely functional, statically typed language. We describe our solution as it was developed in the 72 hours of the contest, plus some later refactoring for readability and bug fixing. We rely on multiple features of Haskell:

Laziness and Purity Each of the different solvers explores (potentially overlapping) parts of the solution space. Calculating the results of a next move is not a cheap operation, as it requires examining the whole map. In particular, there could be falling rocks anywhere on the map, and computing the new state after each move requires updating the positions of each of these rocks. To avoid duplicating this

[3] http://icfpcontest2012.wordpress.com/task/

expensive operation across the different solvers, they share a single, immutable, and lazily generated representation of the search space, by means of a digital search trie.

Higher-order functions We parameterised a class of solvers by a comparison function, used to rank the lambdas when deciding which to visit next. In this way, we could easily develop new solvers which used slightly different heuristics. As the problem specification changed, this turned out to be quite useful. For example, during the course of the contest the problem was changed such that the mine could flood, making lower portions inaccessible. To cope with this, we defined two new solvers: one which collected 'lower' lambdas first, before they were inaccessible; the other collected 'higher' lambdas first to try to prevent the robot from drowning. Both were defined simply by changing the comparison function.

Concurrency Instead of trying to find a single algorithm that performs well on any given mine, we decided to have several solvers compete. Each of these solvers uses different heuristics to compute a solution. The main program forks off threads for each solver and returns the best overall solution.

Exception handling According to the contest specifications, programs are sent a SIGINT (a POSIX interrupt signal) at a fixed time limit. Upon receiving this signal, the program has ten seconds to return a result. Our main program uses this time to kill all auxiliary threads and output the best result. As a consequence, we will 'never' run out of time (as long as the scheduler is fair, the killing of threads is fast, etc.) and our program will print a valid solution. If any individual thread should crash, all other solvers will remain unaffected.

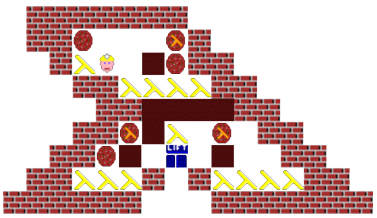


Fig. 1. Graphical representation of a mine

Figure 1 shows a graphical depiction of a game in progress. The goal is to compute a sequence of moves for the robot to collect as many lambdas as possible, without being crushed by falling rocks. If all the lambdas are collected, reaching the exit (portrayed as "LIFT" in Figure 1) gives an extra score bonus.

This paper is organised as follows. Section 2 introduces our model, at the same time explaining the problem specification. Section 3 describes the use of tries to represent the game state and paths through the mine, while Section 4 describes the solvers and how they use the trie. In Section 5 we explain the use of concurrency for running multiple solvers in parallel, and in Section 6 we discuss the problem specification extensions and the associated changes to our solution. We conclude in Section 7, with a brief discussion

on possible improvements, and guidelines for developing Haskell code in a group with tight deadlines.

2 Pure modelling

In this section we describe how we model and simulate the problem in Haskell.

2.1 Model

The model represents the entire state of a mine at any given time, and forms an important interface for the rest of the system: the simulator (Section 2.2) takes one state of the model to the next, the parser must produce a value of this type while the visualiser outputs a visual rendering of the model (Section 2.3) , and various strategies can be employed based on a particular starting state and make use of data held within the model (Section 4).

The basic building block of a mine is a *Tile*, which holds information about what exists at a particular coordinate:

$$\textbf{data}\ \mathit{Tile} = \mathit{Robot} \mid \mathit{Wall} \mid \mathit{Rock\ Bool} \mid \mathit{Lambda} \mid \mathit{Earth} \mid \mathit{Empty} \mid \mathit{Exit}$$

Note that rocks are parameterised by a Boolean which indicates whether or not a rock is falling: when the robot is directly beneath a falling rock, it is crushed. Two convenience functions are provided, one which indicates whether or not a particular tile is a rock, and the other distinguishes rocks that are falling:

$$\begin{aligned}
&\mathit{isRock} :: \mathit{Tile} \to \mathit{Bool}\\
&\mathit{isRock}\ (\mathit{Rock}\ _) = \mathit{True}\\
&\mathit{isRock}\ _ \quad\quad\ = \mathit{False}\\
&\mathit{isRockFalling} :: \mathit{Tile} \to \mathit{Bool}\\
&\mathit{isRockFalling}\ (\mathit{Rock\ True}) = \mathit{True}\\
&\mathit{isRockFalling}\ _ \quad\quad\quad = \mathit{False}
\end{aligned}$$

Each tile in the mine is given a specific coordinate, which is simply a pair of *Int* values named *Coord*:

$$\textbf{type}\ \mathit{Coord} = (\mathit{Int}, \mathit{Int})$$

Putting these elements together, we are interested in an array that is indexed by *Coord*s and contains *Tiles*. This describes the layout of the mine:

$$\textbf{type}\ \mathit{Layout} = \mathit{Array\ Coord\ Tile}$$

Using an array for this representation is appropriate, since we need to perform lookups of elements at coordinates very often, and arrays have constant time lookup.

It is useful to define a function that checks the value of a tile in the layout at a particular coordinate, by dereferencing the appropriate location in the array:

$$
\begin{aligned}
&\mathit{isTile} :: \mathit{Layout} \to \mathit{Coord} \to \mathit{Tile} \to \mathit{Bool} \\
&\mathit{isTile}\ l\ xy\ t = l\,!\,xy \equiv t
\end{aligned}
$$

There is an important caveat to using this function and others like it which make use of $(!)$, the unsafe indexing operator. This operator makes no effort to ensure that the coordinates being sought are within the bounds of the array, and this is a danger which could easily result in an exception being thrown at runtime. We use $(!)$ instead of a safe lookup for efficiency reasons.

Another useful utility function finds the coordinates of all the tiles which satisfy a given predicate:

$$
\begin{aligned}
&\mathit{findTiles} :: (\mathit{Tile} \to \mathit{Bool}) \to \mathit{Layout} \to [\mathit{Coord}] \\
&\mathit{findTiles}\ p = \mathit{map}\ \mathit{fst} \circ \mathit{filter}\ (p \circ \mathit{snd}) \circ \mathit{assocs}
\end{aligned}
$$

This works by getting a list of all the associations in the array and representing these as a value of type $[(\mathit{Coord}, \mathit{Tile})]$. This list is then filtered by the predicate, before the coordinates are extracted.

While the *Layout* structure holds much of the information required during the game, some essential features are lacking, such as the number of moves that have passed since the beginning of the game. The whole state is saved in a structure named *Mine*, which contains all the information required for assessing the current score:

$$
\begin{aligned}
\textbf{data}\ \mathit{Mine} = \mathit{Mine}\ \{ & \mathit{layout} \quad :: \mathit{Layout} \\
, & \mathit{robot} \quad :: \mathit{Coord} \\
, & \mathit{lambdas} :: \mathit{Int} \\
, & \mathit{moves} \quad :: \mathit{Int} \}
\end{aligned}
$$

In particular, *Mine* stores the current position of the robot along with the number of remaining lambdas and the number of moves it has taken to reach this point, since this is an important part of calculating the score.

When the robot has finished collecting all the lambdas, the exit opens and the robot is allowed to leave the mine. Our representation indicates that the robot has exited when the robot's coordinates correspond with the *Exit* tile in the layout:

$$
\begin{aligned}
&\mathit{isDone} :: \mathit{Mine} \to \mathit{Bool} \\
&\mathit{isDone}\ \mathit{mine} = \mathit{isTile}\ (\mathit{layout}\ \mathit{mine})\ (\mathit{robot}\ \mathit{mine})\ \mathit{Exit}
\end{aligned}
$$

The task of ensuring that the robot can only enter an exit when all lambdas have been collected is left to the simulator, which we explain in the next section.

2.2 Simulation

The simulation code determines how the system responds to the robot's actions: each time the robot makes a move, the world is updated, and a new *Mine* value is calculated.

The robot can perform several moves: moving up, down, left, right, waiting, or aborting the mission. For brevity, the data constructors that represent these moves contain only the initial letter of each action:

```
data Move = L | R | D | U | W | A
```

We often calculate coordinates based on a sequence of moves; the following function returns a coordinate that has been shifted by some movement value:

```
(⇝) :: Coord → Move → Coord
(x,y) ⇝ L = (x−1,y  )
(x,y) ⇝ R = (x+1,y  )
(x,y) ⇝ D = (x  ,y−1)
(x,y) ⇝ U = (x  ,y+1)
(x,y) ⇝ _ = (x  ,y  )
```

For example, this operator is used to verify whether the robot has been crushed by a rock, which happens whenever the tile directly above the robot is a falling rock:

```
isDead :: Mine → Bool
isDead mine = isRockFalling (layout mine ! (robot mine ⇝ U))
```

The score is calculated by multiplying a constant factor per collected lambda minus the number of moves the robot made. The constant depends on how the game ended, and is 75 when all lambdas were collected, 25 when the robot dies, and 50 if the robot aborted (which is the default action when no more moves are made):

```
type Score = Int
score :: Mine → Mine → Score
score mine0 mine = multiplier ∗ collected − moves mine
  where collected = lambdas mine0 − lambdas mine
        multiplier | isDone mine = 75
                   | isDead mine = 25
                   | otherwise   = 50
```

The central function used to simulate the robot's progression through a mine is *step*, which takes a current mine, a move, and steps the simulator through that move:

```
step :: Mine → Move → Mine
step mine A    = mine
step mine move = mine′
  where
    (layout′, robot′) = stepRobot mine move
    layout″           = array ((bounds ∘ layout) mine) $
      concat [ updRocks (mine {layout = layout′}) (x,y) (layout′ ! (x,y))
             | y ← [1..h], x ← [1..w]]
    moves′            = 1 + moves mine
    lambdas′  | isTile (layout mine) robot′ Lambda = lambdas mine − 1
              | otherwise                         = lambdas mine
    (w,h)             = (snd ∘ bounds ∘ layout) mine
    mine′             = mine {layout  = layout″
```

```
    , robot    = robot'
    , lambdas  = lambdas'
    , moves    = moves' }
```

When a move other than an *A* is requested, the simulator returns the result of the updated record *mine*′. The *layout* field is updated in two stages. First the value of the layout is calculated after the robot has made its step and stored in *layout*′, and then this value is used in creating a new array, *layout*″ that contains the state of the mine after all the falling of rocks has been calculated. This follows the problem specification.

