GADTless Programming in Haskell 98

Martin Sulzmann[1] and Meng Wang[2]

[1] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg
[2] Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
meng.wang@comlab.ox.ac.uk

Abstract. Generalized algebraic data types (GADTs) allow to write sophisticated, type-safe programs and transformations. But not many languages, respectively their underlying implementations, support GADTs. We show that Pottier's and Gauthier's polymorphic typed defunctionalization, which was supposed to rely on GADTs, can actually be represented in Haskell 98. Our results help to get a better understanding of GADTs and we identify sufficient conditions under which we can replace GADTs with features available in standardized languages such as Haskell 98.

1 Introduction

Generalized algebraic data types (GADTs) are an extension of (boxed) existential types [12]. In contrast to algebraic data types we may refine the type of a GADT depending on the particular constructor. The power behind GADTs is that we can make use of these type refinements when pattern matching over a GADT.

There is a real "industry" that exploits the power of GADTs in sophisticated ways to write more expressive programs and transformations. We refer to [14, 24, 19, 20] for a selection of examples. This trend is supported by an increasing number of functional and object-oriented language implementations [18, 9] which support GADTs and thus allow more users to make use of GADTs in their programs. However, GADTs still have to be adopted by standard language definitions [13, 17] (respectively, whether GADTs will be part of a new language standard is still under active discussion [8]). Hence, it is important to identify conditions under which powerful features such as GADTs can be replaced by simpler features while not substantially changing the programs and their meaning. We refer to this as *GADTless* programming.

In this paper, we consider GADTless programming in Haskell 98 [17]. Haskell 98 represents a minimal, portable version of the Haskell language including an accompanying standard library. We show that some non-trivial GADT-based transformations and programming examples can actually be expressed in "pure"

Haskell 98. As we will see later, we need one extension, rank-2 types, which is not part of Haskell 98 but supported by common implementations.

Specifically, we make the following contributions:

– We review the polymorphic defunctionalization transformation by Pottier and Gauthier to a target language with support for GADTs. We show that the GADTs arising in the defunctionalization of polymorphic programs can be represented in terms of a form of type coercion expressible via Haskell 98's newtype construct (Section 5.2). Thus, polymorphic typed defunctionalization can be supported by type-preserving compilers which support Haskell 98 style newtype but not necessarily GADTs.
– Other uses of GADTs require an alternative encoding approach where we use explicit coercion functions . For the first time, we identify sufficient conditions under which an encoding of GADTs with explicit coercion functions is possible (Section 6).

In some cases, GADTs can also be encoded by type classes which are part of Haskell 98. To be precise, Haskell 98 supports single-parameter and constructor classes. In our opinion, GADTless programming using type classes is more complicated. We defer a detailed discussion to Section 7.

We continue in Section 2 where we introduce some basic notation used throughout the paper. Section 3 reviews the key concepts behind GADTs. In source programs, we use GADTs as supported by GHC. In Section 4, we discuss the key ideas behind the encodings of GADTs via newtype and explicit coercion functions. Section 7 concludes and also discusses related work. The Appendix contains additional material such as proofs of the results stated.

2 Preliminaries

We write $\bar{o}$ to denote a sequence of objects $o_1, ..., o_n$. We write $fv(o)$ to denote the set of free variables in some object o.

We assume that the reader is familiar with the concepts of substitution, unifiers, most general unifiers (m.g.u.) etc [11]. For example, $[t/a]$ denotes the substitution which has the effect of replacing each occurrence of variable a by term t. Often, we abbreviate $[t_1/a_1, ..., t_n/a_n]$ by $[\bar{t}/\bar{a}]$.

We make use of constraints C consisting of conjunction of primitive constraints such as $t_1 = t_2$ describing equality among types t_1 and t_2. The set of types is described below. We also assume basic familiarity with first-order logic. We write $\models$ to denote the model-theoretic entailment relation. We refer to [25] for details.