Updating the robot is left to the *stepRobot* function, which returns the layout modified to take into account the robot's movement, and gives the new coordinate of the robot:

```
stepRobot :: Mine → Move → (Layout, Coord)
stepRobot mine move =
  case l ! xy' of
    Earth      → (l // [(xy', Robot), (xy, Empty)], xy')
    Empty      → (l // [(xy', Robot), (xy, Empty)], xy')
    Lambda     → (l // [(xy', Robot), (xy, Empty)], xy')
    Exit       | lambdas mine ≡ 0
               → (l // [(xy', Robot), (xy, Empty)], xy')
    Rock _     | (move ≡ L ∨ move ≡ R) ∧ isTile l (xy' ⇝ move) Empty
               → (l // [(xy', Robot), (xy, Empty), (xy' ⇝ move, Rock False)], xy')
    _          → (l // [(xy, Robot)], xy)
  where l  = layout mine
        xy = robot mine
        xy' = xy ⇝ move
```

Moving towards earth, an empty tile, or a lambda simply updates the robot position, leaving an empty space behind. Moving towards the exit is only allowed if all the lambdas have been collected. Moving towards a rock is possible if the movement is sideways, and there is empty space next to the rock being pushed. All other movements are invalid, and the robot remains in the same position.

Another crucial function is *updRocks*, which is responsible for updating the position of rocks after the robot has moved:

```
updRocks :: Mine → Coord → Tile → [(Coord, Tile)]
updRocks mine xy (Rock _)
  | isFallDown    l xy = [(xy, Empty), (xy ⇝ D      , Rock True)]
  | isFallRight   l xy = [(xy, Empty), (xy ⇝ D ⇝ R, Rock True)]
  | isFallLeft    l xy = [(xy, Empty), (xy ⇝ D ⇝ L, Rock True)]
  | isFallLambda  l xy = [(xy, Empty), (xy ⇝ D ⇝ R, Rock True)]
  | otherwise          = [(xy, Rock False)]
  where l = layout mine
updRocks _ xy tile = [(xy, tile)]
```

The functions *isFallDown*, *isFallRight*, *isFallLeft*, and *isFallLambda* determine whether the rock will fall in a particular direction. These are all predicates that take a *Layout* and a *Coord*, and simply output the appropriate *Bool*.

Keeping the entire state of a mine as a single value of type *Mine* enables the definition of *step* to remain relatively simple, since all of the required data for an update is held in a single structure. This complete encapsulation of state means that there are no implicit outside dependencies to handle when trying to evaluate a particular mine.

2.3 Input and Output

The input maps are supplied in text format. To read these into our model, a text parser was written using Attoparsec[4], where operations were based *ByteString*s for efficiency reasons. The input format is simple, so the parser is unsurprising and therefore omitted in this presentation.

Visualising the maps in a user-friendly way was not a requirement of the contest. However, especially during development, it is helpful to visualise maps, generated solutions, and to be able to manually play each mine. Due to time considerations we developed only a simple ANSI text-based visualiser, which was enough for our testing purposes.

3 The game trie

One of the key benefits of Haskell is its purity, allowing us to share game states across different solvers. Our strategy for exploiting this is to spawn a number of different agents which explore a shared data structure that holds paths to different game states together with their scores.

3.1 Tries

The structure we use to encode paths on the mine is a non-empty trie (Hinze, 2000):

$$\textbf{data}\ \mathit{Trie}\ k\ v = \mathit{Trie}\ \{\mathit{root} :: v, \mathit{branches} :: \mathit{Map}\ k\ (\mathit{Trie}\ k\ v)\}$$

An important aspect of a value of type *Trie k v* is that it can behave like a map of type *Map* $[k]$ *v*, and this forms the basis of an intuitive interface with a number of well-understood standard functions. These standard functions on *Trie* will prove useful in our strategy code (Section 4), since the entire search space of a game can be encoded as a trie, mapping sequences of moves to a game state:

$$\begin{aligned}&\textbf{type}\ \mathit{GameTrie} \ = \mathit{Trie}\ \mathit{Move}\ \mathit{GameState}\\ &\textbf{data}\ \mathit{GameState} = \mathit{GameState}\ \{\mathit{gameStateMine} :: \mathit{Mine}\\ &\qquad\qquad\qquad\qquad\qquad\quad , \mathit{gameStateScore} :: \mathit{Score}\}\end{aligned}$$

For instance, we can lookup the *GameState* associated with a certain path by using the familiar *lookup* function:

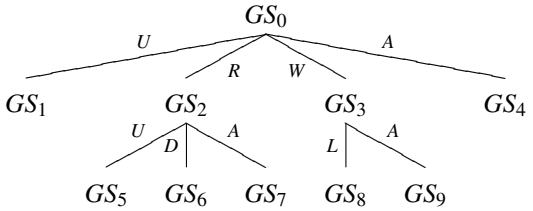
[4] `http://hackage.haskell.org/package/attoparsec`

```
lookup :: (Eq k, Ord k) ⇒ [k] → Trie k v → Maybe v
lookup []       (Trie v _)   = Just v
lookup (k : ks) (Trie _ kvs) = Map.lookup k kvs >>= lookup ks
```

In our setting, a *Path* is a list of moves:

```
type Path = [Move]
```

The type *GameTrie* operates much like the type of *Map Path GameState*, but its encoding is very efficient; each branch of the tree encodes one possible move, as illustrated in the following figure:



In this example, starting from some initial game state GS_0, the robot can move up and die, resulting in game state GS_1, with no further paths. Alternatively, the robot can go right, and then proceed either up, down, or abort. We compute a *GameTrie* by starting with an initial state (of score zero), and considering only valid moves from the current position:

```
mkTrie :: (Eq k, Ord k) ⇒ v → (v → [k]) → (v → k → v) → Trie k v
mkTrie v f next = Trie v (Map.fromList [(k, mkTrie (next v k) f next) | k ← f v])

gameTree :: Mine → GameTrie
gameTree mine0 = mkTrie (GameState mine0 0 (hash mine0))
                        (goodMoves ∘ gameStateMine)
                        (mkGameState mine0 ∘ gameStateMine)
```

We omit the function *mkGameState*, which simply computes the current *GameState*, and function *goodMoves*, which returns the valid moves for the robot. One of the key features of our solution is that the *GameTrie* represents all the paths in the mine, and this trie is shared over the different robot strategy algorithms. This means that states are never computed twice; if strategy one already went down a particular path, strategy two can immediately get the corresponding game state for that path, without having to step through each move.

Another useful property of values of type *Trie k v* is that they behave like trees of type *Tree* $([k], v)$, which brings another family of standard functions that are well understood. In particular, we can traverse the tree in breadth-first order, computing all possible paths of increasing lengths:

```
flatten :: Trie k v → [([k], v)]
flatten = concat ∘ levels
levels :: Trie k v → [[([k], v)]]
```

```
levels tree = (map extract ∘ iterate expand) [([], tree)]
  where
    expand :: [([k], Trie k v)] → [([k], Trie k v)]
    expand = concatMap (λ(sk, Trie _ kts) → map (first (:sk)) (Map.toList kts))
    extract :: [([k], Trie k v)] → [([k], v)]
    extract = map (λ(sk, Trie v' _) → (reverse sk, v'))
```

In Section 3.2 we will use variations of these functions to build efficient pathfinding algorithms that are used to search for solutions within the *GameTrie*.

3.2 Pathfinding

The key to our strategy is finding our way around a *Trie* structure, and identifying a path which leads to a high score. The following function, for example, would find the paths to the exit:

```
solve :: Mine → [(Path, GameState)]
solve mine = (filter (isDone ∘ gameStateMine ∘ snd) ∘ flatten ∘ gameTree) mine
```

Since *flatten* produces a breadth first traversal of the tree, we know that the result at the head of the list will have the shortest path. Furthermore, since the predicate applied is *isDone*, we know that the solution found is for a completed mine. Therefore, the head of this list will contain a solution with the maximal score for a completed mine: problem solved! Well, not quite.

While this strategy would eventually find such a solution for completable mines, it is prohibitively inefficient. In addition, since the tree is potentially very large, and not all mines are necessarily completable, an exhaustive search will generally not be possible. In order to solve this, we break the problem down into finding paths to a number of intermediate states given by some predicate: the basis for the searches will still be variations on breadth first search, but the goal is different. Rather than finding paths to different values of type *GameState*, we will seek values of type *GameTree*, so that we can search for new paths based on the returned tree, thus giving us more sophisticated searching strategies, where intermediate goals are reached and further analysis is performed on the trees that follow on from the paths to those goals.

A useful utility function along these lines is *findPaths*, which looks for paths to a particular coordinate:

```
findPaths :: GameTrie → Coord → [(Path, GameTrie)]
findPaths tree dest = bfs ((≡) dest ∘ robot ∘ gameStateMine) tree
```

This can be used, for example, to find a path to the *Exit* once the task of collecting all the lambdas is complete:

```
findExits :: GameTrie → [(Path, GameTrie)]
findExits tree = findTiles (≡ Exit) (layout (getMine tree)) >>= findPaths tree
```

This works by first finding the appropriate tile, and, if such a coordinate is found, then it is used by *findPath* to calculate a path.

At the heart of *findExits* is an efficient breadth first search algorithm, with a more general interface than that of *solve*. A naive breadth first search can be described as follows:

```
type KTrie k v = ([k], Trie k v)
bfsNaive :: (v → Bool) → Trie k v → [KTrie k v]
bfsNaive p tree = (filter (p ∘ root ∘ snd) ∘ stems) [([], tree)]
```

This makes use of the function *stems*, which is similar to *flatten*, but returns a list of paths with corresponding subtrees:

```
stems :: [KTrie k v] → [KTrie k v]
stems []                                = []
stems ((sk, t@(Trie _ kts)) : skts) = (reverse sk, t) : stems skts'
  where skts' = skts ++ [(k' : sk, t') | (k', t') ← Map.toList kts]
```

The *stems* function produces a breadth-first traversal of the tree, but is certainly not optimal: this function makes no effort to ensure that some common state has not been investigated several times: certain paths lead to exactly the same state, and we have no reason to assume that there will be any implicit sharing of these states.

3.3 Hashing

The lazy construction of the tree structure is unaware of any sharing which might could have been exploited between nodes that are equal. As a result, a search of the tree will likely result in repeated inspections of equal nodes and their children: this happens whenever there is more than one path to a particular state. To avoid this expensive recomputation, the breadth first search algorithm is modified to contain an accumulator that keeps a track of the nodes visited so far, and will not queue nodes whose values have already been visited elsewhere.