Types are stratified into simple types which consist of variables a, functions $t_1 \rightarrow t_2$ and user-definable types $T\ \bar{t}$ (for example lists $[t]$, pairs (t_1, t_2) etc). Type schemes can either be simple types t or of the form $\forall \bar{a}.C \Rightarrow t$. We commonly write σ to refer to type schemes. We write $\forall \bar{a}.t$ as a short-hand for $\forall \bar{a}.b = b \Rightarrow t$.

We assume a functional expression language consisting of variables x, function application $e_1\ e_2$, function abstraction $\lambda x.e$ and let statements $\mathsf{let}\ f =$

e_1 in e_2 which can be optionally annotated let $f :: \forall\bar{a}.C \Rightarrow t; f = e_1$ in e_2 where $\bar{a} = fv(C, t)$. In source programs, we commonly omit the quantifier $\forall\bar{a}$. We also assume case statements case e of $[p_i \rightarrow e_i]_{i\in I}$ where patterns p are of the form x or $K\ \bar{p}$. K is the constructor belonging to a user-definable type $T\ \bar{a}$. We assume that the types of constructors and primitive functions are recorded in some initial type environment Γ_{init}. Type environments Γ are of the form $\{x_1 : \sigma_1, ..., x_n : \sigma_n\}$. We assume that $fv(\Gamma) = fv(\sigma_1) \cup ... \cup fv(\sigma_n)$.

3 Generalized Algebraic Data Types

The power of GADTs comes through (local) type refinement which effectively corresponds to the presence of (local) type equalities $t_1 = t_2$. To illustrate the workings of GADTs, we take look at a classic example. The following defines a GADT `Exp a` to represent a simple expression language and a strongly-typed evaluator `eval` for this language. We will make use of GHC Haskell style syntax [6] in example programs throughout the paper

```
data Exp :: * -> * where
  Zero :: Exp Int
  Succ :: Exp Int -> Exp Int
  Pair :: Exp b -> Exp c -> Exp (b,c)

eval :: Exp a -> a
eval = \ x -> case x of
                Zero -> 0
                (Succ e) -> (eval e) + 1
                (Pair x y) -> (eval x, eval y)
```

At first look it may be surprising that `eval` has type $\forall a.Exp\ a \rightarrow a$. For example, consider the first clause where we return the value `0` of type *Int* which clearly does not match the expected result type a. However, the program is well-typed because of the type refinement enabled by the pattern match over `Zero`.

Here is a more detailed explanation. Variable `x` has type *Exp a*. For the program to be well-typed, we must verify that the pattern clause `Zero->0` has type $Exp\ a \rightarrow a$. When pattern matching over constructor `Zero` of type *Exp Int* we *refine* type a to *Int within the body of the pattern clause*. We can represent type refinement with either substitutions or type equalities. We choose the latter. Hence, we must check that `0` has type a under the (local) type equality $Int = a$. Intuitively, it should be clear that this check is successful.

Let us take a look at the formal underpinnings of GADTs. The GADT type system is based on a variant of the familiar Hindley/Milner type system. Because we choose to use type equalities to represent type refinement we use a variant of the HM(X) type system framework [15] where the constraint domain X is the Herbrand constraint domain. The core GADT system can be summarized in the following four typing rules. All other typing rules are the familiar ones from

HM(X) and are omitted for brevity.

$$(\mathrm{Eq})\ \frac{\begin{array}{c} C, \Gamma \vdash_{GADT}\ e : t_1 \\ C \models t_1 = t_2 \end{array}}{C, \Gamma \vdash_{GADT}\ e : t_2} \quad (\mathrm{Pat})\ \frac{\begin{array}{c} p : t_1 \vdash_{PAT}\ \forall \bar{b}.(D \mid \Gamma_p) \\ \bar{b} \cap fv(C, \Gamma, t_2) = \emptyset \\ C \wedge D, \Gamma \cup \Gamma_p \vdash_{GADT}\ e : t_2 \end{array}}{C, \Gamma \vdash_{GADT}\ p \rightarrow e : t_1 \rightarrow t_2}$$

$$(\mathrm{PVar})\ x : t \vdash_{PAT}\ (True \mid \{x : t\}) \quad (\mathrm{PK})\ \frac{\begin{array}{c} K : \forall \bar{a}, \bar{b}.D \Rightarrow t \rightarrow T\ \bar{a} \\ \bar{b} \cap \bar{a} = \emptyset \quad p : [\bar{t}/\bar{a}]t \vdash_{PAT}\ \forall \bar{b'}.(D' \mid \Gamma_p) \end{array}}{K\ p : T\ \bar{t} \vdash_{PAT}\ \forall \bar{b'}, \bar{b}.(D' \wedge [\bar{t}/\bar{a}]D \mid \Gamma_p)}$$

We assume that constraints C and D contain type equalities. Via rule (Eq) we can change the type of an expression to t_2 if the given constraint C implies that t_1 and t_2 are equal, written $C \models t_1 = t_2$. Applied to our above example, we can deduce that $Int = a, \{0 : Int\} \vdash_{GADT}\ 0 : a$.

In rule (Pat), we type check pattern clauses. Out of the pattern p we extract the constraints arising and compute the pattern binding which assigns types to pattern variables. For simplicity, we assume that constructors K take a single argument only. See rule (PK). We ignore the universal quantifier $\forall \bar{b}$ for the moment. In our representation, we assume that the types of constructors are normalized such that their result type matches the type of the GADT. Thus, we find that `Zero` $: \forall a.Int = a \Rightarrow Exp\ a$. It is straightforward to convert source GADT definitions such as `data Exp :: * -> * where Zero::Exp Int` to the external representation. The main point to note is that the constraint D arising out of p is used in combination with the given constraint C to type check the body e of the pattern clause. Also note that constraint D is only used for type checking e and does not affect other program parts.

The universal quantifier $\forall \bar{b}$ comes into play in case of (boxed) existential types. For example, consider

```
data Exp :: * -> * where Pair::Exp a -> Exp b -> Exp (a,b)
```

which is written `Pair` $: \forall a, b, c.(b, c) = a \Rightarrow Exp\ b \rightarrow Exp\ c \rightarrow Exp\ a$ in the internal system. Notice that variables b and c do not appear in the result type $Exp\ a$. We refer to them as "existential" or "abstract" variables. In programs we are not allowed to make any specific assumptions about them. Hence, logically these variables must be considered as universally quantified. We take care of this in the type system by checking that $\bar{b} \cap fv(C, \Gamma, t_2) = \emptyset$ in rule (Pat).

The challenge for the typed intermediate languages of todays modern compiler systems is to have a sufficiently rich target language which can host GADTs. A naive translation of GADTs to "plain" System F will result in a loss of type-preservation. The reason is that via rule (Eq) (type refinement) we can change the type of an expression without changing the expression itself. There exist sophisticated extensions of System F, for example consider [5, 26, 23], which support a "cast" operator to translate type refinement. For example, in [26] the expression e is translated to $\gamma \blacktriangleright e$ where $\cdot \blacktriangleright \cdot$ represents a new language construct which takes a witness γ for $t_1 = t_2$ and an expression e of type t_1 and yields an expression of type t_2. But not every language and its underlying implementation support such cast operations.

4 GADT Encodings via Newtypes and Explicit Coercions

To encode GADTs in a conventional system such as Haskell 98 we need to model type equalities $t_1 = t_2$ and their effect. The idea is to replace each type refinement step via safe coercion functions. Each type refinement step $e : t_1 \rightsquigarrow e : t_2$ is then turned into the function application $\gamma\ e$ where γ is the coercion representing the "directed" equality $t_1 = t_2$ (from left to right). In essence, the encodings apply the proofs-are-programs principle where the coercion γ represents a proof (term) for $t_1 = t_2$. Of course, to ensure correctness of this encoding scheme, we need to guarantee that at run-time each coercion γ evaluates to the identity by construction.