Rather than have the accumulator store the entire state of each visited mine, and have to perform an expensive equality operation, a hash of the mine is stored instead. We therefore extend the type of a *GameState* so that it contains a *Hash*:

```
type Hash = Int
data GameState = GameState { ...
                           , gameStateHash :: Hash }
```

An instance of *Hashable* is provided, giving us a means of obtaining the hash of a *Mine*:

```
instance Hashable Mine where
  hash mine = hash ((hash ∘ assocs ∘ layout) mine
                   , (hash ∘ robot) mine
                   , (hash ∘ moves) mine)
```

An accumulator, which is a set of hashes, is then added to the machinery of *stems* that allows states which have already been visited to be pruned:

```
stemsPrune :: Hashable v ⇒ Set Hash → [KTrie k v] → [KTrie k v]
stemsPrune _       []                                = []
stemsPrune visited ((sk,t@(Trie v kts)):skts) = case insertM (hash v) visited of
  Nothing      → stemsPrune visited skts
  Just visited' → (reverse sk,t):stemsPrune visited' skts'
  where skts' = skts ++ [(k':sk,t') | (k',t') ← Map.toList kts]
insertM :: Ord a ⇒ a → Set a → Maybe (Set a)
insertM x xs | Set.member x xs = Nothing
             | otherwise       = Just (Set.insert x xs)
```

The idea is to keep an accumulator that checks if the value of the tree being examined has been visited before. If it has been visited, then this value is rejected by the function *insertM*, and the next candidate for traversal is considered. If the value has not yet been visited, then the tree that contains it is added to the output of the search, its content is added to the set of visited values, and children are scheduled for traversal.

This lets us define a breadth first search that does not visit the same subtree twice:

```
bfsPrune :: Hashable v ⇒ (v → Bool) → Trie k v → [KTrie k v]
bfsPrune p t = filter (p ∘ root ∘ snd) ∘ stemsPrune Set.empty $ [([],t)]
```

The beauty of this solution is that it requires only the values *v* of the *Trie k v* structure to be *Hashable*.

Another performance issue is that *stems* uses a list to hold the queue of subtrees left to visit: the performance of appending to the end of a list is poor, and this can be easily improved by using a queue structure instead, and replacing the call to *stemsPrune* with an adequately instantiated call to *stemsPruneQ*.

```
stemsPruneQ :: Hashable v ⇒ Set Hash → Seq (KTrie k v) → [KTrie k v]
stemsPruneQ visited q = case Seq.viewl q of
  Seq.EmptyL                → []
  (sk,t@(Trie v kts)) :< q' → case insertM (hash v) visited of
    Nothing      → stemsPruneQ visited q
    Just visited' → (reverse sk,t):stemsPruneQ visited'
      (foldr (flip (|>)) q' [(k':sk,t') | (k',t') ← Map.toList kts])
bfsPruneQ :: Hashable v ⇒ (v → Bool) → Trie k v → [KTrie k v]
bfsPruneQ p t = (filter (p ∘ root ∘ snd) ∘ stemsPruneQ Set.empty ∘ return) ([],t)
```

This is a relatively straight-forward transliteration of the list based version into one that uses a *Seq* datastructure instead.

On a final note about pathfinding, the *findPaths* function takes a destination coordinate as an argument, and filters out results the results of a breadth-first traversal until a state is found where the robot is at the coordinate. A heuristic for possibly improving the search is by using a distance metric which determines how close a given point is to the destination, and using this information to give priority to certain elements within the queue. This is the basis of the well known A* algorithm, which is widely used in path finding and graph traversal.

To implement this algorithm, much of the structure present in *bfsPruneQ* can be reused, where *Seq* is replaced by a *MinQueue* structure which orders the elements according to some comparison function. For brevity, these details are omitted, but the development revolves around choosing an appropriate comparison function: a valid option would be to use the well-known Manhattan distance between two points, although there are other possible options. This function is then used to form the priorities of elements within the *MinQueue*, which arranges its elements so that those which are closest to the destination are favoured when considering the next value to explore in the search.

4 Robot strategy

Our solution relies on using several simple strategy algorithms competing for finding the best solution. A strategy takes a *GameTrie* and computes possible paths through the mine, together with their score:

```
type Strategy = GameTrie → [(Path, Score)]
```

We can now write a variation of the *solve* function (from Section 3) that produces a *Strategy* using *bfsPruneQ*:

```
solveS :: Strategy
solveS = map (second getScore) ∘ bfsPruneQ (const True)
```

This encodes the strategy of trying all possible paths, in a breadth-first manner. Naturally, this strategy is not very efficient, and will only work on very small maps. We also have a variant strategy that looks ahead only a number steps, and then takes one step along the best path found so far. This strategy finds locally optimal solutions.

An alternative strategy orders the remaining lambdas, tries to reach each one of them, and then walks towards the exit:

```
cmpS :: Comparison → Strategy
cmpS cmp tree
  | lambdas (getMine tree) ≡ 0 = case listToMaybe $ findExits tree of
      Just (p, tree') → [(p    , getScore tree')]
      Nothing         → [([A], getScore tree )]
  | otherwise = case pathToLambda cmp tree of
      []              → [([A], getScore tree)]
      ((p, tree'): _) → (p, getScore tree') : map (first (p++)) (cmpS cmp tree')
```

We omit functions *getMine* and *getScore*, simple accessors of the *GameTrie* data structure. Function *pathToLambda* takes a ranking function for lambdas and returns a list of paths:

```
pathToLambda :: Comparison → GameTrie → [(Path, GameTrie)]
pathToLambda cmp tree = concatMap snd (sortBy cmp dests)
  where dests = map (λcoord → (coord, findPaths tree coord))
                    (findTiles (≡ Lambda) ((layout ∘ getMine) tree))
```

We can now define multiple strategies simply by instantiating the comparison function of *cmpS*:

```
eqCmpS, lowCmpS, highCmpS :: Strategy
eqCmpS    = cmpS (λ_ _ → EQ)
lowCmpS   = cmpS (cmpCoords (λ(_,y) (_,y') → compare y  y'))
highCmpS  = cmpS (cmpCoords (λ(_,y) (_,y') → compare y' y))
```

Strategy *eqCmpS* treats all lambdas equally, while *lowCmpS* prefers lambdas located the lowest in the mine. This strategy might make sense when the lower parts of the mine become harder to access as time goes by (see Section 6.1).

We also have more complicated strategies involving *cmpS*, such as preferring lambdas that are part of large clusters.

5 Concurrency and exception handling

Strategies turn the representation of a game tree into a lists of paths with their corresponding score. By sharing the game tree structure, a number of concurrent worker threads using different strategies can compete with one another to find an optimal solution. The communication between these threads occurs through the use of Haskell's *MVar* values: these are mutable variables which can be shared and synchronised between threads. The task of each worker is to improve this solution with whatever they might encounter in their list of candidate answers.

```
improve :: (Ord s, NFData s, NFData a) ⇒ MVar (a,s) → [(a,s)] → IO ()
improve mvBest = mapM_ (λx → x `deepseq` modifyMVar_ mvBest (cmpBest x))
  where cmpBest x best = return (if snd x > snd best then x else best)
```

Here, each solution x is a tuple of type (s,a), where s is a score that will be maximised, and a the answer itself. We require s and a to have an *NFData* instance to be able to force evaluation using *deepseq*, since the entire computation of the value of x should occur before blocking on the *mvBest* variable. The *MVar* is a reference to the best solution found so far; *improve* updates this *MVar* whenever a better solution is found. As this worker might be interrupted before the list is fully evaluated, it is important that *modifyMVar_* is an atomic operation: if the worker raises an exception while it is modifying *mvBest*, then the value is restored to its original state.

The workers are spawned by *spawnWorkers*, which creates a new asynchronous thread for each of the answers returned by the strategies, and then waits for all the threads to finish.

```
spawnWorkers :: (Ord s, NFData s, NFData a) ⇒ MVar (a,s) → [[(a,s)]] → IO ()
spawnWorkers mvBest xss = do workers ← mapM (async ∘ improve mvBest) xss
                             mapM_ waitCatch workers
```

An important feature of this function is that the failure of one worker does not affect the others, since *waitCatch* will silently ignore any worker which has thrown an exception. While deceptively succinct, these two functions provide a powerful mechanism by

which multiple concurrent workers can be spawned to improve the value of a solution, all the while dealing with exceptions in a safe way by allowing the best known solution to prevail in the case of failure.

Since we can rely on the fact that the best solution will not be lost when the workers fail, we can make use of this mechanism to allow the system to demand an immediate answer at any point during the computation. This fits nicely into the framework of the contest, where programs are given a set amount of time within which to find a solution, and then given a signal which raises an exception when time is up and an answer is required. To exploit this, the function *run* is used, which spawns the workers to perform the task of finding the best solution, and provides a callback that should be executed whether the computation terminates naturally, or an exception is thrown.

```
run :: (Ord s, NFData s, NFData a) ⇒ (a, s) → [[(a, s)]] → ((a, s) → IO ()) → IO ()
run best xss callback = catchUserInterrupt $
  bracket (newMVar best)
          (λmvBest → takeMVar mvBest >>= callback)
          (λmvBest → spawnWorkers mvBest xss)
```

The function $bracket :: IO\ a \to (a \to IO\ b) \to (a \to IO\ c) \to IO\ c$ takes three arguments: the initial computation, which initialises the best result found so far, the final computation, which reads the best result found and calls the callback, and the intermediate computation, which spawns the workers and waits for all threads for finish. The final computation of a *bracket* is performed even if an exception is raised, which is precisely the behaviour required here when the callback is an action which outputs the best known solution.

One problem remains: if an exception is raised within a *bracket*, then after the final computation has been executed, the exception will be re-raised so that it can be handled elsewhere in the system. If left unhandled, the program would exit and indicate that there was an error. The *catchUserInterrupt* function is a helper which allows the program to gracefully exit when the interrupt signal which is expected from the judging environment is received.

```
catchUserInterrupt :: IO () → IO ()
catchUserInterrupt = handle (λe → case e of UserInterrupt → return ()
                                            _             → throwIO e)
```

Note that if the exception received is not one that is expected, then the exception is thrown again and allowed to propagate further.