There are two approaches known in the literature to encode such coercion functions. One approach, employed in [1, 3, 16, 28], uses "Leibniz" equality

```
newtype a :=: b = Proof { coerce :: forall f . f a -> f b }
refl :: a :=: a
refl    =  Proof id
newtype Flip f a b =  Flip { unFlip :: f b a }
symm :: a :=: b -> b :=: a
symm p =  unFlip (coerce p (Flip refl))
trans :: a :=: b -> b :=: c -> a :=: c
trans p q =  Proof (coerce q . coerce p)

newtype Id a  =  Id { unId :: a }
to :: a :=: b -> (a -> b)
to p =  \ a -> unId (coerce p (Id a))
newtype Inv a b               =  Inv { unInv :: b -> a }
from :: a :=: b -> (b -> a)
from p = to (symm p)
```

In the above, we employ newtype declarations as supported by Haskell 98 extended with rank-2 type (notice the `forall f`). The purpose of the rank-2 type is to ensure that the identity function is the only inhabitant of `a :=: b`. We ignore here the two "exceptional" cases $\perp$ and `\ x ->` $\perp$.

Types introduced via the `newtype` declaration are new, distinct types isomorphic to existing ones. We will refer to such types as *newtypes* from now on. They are similar to (labeled) data type declaration such as

```
data a :=: b = Proof { coerce :: forall f . f a -> f b }
```

The difference is that newtypes cause no run-time type overhead. Constructor `Proof` coerces a value from type `a :=: b` to type `forall f . f a -> f b`. The important point is that these coercions can be implemented without execution time overhead; newtype does not change the underlying representation of an object [17].

The up-shot of the above is that the new type `a:=:b` encodes Leibniz' law which states that if `a` and `b` are equivalent then we may substitute one for the other in any context. In the above, we provide a few sample type coercion functions representing the expected equality laws: reflexivity, symmetry and transitivity. Functions `to` and `from` represent directed coercions. Thus, we can encode the GADT program from Section 3 in Haskell 98 as follows.

```
data Exp' a where
  Zero' :: a :=: Int -> Exp' a
  Succ' :: a :=: Int -> Exp' a -> Exp' a
  Pair' :: a :=: (b,c) -> Exp' b -> Exp' c -> Exp' a


eval' :: Exp' a -> a
eval' = \ x -> case x of
                 (Zero' p) = (from p) 0
                 (Succ' p e) = (from p) (((to p) (eval' e)) + 1)
                 (Pair' p x y) = (from p) ((eval' x, eval' y))
```

Each GADT equation $t_1 = t_2$ appearing in a constructor is turned into an additional argument $\mathtt{t}_1\mathtt{:=:t}_2$. For example, $\mathtt{Zero} : \forall a.Int = a \Rightarrow Exp\ a$ is turned into $\mathtt{Zero}' : \forall a.Int :=: a \rightarrow Exp'\ a$. We find that `Exp' a` is a Haskell 98 data type (though we use here GHC syntax for writing data type declarations). In the body of each pattern clause, we replace each type refinement step (Eq) via an application of the directed coercion functions `to` or `from`.

The alternative to the newtypes method is to represent type equality via some explicit coercion functions [2].

```
type EQ a b = (a->b,b->a)
refl :: EQ a a
refl = (id,id)
sym :: EQ a b -> EQ b a
sym (f,g) = (g,f)
trans :: EQ a b -> EQ b c -> EQ a c
trans (f1,g1) (f2,g2) = (f2.f1,g1.g2)
list :: EQ a b -> EQ [a] [b]
list (f,g) = (map f, map g)
```

The gist of the GADT encoding method remains the same. In contrast to the newtypes approach, the explicit coercion approach incurs a run-time penalty. For example, consider the compositional law for lists where we actually have to traverse the entire list to coerce each element into its proper type. We also need to guarantee that each coercion evaluates to the identity at run-time. This is enforced by the type system in case of the newtypes approach.

As we will shortly show in the upcoming Section 5.2, the GADTs which arise in Pottier's and Gauthier's polymorphic typed defunctionalization transformation can be encoded via newtypes. For many other GADT programs we seem to require explicit coercions in the encoding. The details are in Section 6.

5 Defunctionalization

Defunctionalization [21, 22] is a global program transformation to turn higher-order programs into first-order order ones. The basic idea is to replace every lambda abstraction with its own data constructor that will carry what environment is needed, and replace every application of the higher-order function with an apply function that will interpret the data structure.

For example, defunctionalization of the higher-order program

```
empty = \ x1 -> False
insert = \ x2 -> \ x3 -> \ x4 -> (x2==x4) ||(x3 x4)
res = insert 1 empty 2
```

yields the following first order program

```
apply = \ f -> \ arg ->
   case f of
     X1 -> False
     X2 -> let x2 = arg in X3 x2
     (X3 x2) -> let x3 = arg in X4 x2 x3
     (X4 x2 x3) -> let x4=arg in (x2==x4) || (apply x3 x4)

empty = X1
insert = X2
res = apply (apply (apply (insert 1)) empty) 2
```

We assume that primitives `==` and `||` are not affected by defunctionalization. Function closure construction is encoded as injection (construction of a sum of type *Arrow a b*) whereas function application is encoded as case analysis (sum elimination). We represent *Arrow a b* via a data type where for each lambda abstraction we introduce the following constructors. Free variables are passed in as arguments to constructors.

$\texttt{X}_1 : \forall a. Arrow\ a\ Bool$
$\texttt{X}_2 : \forall a. Arrow\ a\ (Arrow\ (Arrow\ a\ Bool)\ (Arrow\ a\ Bool))$
$\texttt{X}_3 : \forall a. a \rightarrow Arrow\ (Arrow\ a\ Bool)\ (Arrow\ a\ Bool)$
$\texttt{X}_4 : \forall a. a \rightarrow Arrow\ a\ Bool \rightarrow Arrow\ a\ Bool$

We have achieved a defunctionalization of programs, but the problem is that the transformation is not type-preserving (assuming we use Hindley/Milner or System F as our target language). The first problem is that the above constructors do not share the same (result) type. Hence, we cannot represent them via an algebraic data type. The second problem is that function `apply` is not typable in Hindley/Milner or System F. For example, the first clause of `apply` has return type *Bool* whereas the second clause returns a value of different type *Arrow* (*Arrow a Bool*) (*Arrow a Bool*).

A possible work-around is to specialize function `apply` with each monomorphic type; in the case of (parametric) polymorphism, monomorphization is needed prior to defunctionalization. However, this leads to code duplication and is even impossible in case the source language supports polymorphic recursion (which is the case for Haskell 98).

Fortunately, Pottier and Gauthier [19] came up with an ingenious idea to solve this problem. They show how to define a type-preserving defunctionalization of polymorphic programs if the target language is enriched with GADTs. We rewiew the gist of their method in Section 5.1. A closer inspection of their method reveals that the GADTs arising in the defunctionalization of polymorphic programs can be replaced via newtypes. We formalize this insight in Section 5.2.

5.1 Polymorphic Typed Defunctionalization via GADTs

We apply Pottier's and Gauthier's type-preserving defunctionalization transformation to our running example. Their key idea is to use a GADT (instead of an algebraic data type) to represent the sum type *Arrow*.

```
data Arrow :: * -> * -> * where
   X1 :: Arrow a1 Bool
   X2 :: Arrow a1 (Arrow (Arrow a1 Bool) (Arrow a1 Bool))
   X3 :: a -> Arrow (Arrow a Bool) (Arrow a Bool)
   X4 :: a1 -> Arrow a1 Bool -> Arrow a1 Bool
```

In the GADT type system, function `apply` is well-typed.

```
apply :: Arrow a1 a2 -> a1 -> a2
apply = \ f -> \ arg ->
   case f of
     X1 -> False
     X2 -> let x = arg in X3 x2
     (X3 x2) -> let x3 = arg in X4 x2 x3
     (X4 x2 x3) -> let x4=arg in (x2==x4) || (apply x3 x4)
```

For example, consider the last case. Pattern matching over `(X4 x2 x3)` yields a binding $\{x2 : a1; x3 : \mathit{Arrow}\ a1\ \mathit{Bool}\}$ and (local) constraint $\mathit{Bool} = a2$. The let statement yields the binding $\{x4 : a1\}$. We straightforwardly find that `(x2==x4) || (apply x3 x4)` has type *Bool*. From $\mathit{Bool} = a2$ and via application of rule (Eq) we can conclude that this program text has also type $a2$. Hence, the pattern clause agrees with the type annotation. Similarly, we can verify type correctness of the first three cases.