For testing purposes it is convenient to be able to kill worker threads after a particular amount of time, in order to simulate the judging environment. This is implemented using the *timeout* function which runs an *IO* computation within a thread and kills the thread if no result is returned within a given time limit.

```
runWithTimeout :: (Ord s, NFData s, NFData a)
                  ⇒ Int → (a, s) → [[(a, s)]] → ((a, s) → IO ()) → IO ()
runWithTimeout t best xss callback = timeout t (run best xss callback) >> return ()
```

This works as expected since exceptions are used to kill a thread that has expired.

6 Changing specifications

One of the challenges was to deal with changing specifications. This was very easy to cope with in our model, and only minor extensions were required, mostly confined to the *Mine* and *Tile* datatypes, and the *stepRobot* and *updRocks* functions.

6.1 Flooding

The first extension was to add flooding to the mines. In certain maps, there is a rising level of water. The robot operates normally underwater, but it gets destroyed if it spends too many turns underwater. Modelling flooding requires changing the *Mine* data structure, extending it to contain additional information:

```
data Mine = Mine { ...
                 , flood      :: Int
                 , waterproof :: Int
                 , water      :: Int }
```

These fields store the rate of flooding, how long the robot can last underwater, and the current level of water.

6.2 Trampolines

The second extension introduces trampolines, which act like teleporters. Once entering a trampoline, the robot gets instantly moved to a fixed destination location, and the trampoline disappears.

Similarly to flooding, trampolines requiring adding extra information to the *Mine* data structure:

```
data Mine = Mine { ...
                 , trampolines :: Set Coord
                 , targets     :: Set Coord }
```

These fields store the current position of trampolines and their associated targets. Additionally, the *stepRobot* function has to consider the case of moving into a trampoline, and we need two new tile types: trampolines and targets.

6.3 Beards and razors

The third extension introduces beards. Beards are a new type of tile, that expand into the surrounding empty spaces in a fixed number of turns. The robot cannot traverse beards, but can collect and apply razors, which eliminate all beards surrounding the robot.

Again, the *Mine* structure has to be extended, this time with a growth factor and the number of available razors:

```
data Mine = Mine { ...
                 , growth :: Int
                 , razors :: Int }
```

Two new tile types are added (beard and razor). A new robot "movement" is to apply a razor, and the *updRocks* function now needs to update the tiles adjacent to beards as well.

6.4 Higher order rocks

The last extension introduces higher order rocks, which are rocks that upon impact (from falling) transform into a lambda. Each higher order rock counts as a lambda for the purpose of determining whether all lambdas have been collected.

We add a second Boolean to the *Rock* constructor to distinguish higher order rocks from normal rocks:

$$\textbf{data}\ \mathit{Tile} = \ldots \mid \mathit{Rock}\ \mathit{Bool}\ \mathit{Bool}$$

The *updRocks* function now treats higher order rocks just like ordinary rocks, apart from a small special case to check if a higher order rock should be transformed into a lambda. Additionally, the calculation of the number of lambdas after a step ($lambdas'$ in Section 2.2) becomes more complicated. Two falling rocks can fall into the same spot, with one disappearing. If the rock that disappears is a higher order rock, then there is one fewer lambda in the mine. For simplicity, we calculate the number of remaining lambdas by traversing the entire layout:

$$lambdas' = length\ \$\ findTiles\ (\lambda t \to t \equiv Lambda \vee isRockLambda\ t)\ layout''$$

7 Conclusion

We have described our solution to the 2012 ICFP programming contest, and seen how Haskell's features are useful during fast paced prototyping. Both low-level features (such as concurrency and exception handling) and high-level features (such as purity and laziness) are important to our development, and Haskell has reached an excellent maturity status, both with regard to its and features as well as to the quality of libraries available. We now give some general advice for code development in similar situations, based on our experience, and reflect briefly on possible improvements to our solution.

7.1 Practical guidelines

Testing Even though Haskell's strong type system prevents many common programming mistakes, we still had many bugs in our code. In particular, our submitted version often returns rather poor solutions because of bugs in the simulator. We focused our development in supporting the extensions and improving the strategies, but it would have been more effective to find and eliminate bugs.

Communication Our team was split into two groups in different locations. We found that frequent short meetings were helpful to keep the team up-to-date with the whole development, while allowing individual team members to work on separate parts of the program. Video communication, and screen/application sharing is useful for distance communication, but whiteboard brainstorming is invaluable, and hard to mimic in a distance communication.

Model first We started developing our solution by writing the model (Section 2.1). With this in place, different team members could develop the surrounding infrastructure more or less independently. Changes to the model were discussed with everyone before being implemented, and applied as soon as possible. This helped to minimise the mismatch between different components, and to allow development in parallel effortlessly.

Pair programming We have alternated our development between whole team discussion, individual coding sessions, and pair programming. We found pair programming to be an effective way of coding the more challenging parts of our solution, with the advantage that both team members become familiar with the code.

With regard to possible improvements to our solution, while the pathfinding algorithms take care to avoid going back to the same state several times, it would be nice to have this built into the tree structure itself. However, this would mean not using a tree structure, but rather some kind of directed graph. The lazy construction of such a graph requires the use of an appropriate constructor function to be called when elements are missing in a node lookup. The details of such an implementation are beyond the scope of this paper.

Bibliography

Hinze, R. (2000). Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351.

Hughes, J. (1989). Why functional programming matters. *The computer journal*, 32(2):98–107.

Peyton Jones, S., editor (2003). *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press. Journal of Functional Programming Special Issue 13(1).

*De*composing Metaheuristic Operations

Richard Senington and David Duke

University Of Leeds, Leeds LS2 9JT, UK
sc06r2s,d.j.duke@leeds.ac.uk

Abstract. Non-exhaustive local search methods are fundamental tools in applied branches of computing such as operations research, and in other applications of optimisation. These problems have proven stubbornly resistant to attempts to find generic meta-heuristic toolkits that are both expressive and computationally efficient for the large problem spaces involved. This paper complements recent work on functional abstractions for local search by examining three fundamental operations on the states that characterise allowable and/or intermediate solutions. We describe how three fundamental operations are related, and how these can be implemented effectively as part of a functional local search library.

Keywords: Search, Optimisation, Stochastic, Combinatorial

1 Introduction

Metaheuristics (also known as local search) refer to a collection of methods for tackling combinatorial problems which are ubiquitous in areas including the sciences, engineering, economics, business and logistics [1]. These methods stand in contrast to "exhaustive" (global) search (such as Branch & Bound (B&B) which explicitly and/or implicitly examine all candidates in the solution space), in that they do not guarantee to find an optimal solution to the problem.

In many tasks however time is limited, and finding higher quality solutions is more important than finding a provably optimal solution. When problem sizes become large enough global methods are unable to complete in practical time limits and in these cases metaheuristics have been shown to give better solutions to the same problems in practical time bounds.

Combinatorial problems of significant size, and of particular interest to metaheuristic research, are often derived from real world problems. The development of metaheurisitcs to tackle these often leads to specialised hybrids which can include exhaustive search methods as components[2,3].

Toolkits to aid in metaheuristic design and research have been created, however they are complex, frequently (though not by design) obscuring their inner workings. By analogy with crafting wood, these tools are adequate when one works 'with the grain' defined by the tools' abstractions and interfaces, but their limitations[4] are painfully exposed when a new problem requires working 'against the grain'.

One solution is to build tools from finer-grained components, and we have previously argued [5] that functional abstractions provide a powerful substrate for developing metaheuristics from combinators. Functional languages like Haskell, and implementations like the Glasgow Haskell Compiler, contribute layers of mechanism for the translation of these high level abstractions into efficient low level code, thus helping to resolve the tension between expressiveness and efficiency.

This paper contributes a substantial case study in abstraction design and implementation, providing the first account of how low level local search operations can be effectively implemented in a functional setting.

The paper is structured as follows. Section 2 provides a brief description of the library that is used in this paper, and how the low level operations for interaction with combinatorial problems interact with it. Section 3 reminds the reader of the Travelling Salesperson Problem, a well known and intensively studied problem that is used as an example in this paper. Section 4 describes perturbation and neighbourhood methods, and shows how higher order functions can facilitate the easy conversion between these two

Section 5 decomposes Perturbation methods, providing finer grained operators for the design of low level interactions with problems. Section 6 describes Recombination methods and shows how they have elements in common with some perturbation methods. Section 7 Uses the decomposition to investigate a broad range of different perturbation operations for the TSP. Section 8 details conclusions further work.

2 Combinators for Metaheuristics

This paper uses an experimental library [5] for the expression of metaheuristic algorithms in the pure functional language Haskell. Components of metaheuristics are expressed as transformations over *streams* of values, usually solutions to problems. These components are built, managed and manipulated by a library of combinators. The stream transformations are *looped* and given seed data, to create list generators that can be examined to yield the results of the metaheuristic search.

For example, a first-found iterative improver is the repeated selection of the first element from a series of neighbourhoods which are transformed so that they always improve upon their seed. The concept of an *improving* neighbourhood is captured using a higher order function called *improvement* which transforms a neighbourhood function. The selection of the *first* element, is simply *map head*, from the standard Haskell libraries, and using this would give a *First Found Iterative Improver*. A completed and looped first-found iterative improver can be defined at the ghci command-line by the expression below, where *nF* is the problem specific neighbourhood function, and *seed* is an initial solution for the program to explore from.

$$ffii\ nF = map\ head \circ improvement\ nF$$

```
> loopP ffii nF seed
```

The library frequently uses finite lists to represent collections or groups, but also uses lazy lists to represent streams of unlimited length. This can cause confusion, so to distinguish between the uses of lists we define the following type synonyms:

$$\textbf{\textit{type}}\ \mathit{Stream}\ a = [a]$$
$$\textbf{\textit{type}}\ \mathit{Group}\ a = [a]$$

The various well known metaheuristic algorithms (Iterative Improvement, TABU search, Simulated Annealing, Genetic Algorithms) all interact with their problems through one of the three basic operations. Within the functional toolkit, these generic operations will be expressed as forms of stream transformation:

perturbation :: $\mathit{Stream}\ s \rightarrow \mathit{Stream}\ s$ — a single solution is changed to yield a different, but similar, solution. This can be seen in algorithms such as random walk and simulated annealing.

neighbourhood :: $\mathit{Stream}\ s \rightarrow \mathit{Stream}\ (\mathit{Group}\ s)$ — a single solution is used as the seed to generate a collection of similar solutions. This can be seen in iterative improvement and TABU search.

recombination :: $\mathit{Stream}\ (\mathit{Group}\ s) \rightarrow \mathit{Stream}\ s$ — a collection of solutions are merged in some way to yield a new solution sharing characteristics of the *parents*. Genetic algorithms are the classic example of this type.