The upshot of Pottier's and Gauthier's method is that defunctionalization can be added to the toolbox of type-preserving compiler writers *if* the target language supports GADTs. Our insight is that GADTs are not necessary in Pottier's and Gauthier's defunctionalization method. We can always replace the GADTs arising in the defunctionalization via newtypes while retaining type preservation. That is,

First, we replace the `Arrow` GADT by

```
data Arrow':: * -> * -> * where
  X1' :: a2 :=: Bool -> Arrow' a1 a2
  X2' :: a2 :=: (Arrow' (Arrow' a1 Bool) (Arrow' a1 Bool)) -> Arrow' a1 a2
  X3' :: a1 :=: Arrow' a Bool -> a2 :=: Arrow' a Bool -> a -> Arrow' a1 a2
  X4' :: a2 :=: Bool -> a1 -> Arrow' a1 Bool -> Arrow' a1 a2
```

Each GADT constructor such as `X1:∀a1,a2.a2=Bool ⇒ Arrow a1 a2` is turned into `X1':∀a1,a2.a2:=:Bool → Arrow' a1 a2`. We find that `Arrow'` is a Haskell 98 data type (though we use here GHC syntax for writing data type declarations).

The challenge is to replace `apply` by an equivalent definition which is typable in Haskell 98. The key observation is that uses of the GADT rule (Eq) for typing `apply` can be omitted by inserting newtypes coercions `to` or `from`.

```
apply' :: Arrow' a1 a2 -> a1 -> a2
apply' = \ f -> \ arg ->
   case f of
     (X1' ep2) -> (to ep2) False
     (X2' ep2) -> let x2 = arg in (to ep2) (X3' x2 refl refl)
     (X3' ep1 ep2 x2) -> let x3 = arg in (to ep2) (X4' x2 ((from ep1) x3) refl)
     (X4' ep2 x2 x3) -> let x4=arg in (to ep2) ((x2==x4) || (apply' x3 x4))

empty = X1' refl
insert = X2' refl
res = apply' (apply' (apply' insert 1)) empty) 2
```

For example, in the last clause application of the coercion `(to ep1)` has the same effect as application of the typing rule (Eq) under constraint $Bool = a2$. Thus, we achieve a type-preserving defunctionalization via newtypes for our running example. The above observation can be generalized for all GADTs which arise in Pottier's and Gauthier's defunctionalization method. The details are in the next section.

5.2 Polymorphic Typed Defunctionalization via Newtypes

Our scheme follows the approach in [19]. Defunctionalization is defined in terms of judgments $\Gamma \vdash^{DF} e : t \rightsquigarrow e'$ where we assume that source expression e has type t under environment Γ, and the target expression e' is the defunctionalized version of e. We write $[\![t]\!]$ to denote the defunctionalized version of type t (which naturally extends to type environment $[\![\Gamma]\!]$):

$$[\![a]\!] = a \qquad [\![t_1 \to t_2]\!] = Arrow' \; [\![t_1]\!] \; [\![t_2]\!] \qquad [\![T \; \bar{t}]\!] = T \; [\![\bar{t}]\!]$$

The crucial difference to [19] is that we only rely on Haskell 98 features in the target language. For each lambda abstraction $\lambda x.e : t_1 \to t_2$, where $\bar{y} = fv(\lambda x.e)$ and each y_i has type t_i in the given type environment, we introduce a Haskell 98 data constructor `X'` $: \forall a_1, a_2, fv(t_1, t_2, \bar{t}).a_1 :=: [\![t_1]\!] \to a_2 :=: [\![t_2]\!] \to [\![\bar{t}]\!] \to$ `Arrow'` $a_1 \; a_2$ to the initial type environment. We assume that a_1 and a_2 are fresh variables. For example, for `\ x`$_1$ `-> False` we introduce `X1'` $: \forall a1, a2, b.a1 :=:$

$$\text{(Var)}\ \frac{(x : \forall \bar{a}.t) \in \Gamma}{\Gamma \vdash^{DF} x : [\bar{t}/\bar{a}]t \leadsto x} \quad \text{(App)}\ \frac{\Gamma \vdash^{DF} e_1 : t_2 \rightarrow t \leadsto e_1' \quad \Gamma \vdash^{DF} e_2 : t_2 \leadsto e_2'}{\Gamma \vdash^{DF} e_1\ e_2 : t \leadsto \mathit{apply}'\ e_1'\ e_2'}$$

$$\text{(Abs)}\ \frac{\begin{array}{c}\bar{y} = \mathit{fv}(\lambda x.e) \quad \bar{y} = y_1, ..., y_n \quad \Gamma \vdash^{HM} y_i : t_i \quad \text{for } i = 1, .., n \\ \Gamma \vdash^{HM} X' : (a_1 :=: [\![t_1]\!]) \rightarrow (a_2 :=: [\![t_2]\!]) \rightarrow [\![\bar{t}]\!] \rightarrow \mathit{Arrow}\ a_1\ a_2 \\ \Gamma \cup \{x : a_1, ep_1 : a_1 :=: t_1, ep_2 : a_2 :=: t_2\} \vdash^{DF} e : t_2 \leadsto e'\end{array}}{\Gamma \vdash^{DF} \lambda x.e : t_1 \rightarrow t_2 \leadsto X'\ \mathit{refl}\ \mathit{refl}\ \bar{y}}$$

$$\text{(Eq}_1\text{)}\ \frac{\begin{array}{c}\Gamma \vdash^{DF} e : t_1 \leadsto e' \\ p : (t_1 :=: t_2) \in \Gamma\end{array}}{\Gamma \vdash^{DF} e : t_2 \leadsto (\mathit{from}\ p)\ e'} \quad \text{(Eq}_2\text{)}\ \frac{\begin{array}{c}\Gamma \vdash^{DF} e : t_2 \leadsto e' \\ p : (t_1 :=: t_2) \in \Gamma\end{array}}{\Gamma \vdash^{DF} e : t_1 \leadsto (\mathit{to}\ p)\ e'}$$

Fig. 1. GADTless Defunctionalization

$b \rightarrow a2 :=:$ `Bool` $\rightarrow$ `Arrow'` $b\ a2$ which could be simplified to `X1'` $: \forall a1, a2.a2 :=:$ `Bool` $\rightarrow$ `Arrow'` $a1\ a2$ as we have done in the previous section.

Let us take a closer look at the defunctionalization rules in Figure 1. Each lambda-abstraction is injected into the datatype Arrow' with the corresponding data constructor. See rule (Abs). Proof witnesses encapsulating the identity function (refl) and free term variables ($\bar{y}$) are passed in as arguments. We write $\Gamma \vdash^{HM} y_i : t_i$ to denote well-typing in Hindley/Milner extended with newtypes. The defunctionalization of function body e is done under an environment extended with two proof witnesses named as ep_1 and ep_2: one for the input type and one for the output type. Note that we bind the lambda variable x with type a_1 instead of t_1 in the environment under which e is typed. Rule (Eq$_1$) is used to coerce the type of x from a_1 to t_1. As a result, every occurrence of x in e is expected to be substituted with `(from ep`$_1$`)` x in e'. The defunctionalization result e' does not appear in rule (Abs)'s conclusion; instead, it is used in constructing the special function `apply'` during a post-translation step (see below). This reflects the fact that in a defunctionalized program, actual function bodies are all collected by the definition of `apply'`, and each closure residues at the invocation site only encapsulates an unique identifier and a value environment. For each function application, the closure is passed to function `apply'` for proper dispatching. See rule (App).