We will explore higher order functions to aid in the translation between these different classes of interaction function. Problem-specific specialisations are then used to realise these operations to in metaheuristic algorithms, providing the low level interactions with the problem data. For example, the first-found iterative improver has the following type;

$$\mathit{ffii} :: \mathit{Ord}\ s \Rightarrow (\mathit{Stream}\ s \rightarrow \mathit{Stream}\ (\mathit{Group}\ s)) \rightarrow \mathit{Stream}\ s \rightarrow \mathit{Stream}\ s$$

3 Example Problem: TSP

The Travelling Salesperson Problem (TSP) is one such combinatorial problem, which is often used as an example and test problem. The TSP is defined as finding a Hamiltonian cycle (a tour of a graph going through each node exactly once), of minimum *cost*, in a complete graph, where the cost of each edge of the graph is known.

A TSP may be symmetric, where the cost of an edge is the same whichever way it is traversed, or asymmetric where this constraint does not necessarily hold. This paper will use the TSP as the example problem for illustrative purposes. Other combinatorial problems will also be mentioned.

Throughout the paper we will assume that there is a data type called *TSP*, which supports equality testing and ordering based upon the relative quality of the solutions. Internally this can be thought of as a String, with special extensions. It will be defined in the following form;

$$\textbf{\textit{data}}\ \mathit{TSP} = \ldots$$

4 Perturbation & Neighbourhoods

Both perturbation & neighbourhood functions can be defined in a specialist, monolithic way for any given problem. A common example neighbourhood function for the TSP is the *adjacent swap* neighbourhood. In this function a set of new solutions are defined as the exchange of adjacent cities in the original, e.g.

abcde...
→ **ba**cde...
→ a**cb**de...
→ ab**dc**e...
⋮

We will assume a function called *swap*, which works with the previously defined *TSP* data type. *Swap* will take two indices and an instance of a TSP solution and returns a new TSP solution with those indices swapped. The adjacent exchange function, taking the number of cities as a constant parameter, can then be implemented as follows:

```
adjNeighborhood :: Int → Stream TSP → Stream (Group TSP)
adjNeighborhood nCities = map (λt→map (λi→swap i (i+1) t) [0 .. nCities])
```

However this hides a general relationship between perturbation and recombination, which permits each to be described in terms of the other:

- Perturbation is the selection of one element from a neighbourhood; and
- Neighborhood is the application of perturbation to a single solution many times and gathering up the results.

4.1 Neighbourhood to Perturbation

A neighbourhood function can be adapted to become a perturbation function through the composition of the function with some form of selection function. The selection function, operating over streams, will have the type[1]:

```
type Selection a = Stream (Group a) → Stream a
```

The methods that may be used for selection are numerous and fall into two major categories:

Deterministic: such as selecting the *first, last, maximum or minimum* valued solutions from each neighbourhood. Of these, *first* might be used because in combination with lazy evaluation it will limit the runtime requirements of the program; where as *minimum* might be used to move towards a local minima in the shortest number of iterations. Deterministic operations can be lifted to operate over streams using the standard *map* function.

[1] Note that this is also the type of recombination, so any recombination method could be used at this point, if it was felt that it was appropriate to do so.

Stochastic: while uniform likelihood selection is the most obvious concept here, other options include stochastic selection with varying likelihood based upon quality of the solutions in the underlying group. A function with the type:

$$System.Random.Random\ r \Rightarrow r \rightarrow Group\ s \rightarrow s$$

may be lifted to operate over streams using the *zipWith* function e.g.

```
> zipWith selectFunction (randoms g) :: Selection a
```

4.2 Perturbation to Neighbourhood

The repeated application of a perturbation operation to elements of an underlying stream, and the subsequent collection of these results into a group can be achieved using a function called *doMany* from the local search library. This is defined as follows:

```
doMany :: Int → (Stream b → Stream s) → Stream b → Stream (Group s)
doMany n f = chunk n ∘ f ∘ stretch n
```

The *doMany* combinator works by duplicating the underlying elements creating a stream that is n times longer than the original (*stretch*), and when the function f is applied to this it is equivalent to applying it many times to each value in the underlying stream. *chunk* is then used to divide the output of this process into a stream of regularly sized blocks, gathering the results back together into a new group.

Using *doMany*, different forms of neighbourhood can be created from a single perturbation function. For example, using the *swap* function for TSP,

– a *deterministic* neighbourhood which performs the same operation on each seed can be created by cycling a specific pattern of cities to be exchanged, for example:

```
tspDNF :: [(Int,Int)] → Stream TSP → Stream (Group TSP)
tspDNF p
  = doMany (length p) (zipWith3 swap (cycle pA) (cycle pB))
    where (pA,pB) = unzip p
```

This allows the previous *adjNeighborhood* for TSP to be implemented as

```
adjNeighborhood nCities = tspDNF (zip [0..nCities] [1..nCities])
```

– since any perturbation transformation may be used as a parameter for *doMany*, a *stochastic* neighbourhood is actually a generalisation, relaxing the source of the city indices to be swapped into streams in their own right;

```
tspSNF :: Int → Stream Int → Stream Int
                →Stream TSP→Stream (Group TSP)
tspSNF nSize ns ms = doMany nSize (zipWith3 swap ns ms)
```

The streams of integers are providing the cities to be exchanged and are assumed to be stochastically generated.

5 Decomposition of Perturbation

The swapping operation for TSP, which has been used so far as the most basic operation in these examples, is known not to be particularly effective. A better method is to model a solution as a collection of edges, rather than a sequence of cities, and make changes by deleting edges and then reconnecting the resulting fragments. The swapping of cities can be seen as a very restricted subset of this model, where the cities being swapped determine exactly which edges are to be removed and inserted.

More generally this gives rise to two different activities, which when paired give rise to a perturbation technique, *damage and repair*. In this general pattern the damage phase removes something from the solution, leaving a data structure that is no longer a valid or complete solution to the combinatorial problem. The repair phase is then required to create a completed solution from the incomplete data structure.

The *damage/repair* model of perturbation is also more effective when considering more general models of combinatorial problems than the specific TSP example, for example when modelling problems using constraints. A constraint model provides a solution as a collection of constraints, generated through a constructive search process. Once a completed solution is achieved, what does it mean to *swap* or otherwise make changes to the constraints. In many cases arbitrary changes to the constraints will make them inconsistent, however using a damage repair model to define methods for deleting and reconstructing from the remaining constraints can give rise to effective algorithms.

5.1 Damage Methods

The characteristics of damage methods fall along two categories and a rough diagram of how these overlap can be seen in Figure 1. All decisions in the damage method are made with respect to some problem specific heuristic, for example in the TSP edge length is a simple heuristic for evaluating the quality or usefulness of any give edge. A decision can be made in an entirely stochastic way, ignoring this underlying heuristic (usually resulting in each edge having a *uniform likelihood* of selection), or can be made with no stochastic element, resulting in a *most likely* or *greedy* deletion method. Between these extremes is an approach where decisions involve a stochastic element, but it is biased with respect to the heuristic, so that worse decisions are less likely. For example, in the TSP the selected edges could be ordered by length and then selected from based upon a probability distribution.

On the other axis is how the scale of damage to be done will be selected. At the deterministic end is a fixed level of damage, for example three or six edges to be deleted from each solution. At the stochastic end is that any number of edges can be deleted and how many will be chosen with uniform likelihood.

In the centre of Figure 1 is a situation where each decision is made stochastically, but with a reasonable respect for the heuristic. Each decision is independent of the others, so that any number of edges might be deleted, but it is unlikely it

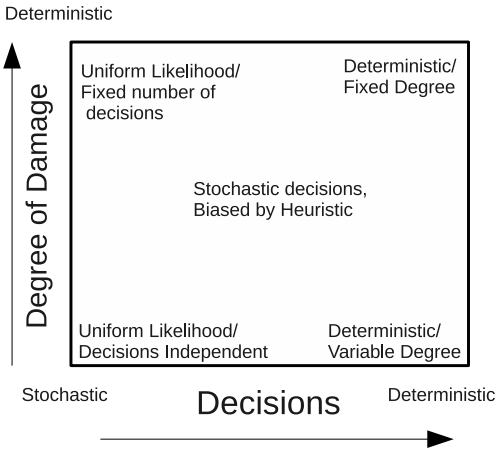


Fig. 1. Characteristics of damage methods

will be all, and the number is dependent upon the qualities of the edges in the solution at the time.

Implementation of the damage methods are required to operate over streams, giving rise to a transformation from a stream of solutions, to a stream of damaged solutions. Internally stochastic elements and logic can be threaded using *zipWith* and *map* as with the previous combinators.

5.2 Repair Methods

Unlike damage methods, repair methods cannot easily have a *degree* of repair, so there is only one spectrum, the level of stochastic computation involved in each decision. Like damage methods a heuristic is used to guide decisions, and at one end of the spectrum is uniformly random likelihood of any *legal*[2] decision being made, at the other a greedy heuristic.

Repair methods can make use of a further style of operation, exhaustive search. Due to the level of repair usually being limited, exhaustive methods such as *Branch & Bound* can be used with confidence that they will complete. This can be seen as a variant on a neighbourhood, where a number of solutions are considered, and only the best is accepted, however it is more simply defined at this time as a separate operation, rather than breaking it down into the generation of solutions and selection.

In section 4.2 it was seen how neighbourhoods could be created through the repeated application of a perturbation operation. The decomposition of perturbation operations allows for an alternative form, where damage is carried out only once, and a stochastic repair procedure is then used several times to yield

[2] An illegal decisions would result in an invalid solution, for example sub-loops in a TSP.

a neighbourhood. The reverse of this, where damage is carried out many times and then each is repaired is the equivalent of a neighbourhood built from a perturbation operation.

6 Recombination

As with neighbourhoods and perturbation recombination can be defined monolithically, and often is. When considering problems such as Boolean Satisfiability (SAT), a simple recombination method is to cut two strings at the same point and concatonate the substrings, for example see Figure 2. However such an ap-

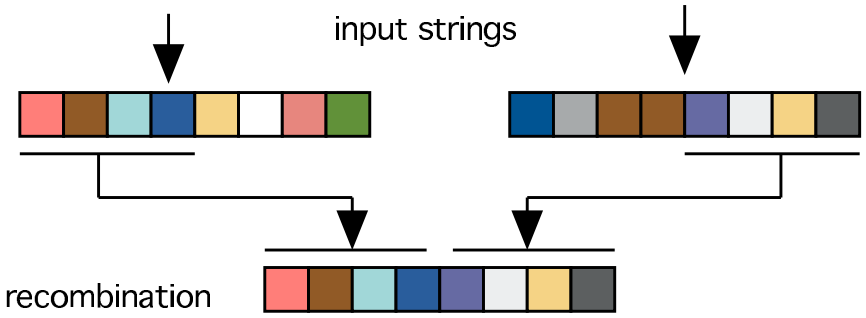


Fig. 2. Illustration of a simple crossover mechanic in SAT

proach is not effective for the TSP, because it tends to result in duplicated cities. This issue can be fixed by creating the second part of the solution through filtering the second solution, removing any city found in the first part created through cutting the first solutions string, hence preserving some sense of the order of the original solutions.

However, as with perturbation methods, the most effective recombination algorithms for TSP consider the solutions in terms of the edges they use, rather than the order of the cities. The most effective genetic algorithm for the TSP [6], made use of a recombination method that was *maximally respectful* of the edges in the parent solutions. This meant that it identified edges common to both parents and ensured that they were present in the new solution. Each other edge in each parent was then selected probabilistically, on the condition that the result was not invalid. Any final gaps were filled in using a greedy heuristic.

These examples of the TSP and SAT do not provide guidance on creating recombination techniques for other problems. For example, the recombination of a problem modelled using constraints, where constraints are simply selected from the parents, once again runs a considerable risk of having irreconcilable conflicts. The final example of TSP recombination, using a greedy heuristic to

complete a solution legally does however provide inspiration for an approach which can be explored.

The process of recombination can be described as following a pattern of analysis of the parents, followed by the construction of a new solution based upon the analysis. The construction process has a strong similarity to repair concepts seen in perturbation, as in the TSP example already seen. This suggests that analysis forms a new class of operations, but that the repair operations can be reused.

7 Which perturbation algorithm?

The *No Free Lunch Theorem*[7] says that there is no one metaheuristic, nor perturbation method that is best for all problems. So it is of value to be able to experiment upon specific problems and see how different perturbation algorithms compare.

This paper has proposed a collection of building blocks, specifically related to the TSP, which may be used to construct perturbation methods. To demonstrate their effectiveness, and requirement with relation to the No Free Lunch Theorem, we built a simple program to test a number of perturbation methods in the context of a single problem and metaheuristic. The problem chosen was *fl417* a symmetric TSP problem drawn from the TSPLIB[8] and the metaheuristic was simply the repeated application of the perturbation method to the last solution seen.

The program combined various damage and repair elements to generate different perturbation methods. In the event that uniform damage and uniform repair are used, this results in a form of random walk of the solution space. Damage levels of three edges and six edges were used, to compare the result when different degrees of damage occurred, and how this affected the performance of each metaheuristic. The results of each metaheuristic were processed to preserve only the best solution seen at that point, and were sampled at 10,000 iterations. Each test was run 25 times and the average is presented here.

Geo: a curved distribution, where the heuristically best choice is most likely, and it reduces in likelihood at each choice.

Uniform : an entirely uniform likelihood of any choice being made.

Most Likely: a deterministic greedy algorithm

Exhaustive: an exhaustive repair operation

BU 0.05: is an irregular distribution, where the heuristically best choice has a likelihood of 0.05, but if not chosen a choice is made from the remainder with uniform likelihood.

3 edges

Repair	Damage			
	Geo	Uniform	Mostlikely	BU 0.05
Geo	32208	182563	171819	151743
Uniform	31506	191036	174718	155698
MostLikely	32232	85423	175227	53355
Exhaustive	34381	32413	188427	25426

6 edges

Repair	Damage			
	Geo	Uniform	Mostlikely	BU 0.05
Geo	25280	183383	63284	150839
Uniform	25413	194835	61543	157941
MostLikely	24532	83966	56093	54676
Exhaustive	29178	19627	253509	16728

These results exhibit some broad patterns which are consistent the expected characteristics of the combinations, but also some interesting diversity worth deeper consideration. The worst results are seen in algorithms which are highly stochastic, such as uniform likelihood of damage and repair, or purely deterministic such as greedy damage with exhaustive repair. This is correct for both three and six edge experiments, however there is a particularly interesting result, the high weakness of greedy damage, combined with exhaustive rebuild over this shift. This suggests that, rather than the increased size of damage improving performance through widening the options that might be considered, for this algorithm the change causes it to more rapidly find and become stuck in a local minima.

The geometric damage patterns perform consistently well at both levels, with any form of repair technique, however the best perturbation method uses the Biased Uniform damage strategy, with an exhaustive repair technique. This pattern of successful algorithms also supports common wisdom, that the best results come from a careful marriage of the level of damage, stochastic and deterministic components. However these results also show the significant variances that occur as components are exchanged, and how other parameters, such as the level of damage can change the apparent performance of particular combinations on particular problems. This all supports the idea that flexibility and ease of experimentation are important characteristics of any library or toolkit for metaheuristic implementations.

To further investigate the use of these combinators a test was built using a Set Covering problem, drawn from train scheduling algorithms. Integer Linear Programming (ILP) provides the most effective tool for tackling these problems, providing the best known solutions though it suffers the the usual limitations of an exhaustive search, that it cannot complete for most problem instances.

The Hypermutation metaheuristic[9] is also known to give interesting results, though not actually able to compete with ILP. Hypermutation works through an iterated perturbation, where the perturbation is achieved through the composition of a stochastic damage method, biased towards heuristically selected el-

ements of a solution and a greedy repair strategy. An exhaustive repair strategy has not, to the authors knowledge, been tried in the context of Hypermutation.

The TSP experiments had suggested that using an exhaustive repair strategy, in combination with a highly random damage strategy should provide the best results. Rather than relying upon a heuristic measure to bias the stochastic selection of covering components, a purely uniform damage strategy was used.

Repair was carried out using an ILP solver, with the elements of the previous solution designated for preservation *fixed* in the constraints of the ILP model[3]. This more constrained problem could be completed at each iteration, though a search on the problem instance in general would not complete (though would yield some solutions). The ILP system used was the GNU Linear Programming Kit Version 4.25[4] and linked to Haskell using *glpk-hs*[5],

The solutions found were good, superseding the Hypermutation previously described, and over runtimes of between 15 minutes and 2 hours the metaheuristic gave stronger results than the ILP method alone over the same time limit. We were pleased that the use of concepts from the previous TSP study gave such promising results, however they do not equal results yielded from a commercial solver, based upon ILP methods using specialised extensions.

8 Conclusion

This paper has examined perturbation, recombination and neighbourhood methods, used as the low level interaction operations in metaheuristics, from a purely functional perspective. This has resulted in the creation of combinators for moving between these various methods and the decomposition of the monolithic functions into three alternative classes of function; analysis, damage and repair. A number of types and variations upon each these have been proposed.

This decomposition into smaller building blocks makes visible a broad range of alternative perturbation, neighbourhood and recombination methods, through picking and choosing from the options available. The visibility of the elements being used in each composition allows for clearer comprehension of how they interact and how larger methods operate, aiding in the design of new variations.

To demonstrate this a short investigation of a specific TSP problem has been shown, mixing and matching a range of both well known and less frequently seen operations, yielding some useful results. We see that the right set of abstractions, here as elsewhere, can provide powerful tools to aid in the investigation of problems and the construction of algorithms.

Haskell's expressiveness aids in these forms of investigation, with the type system providing clues and pointers as to how components may be combined. This in turn proposes lines of investigation, sometimes unconsidered, or shows

[3] Other uses of ILP as a component in the construction of metaheuristics may be seen in[2,3]

[4] GLPK may be found at `http://www.gnu.org/software/glpk/glpk.html`

[5] glpk-hs is written by Louis Wasserman and may be found in the Haskell libraries at `http://hackage.haskell.org/package/glpk-hs`

where more sophisticated conversion techniques will be required to facilitate a desired line of research.

The next stage in this investigation is the further hybridisation of these operations. At present damage and repair alternatives have been created, but a more complex approach might use a number of damage and repair strategies in a single perturbation method. For example, fixing one part of the solution using a greedy method, another part of a solution using a uniform likelihood and finally an exhaustive technique. This suggests a new range of combinators that can be explored to improve the expression of these hybrids.

We see this work as moving in the direction of superior methods for investigating metaheuristic methods, and automated experimentation through combining well understood building blocks. This places the work in the realm of hyper-heuristics[10], a branch of metaheuristic research that attempts to automate the design of algorithms for specific problems.

References

1. Hoos, H., Stützle, T.: Stochastic Local Search: Foundations & Applications. Morgan Kaufmann Publishers Inc. (2005)
2. Contardo, C., Cordeau, J.-F., Gendron, B.: A grasp + ilp-based metaheuristic for the capacitated location-routing problem. Technical report (2011)
3. Prins, C., Prodhon, C., Ruiz, A., Soriano, P., Calvo, R.W.: Solving the capacitated location-routing problem by a cooperative lagrangean relaxation-granular tabu search heuristic. Transportation Science **41**(4) (November 2007) 470–483
4. Masrom, S., Siti, A.Z., P.N., H., Rahman, A.A.: Towards rapid development of user defined metaheuristics hybridisation. International Journal of Software Engineering and Its Applications **5**(2) (2011)
5. Senington, R., Duke, D.: Combinators for meta-heuristic search. Submitted for Publication (2012)
6. Merz, P., Freisleben, B.: Memetic algorithms for the traveling salesman problem. Complex Systems **13**(4) (2001) 297–345
7. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimisation. IEEE Transations on Evolutionary Computation **1**(1) (1997) 67–82
8. Reinelt, G.: TSPLIB - A Traveling Salesman Problem Library. INFORMS Journal on Computing **3**(4) (1991) 376–384 http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.
9. Li, J., Kwan, R.S.K.: A fuzzy genetic algorithm for driver scheduling. European Journal of Operational Research **147**(2) (2003) 334–344
10. Burke, E., Hart, E., Kendall, G., JimNewall, Ross, P., Schulenburg, S.: Hyper-heuristics: An emerging direction in modern search technology. In: Handbook of Metaheuristics. International Series in Operations Research & Management Science. Kluwer (2003) 457–474

Rational Term Equality, Functionally

Tom Schrijvers[1] and Bruno C. d. S. Oliveira[2]

[1]Universiteit Gent, Belgium
tom.schrijvers@ugent.be
[2] National University of Singapore
oliveira@comp.nus.edu.sg

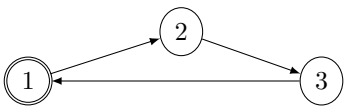
Abstract. This paper presents an elegant purely functional algorithm for deciding the equality of rational terms. The algorithm adapts Hopcroft and Karp's classic algorithm for equality of finite state automata to structured graph representation of rational terms.

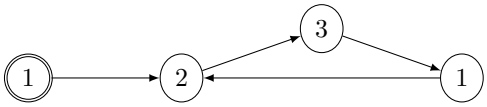
1 Introduction

Rational terms are infinite terms that are finitely representable. They consist of a finite number of distinct (possibly infinite) subterms. The finite representation of choice are *graphs*, where infinity is captured in cycles.

Rational terms have many applications in logic and functional programming languages without explicit pointers: graphics [1], parser generation and grammar manipulation [2, 3], finite-state automata [2], natural language processing [4–6], interpreters for control constructs [7], (equi-)recursive types [8], ...

The following two cyclic graphs both denote a rational term, a stream:





The infinite rational terms are recovered by unfolding the graphs from their root, marked with a double circle. When doing so, despite not being structurally equal, the two graphs yield the same infinite stream $[1, 2, 3, 1, 2, 3, ...$

However, it is clearly not pratical to fully compare two infinite terms to decide equality. As far as we can tell, Hopcroft and Karp [9] were the first to propose an efficient algorithm to decide the equality of two rational terms directly from their finite representations. Their algorithm is formulated in terms of a slightly different finite guise of rational terms, finite state automata. More recent imperative incarnations of their algorithm deal with pointer-based representations of graphs and rely on pointer equality.

While graphs are pervasive in computer science, the efficient representation of graphs and elegant implementation of graph algorithms are notoriously hard in pure Functional Programming. A primary reason is that explicit pointer-based

representations and pointer equality cannot be used, as they violate *referential transparency*.

This paper presents an elegant purely functional implementation of Hopcroft and Karp's algorithm for computing the equaltiy of rational terms from their graph representations. It is based on Oliveira and Cook's purely functional representation of structured graphs [10]. The key step in our approach is to infinitely unfold the finite representation of the structured graph, but to tag each unique node with a pointer-like identifier.

2 Generic Structured Graphs

We base ourselves on the new functional representation of rational terms as structured graphs by Oliveira and Cook [10]. This representation employs lightweight *datatype-generic programming* [11] techniques to be generic in the particular structure of the graph; this genericity of representation lends a generality to our approach.

The representation is based on binders: the cycles in the graph are denoted by greatest fixpoints of functions, where the binder denotes one end of the cycle and an occurrence of the bound variable is the other end. The particular binder representation used is parametric higher-order abstract syntax (PHOAS) [12].

$$\begin{aligned}&\textbf{data}\ \mathit{Rec}\ f\ a = \\ &\quad \mathit{Var}\ a \\ &\quad \mid \mathit{Mu}\ ([a] \to [f\ (\mathit{Rec}\ f\ a)]) \\ &\quad \mid \mathit{In}\ (f\ (\mathit{Rec}\ f\ a))\end{aligned}$$
$$\textbf{newtype}\ \mathit{Graph}\ f = \downarrow \{\uparrow :: \forall\ a.\mathit{Rec}\ f\ a\}$$

The representation separates the datatype-specific parts of structured graphs (the *In* constructor) from the generic binding infrastructure (the constructors *Var* and *Mu*).

Generic Structure Support The datatype-specific parts are captured in the f type parameter: f is the *pattern* functor of the recursive datatype. The *In* constructor takes the fixpoint of this functor. Note that, ignoring *Var* and *Mu* constructors, we get the traditional formulation of fixpoints of functors used in various simple datatype-generic programming approaches [11, 13]:

$$\textbf{newtype}\ \mathit{Fix}\ f = \mathit{In}'\ \{\mathit{out}' :: f\ (\mathit{Fix}\ f)\}$$

The following are pattern functors for lists and internally labelled trees

$$\begin{aligned}&\textbf{data}\ \mathit{List}_F\ e\ a = []_F \mid e :_F a \\ &\textbf{data}\ \mathit{Tree}_F\ e\ a = \mathit{Leaf} \mid \mathit{Fork}\ e\ a\ a\end{aligned}$$

and values like:

```
alist :: Rec (List_F Int) a
alist = In (1 :_F In (2 :_F In []_F))
atree :: Rec (Tree_F Int) a
atree = In (Fork 3 (In Leaf) (In Leaf))
```

encode a list $[1, 2]$ and a tree *Fork* 3 *Leaf Leaf* respectively.

Binder Support The constructors *Var* and *Mu* provide the PHOAS binder structure. The type variable a of *Rec f a* is the PHOAS type parameter that denotes the type of variables. In the wrapper type *Graph* this type parameter is universally quantified to enable different possible instantiations from the same graph.

The constructor *Var* is conventional for PHOAS; it denotes an occurrence of a variable. The *Mu* constructor is a generalization from a simple binder to one that binds a group of potentially mutually recursive definitions: $[a]$ is a list of names that can be used in all of $[f\ (Rec\ f\ a)]$.

Note that the cycles must be *productive*. A non-productive loop like $Mu\ (\lambda[x] \to [Var\ x])$ is not allowed. Productivity is enforced because every cycle starts with an occurrence f of the pattern functor.

The following represents a cyclic list that starts with $[1, 2]$ and then loops back to the beginning:

```
l1 :: Graph (List_F Int)
l1 = ↓ (Mu (λ[x] → [1 :_F In (2 :_F Var x)]))
```

3 Structured Graph Equality

The *geq* function defines *structural* equality of structured graphs generically.

```
geq :: ∀ f.Eq_F f ⇒ Graph f → Graph f → Bool
geq g1 g2 = runReader (go (↑ g1) (↑ g2)) 0 where
  go :: Rec f Int → Rec f Int → Reader Int Bool
  go (Var x) (Var y) = return (x ≡ y)
  go (Mu g)  (Mu h)  =
    do n ← ask
       let a = g (iterate succ n)
           b = h (iterate succ n)
       and ⟨$⟩ local (+length a) (zipWithM (eq_F go) a b)
  go (In x)  (In y)  = eq_F go x y
  go _       _       = return False
```

The auxiliary function *go* deals with the generic binding structure, while the type class Eq_F provides equality for the structure-specific parts of the rational tree:

```
class Functor f ⇒ Eq_F f where
  eq_F :: Applicative m ⇒ (r → r → m Bool) → f r → f r → m Bool
```

Both the type r of recursive occurrences and the type m of applicative effects [14] are treated as abstract types. The recursive call, which knows how to deal with concrete r and m types is explicitly provided. The eq_F function is used by the function *geq*, which calls eq_F with the auxiliary function *go* as the recursive call argument. This technique avoids leaking implementation details of *geq* (such as dealing with fresh variables) into the code users write at the instances of Eq_F.

An example instance of Eq_F is that for $List_F$:

```
instance Eq e ⇒ Eq_F (List_F e) where
  eq_F (∼) []_F []_F = pure True
  eq_F (∼) (x1 :_F p1) (x2 :_F p2)
    | x1 ≡ x2        = p1 ∼ p2
  eq_F (∼) _ _       = pure False
```

The *go* function numbers all variables with the "nesting depth" of their binder; this depth is maintained in the environment of the *Reader* monad.

3.1 The Problem with Structured Graph Equality

While structured graph equality is fairly simple and straightforward, it is overly restrictive. In particular, it is highly sensitive to incidental encoding differences of structured graphs. For instance, the following cyclic list is a one-step unfolding of l_1.

```
l2 :: Graph (List_F Int)
l2 = ↓(In (1 :_F In (2 :_F Mu (λ[x] → [1 :_F In (2 :_F Var x)]))))
```

Yet, it is structurally different from l_1 according to *geq*. To ignore the encoding differences, we need to consider proper *rational term equality*.

4 Rational Term Equality

Definition 1 (Rational Term Equality). *Two rational terms are equal if the infinite unfoldings of their finite representations are equal:*

```
(∼_RT) :: Eq_F f ⇒ Graph f → Graph f → Bool
g1 ∼_RT g2 = unfold g1 ≡_RT unfold g2
```

where unfolding is defined as:

```
unfold :: ∀f.Functor f ⇒ Graph f → Fix f
unfold g = go (↑ g) where
  go (Var x) = x
  go (Mu g)  = head $ fix (map (In′ ∘ fmap go) ∘ g)
```

```
    go (In fr)  = In' $ fmap go fr
  fix f = let r = f r in r
```

and equality is purely structural:

```
  (≡RT) :: ∀f. Eq_F f ⇒ Fix f → Fix f → Bool
  x1 ≡RT x2 = runIdentity (go x1 x2) where
    go (In' y1) (In' y2) = eq_F go y1 y2
```

The above definition only serves as a specification. It is not practically executable, because $\equiv_{RT}$ does not terminate for infinite structures. What we need is a different approach that properly takes into account the finitely representable nature of rational trees.

5 Pointed Structures

Pointed f augments *Fix f* with node identifiers, called *pointers*. This exposes sharing in the infinite unfolding of the rational term, which is needed to compute equality in finite time.

```
  type Ptr = Int
  data Pointed f = P Ptr (f (Pointed f))
```

Since the graph has a finite number of nodes, the infinite unfolding also contains only a finite number of distinct pointers.

The two cyclic lists above are both represented as follows in this encoding:

```
  let cl0 = P 0 (1 :F cl1)
      cl1 = P 1 (2 :F cl0)
  in cl0 :: Pointed (List_F Int)
```

5.1 From *Graph* to *Pointed*

Every *Graph* can be unfolded into a *Pointed* structure, provided that its pattern functor f is an instance of *Traversable* [14, 15].

```
  toPointed :: ∀ f. Traversable f ⇒ Graph f → Pointed f
  toPointed g = evalState (go$ ↑ g) 0
  where
    go :: Rec f (Pointed f) → State Ptr (Pointed f)
    go (Var x) = return x
    go (Mu g)  = head ⟨$⟩ mfix (traverse go_F ∘ g)
    go (In r)  = go_F r
    go_F :: f (Rec f (Pointed f)) → State Ptr (Pointed f)
```

```
go_F r        = P ⟨$⟩ fresh ⊛ traverse go r
fresh :: State Ptr Ptr
fresh         = do n ← get
                   put (n + 1)
                   return n
```

This conversion function instantiates the PHOAS variable type to the result type *Pointed f*, in correspondence with the unfolding that replaces *Var* constructors by recursive occurrences of terms.

In order to tag every f constructor with a unique pointer, the state monad threads the unique name supply through the conversion. Moreover, the *Traversable f* constraint enables the recursive application of monadic conversion through f constructors.

The most complex case of the conversion is the *Mu* constructor: The (greatest) fixpoint of the binder function g yields the unfolding. This fixpoint is the monadic *mfix*[1] because the unfolding is interleaved with the monadic tagging of constructors. In the process, the multi-binder is turned into a tree, which is assumed to be defined by the first binder.

6 Hopcroft and Karp, Functionally

The *Pointed* structure removes structural differences from *Graph* that are due to the degree of freedom in encoding cycles. However, it does introduce its own incidental variation: the pointers. Moreover, the co-inductive structure makes it hard to determine how much of two graphs must be compared to be certain of their equality. It is with these two challenges that we deal in the last step, using Hopcroft and Karp's algorithm.

The solution to both problems is to build *equivalence classes* of pointers.

```
hk :: ∀ f.Eq_F f ⇒ Pointed f → Pointed f → Bool
hk p_1 p_2 = runUF_M $ p_1 ∼ p_2 where
  (∼) :: Pointed f → Pointed f → UF_M Bool
  P ptr_1 s_1 ∼ P ptr_2 s_2 = (ptr_1 ≡_Ptr ptr_2) ∨_M (s_1 ≡_Stream s_2)
  (≡_Stream) :: f (Pointed f) → f (Pointed f) → UF_M Bool
  s_1 ≡_Stream s_2 = eq_F (∼) s_1 s_2
```

The *hk* function is defined in terms of the auxiliary function $\sim$ that runs in a monad UF_M encapsulating the management of equivalence classes. Two pointed structures are equivalent if either their pointers are equivalent or their structures are.

The $\equiv_{Ptr}$ operator checks whether ptr_1 and ptr_2 are in the same equivalence class. As a side-effect, if they are not, $\equiv_{Ptr}$ merges their equivalence classes.

[1] Note that g must be sufficiently lazy to tie the knot, e.g., use lazy pattern matching as in $\lambda(\sim[x]) \to 1 :_F (Var\ x)$.

Moreover, if they are not equal, the monadic variant of lazy disjunction $\vee_M$, defined as

$$(\vee_M) :: Monad\ m \Rightarrow m\ Bool \to m\ Bool \to m\ Bool$$
$$x \vee_M y = x \gg\!= \lambda b \to \textbf{if}\ b\ \textbf{then}\ return\ b\ \textbf{else}\ y$$

checks for strucural equivalence with $(\equiv_{Stream})$. This operator performs the structural check eq_F for matching constructors and recursively applies the $\sim$ check on subterms.

The equivalence classes solve the two problems:

1. It overcomes the incidental difference in pointer identity by merging different pointers into the same equivalence class.
2. It ensures termination because if no structural difference is found. The algorithm continues as long as two pointers are discovered that do not belong to the same equivalence class. Given that there are only a finite number of pointers, this means that the algorithm runs out of distinct pointers after a finite number of steps.

6.1 Union-Find

A *union-find* structure manages the pointer equivalence classes. We opt for a simple map-based representation:

$$\textbf{type}\ TPtr = Either\ Ptr\ Ptr$$
$$\textbf{type}\ UF = Map\ TPtr\ TPtr$$

Note that we tag the pointers of the two graphs with *Left* and *Right* respectively in order to tell them apart – this prevents confusion when the same identifier is reused in the two graphs. Then the UF_M monad is just an alias for the *State* monad:

$$\textbf{type}\ UF_M\ a = State\ UF\ a$$
$$runUF_M\ m = evalState\ m\ empty$$

The main pointer equality operator $\equiv_{Ptr}$ is defined in terms of the two core union-find operations, *find* and *union*:

$$\begin{aligned}(\equiv_{Ptr}) :: Ptr \to Ptr \to UF_M\ Bool\\ x \equiv_{Ptr} y = \textbf{do}\ rx \leftarrow find\ (Left\ x)\\ \qquad\qquad\quad ry \leftarrow find\ (Right\ y)\\ \qquad\qquad\quad when\ (rx \not\equiv ry)\ (union\ rx\ ry)\\ \qquad\qquad\quad return\ (rx \equiv ry)\end{aligned}$$

where *find* returns the representative or root of an equivalence class:

$$\begin{aligned}find :: TPtr \to UF_M\ TPtr\\ find\ x = \textbf{do}\ uf \leftarrow get\end{aligned}$$

```
                return (find' uf x)
find' :: UF → TPtr → TPtr
find' uf x = go x (lookup x uf) where
  go x Nothing = x
  go x (Just y) = go y (lookup y uf)
```

and *union* merges the equivalence classes:

```
union :: TPtr → TPtr → UF_M ()
union x y = modify (λuf → union' uf x y)
union' :: UF → TPtr → TPtr → UF
union' uf x y = let rx = find' uf x
                    ry = find' uf y
                in insert rx ry uf
```

Note that the runtime complexity can be further improved at both the algorithmic level, by adding path compression and performing union-by-rank [16], and the implementation level, by replacing the *Map* in the *State* monad with an *STArray* in the *ST* monad.

7 Related Work

There is much work related to the equality of rational trees. This section only covers a small, but closely related fragment.

Finite State Automata and Bisimulation There is an obvious connection between the equality of finite state automata (FSAs) and rational terms. It would be interesting to investigate whether other FSA operations, like minimizations [17], can be implemented elegantly for the *Graph* representation.

Equality of FSAs is a special case of *bisimulation* [18], the mutual simulation of two state transition systems. *Weak bisimulation* is an extension of bisimulation where the state transition systems can have *silent* (or *internal*) transitions that are ignored. This notion is not relevant for *Graph* where the *Mu* constructor is explicitly productive, but is useful for, e.g., comparing grammars.

Equi-Recursive Types Gauthier and Pottier [19] study the problem of efficient type-checking in an extension of System F with equi-recursive types, System F_μ. In this setting, equi-recursive types are rational terms, extended with binders denoting universal quantifiers. Gauthier and Pottier provide an encoding that eliminates these binders and reduces the problem of deciding equality to traditional rational term equality. As equi-recursive types are often used in the context of object-oriented programming, another important operation is testing *subtyping* [20].

Rational Term Unification Rational (Herbrand) terms arose naturally in Prolog systems that did not implement the occurs check (by default). Currently, they are considered a desirable feature and actively supported by most major Prolog systems (e.g., SWI-Prolog [21]). In those systems not equality but unification is the central operation. Rational tree unification [22] attempts to make two two rational trees by suitably substituting logical variables. It can be implemented as an extension to Hopcroft and Karp's algorithm.

8 Conclusion

Dealing with graphs and graph algorithms in functional programming languages has always been challenging. Structured graphs show that call-by-need languages like Haskell can conveniently represent certain types of graph structures. However it is still unknown how many classic graph algorithms can be conveniently encoded in this representation.

This paper shows that a classic algorithm for computing the equivalence of finite state automata can be nicely adapted to the setting of structured graphs. Interestingly, the purely functional algorithm mimics a traditional imperative pointer-based algorithm using *pointed structures*. We believe that pointed structures can be useful to similarly adapt other imperative pointer-based algorithms.

References

1. Eggert, P.R., Chow, K.P.: Logic Programming, Graphics and Infinite Terms. Technical report, Department of Computer Science, University of California at Santa Barbara (1983)
2. Colmerauer, A.: Prolog and Infinite Trees. In Clark, K.L., Tärnlund, S.A., eds.: Logic Programming. Academic Press, London (1982) pp. 231–251
3. Giannesini, F., Cohen, J.: Parser Generation and Grammar Manipulation Using Prologs Infinite Trees. Journal of Logic Programming **3**, pp. 253–265 (1984)
4. Pollard, C., Sag, I.A.: Head-Driven Phrase Structure Grammar. University of Chicago Press, Chicago (1984)
5. Carpenter, B.: The Logic of Typed Feature Structures with Applications to Unication-Based Grammars, Logic Programming and Constraint Resolution. In: Cambridge Tracts in Theoretical Computer Science. Volume 32. Cambridge University Press (1992)
6. Erbach, G.: ProFIT: Prolog with Features, Inheritance and Templates. In: 7th Conference of the European Chapter of the Association for Computational Linguistics. pp. 180–187. (1995)
7. Carro, M.: An application of rational trees in a logic programming interpreter for a procedural language. CoRR **cs.DS/0403028**, (2004)
8. Crary, K., Harper, R., Puri, S.: What is a recursive module? In: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation. PLDI '99, New York, NY, USA, pp. 50–63. ACM (1999)
9. Hopcroft, J., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, University of California (1971)

10. Oliveira, B.C.d.S., Cook, W.R.: Functional programming with structured graphs. In: ICFP'12. (2012)
11. Gibbons, J.: Datatype-generic programming. In: Spring School on Datatype-Generic Programming. Volume 4719 of Lecture Notes in Computer Science. Springer-Verlag (2007)
12. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP'08. (2008)
13. Jansson, P., Jeuring, J.: Polyp – a polytypic programming language extension. In: POPL'97. (1997)
14. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming **18**(01), pp. 1–13 (2008)
15. Gibbons, J., Oliveira, B.: The essence of the Iterator pattern. In: Journal of Functional Programming. Volume 19. pp. 377–402. (2009)
16. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2), pp. 245–281 (March 1984)
17. McCluskey, E.J.: Introduction to the Theory of Switching Circuits. McGraw-Hill Book Company, Inc., New York
18. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
19. Gauthier, N., Pottier, F.: Numbering matters: first-order canonical forms for second-order recursive types. In: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming. ICFP '04. (2004)
20. Colazzo, D., Ghelli, G.: Subtyping, Recursion and Parametric Polymorphism in Kernel Fun. Information and Computation **198**(2), pp. 71–179
21. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), pp. 67–96 (2012)
22. Jaffar, J.: Efficient unification over infinite terms. New Generation Computing **2**, pp. 207–219 (1984)