The purpose of rules (Eq$_1$) and (Eq$_2$) is to mimic the GADT typing rule (Eq) by inserting appropriate directed type coercions. Recall that the difficulty of defunctionalization lies in defining a well-typed function `apply`. In the GADT-based defunctionalization setting, given `apply`'s type $\mathit{Arrow}\ a_1\ a_2 \rightarrow a_1 \rightarrow a_2$, each defunctionalized function `\ x ->` e' of type $[\![t_1]\!] \rightarrow [\![t_2]\!]$ must be made to match `apply`'s signature. In other words, a function of type $[\![t_1]\!] \rightarrow [\![t_2]\!]$ must accept an input of type a_1 and returns an output of type a_2. Consider the third case of our running example, the function `\ x3 -> X4 x2 x3` is of type

Arrow a Bool → *Arrow a Bool* which does not match function `apply`'s signature $a_1 \to a_2$. This is the very and only place (i.e. the definition of `apply`) where we deploy rule (Eq) to achieve type preservation in the GADT-based defunctionalization scheme. By construction, pattern `X` for function `\ x ->` e' always carries the assertion $a_1 = [\![t_1]\!]$ and $a_2 = [\![t_2]\!]$, which allows rule (Eq) to coerce from a_1 to $[\![t_1]\!]$ and from $[\![t_2]\!]$ to a_2 when e' is typed. In the case of `\ x3 -> X4 x2 x3`, pattern matching `X3` yields the constraints $a_1 = \mathit{Arrow}\ a\ \mathit{Bool}$ and $a_2 = \mathit{Arrow}\ a\ \mathit{Bool}$, which justifies the type $a_1 \to a_2$. Note that in defunctionalization, uses of local type equality constraints are always "straightforward": coercions are only between types where a equality constraint of the two syntactically appears in the store. Thus, the logic implication ($\models$) used in rule (Eq) can be simplified as set inclusion ($\in$). Applied to our Haskell 98-based type-directed translation, this property guarantees that either rule (Eq$_1$) or (Eq$_2$) can be used in place of (Eq) when needed, as proof witnesses of correct types should be directly available from the environment.

The last step of defunctionalization is a post-translation process which completes the definition of function `apply'`. For each lambda abstraction in the source program whose defunctionalization derivation ends with

$$(\mathrm{Abs})\ \frac{\begin{array}{c}\bar{y} = \mathit{fv}(\lambda x.e) \quad \bar{y} = y_1, ..., y_n \quad \Gamma \vdash^{HM} y_i : t_i \quad \text{for } i = 1, .., n \\ \Gamma \vdash^{DF} X' : (a_1 :=: [\![t_1]\!]) \to (a_2 :=: [\![t_2]\!]) \to [\![\bar{t}]\!] \to \mathit{Arrow}\ a_1\ a_2 \\ C, \Gamma \cup \{x : a_1, ep_1 : a_1 :=: t_1, ep_2 : a_2 :=: t_2\} \vdash^{DF} e : t_2 \leadsto e'\end{array}}{C, \Gamma \vdash^{DF} \lambda x.e : t_1 \to t_2 \leadsto X'\ \mathit{refl}\ \mathit{refl}\ \bar{y}}$$

we generate a pattern clause patcl_x of the form

```
X' ep₁ ep₂ ȳ -> let x = arg in (to ep₂) e'
```

We collect all such clauses to build

```
apply :: Arrow a₁ a₂ -> a₁ -> a₂
apply = \ f -> \ arg ->
   case f of $\overline{\mathit{patcl}_x}$
```

It follows straightforwardly that defunctionalized programs produced by our scheme are operationally equivalent to the target program generated in [19]. The only difference is the additional occurrence of coercions `to refl` or `from refl` which evaluate to the identity. Recall that coercions among new, isomorphic types can be implemented without execution time overhead because newtype does not change the underlying representation of an object [17].

The important result is that defunctionalized programs are well-typed Haskell 98 programs. Recall that $\vdash^{HM}$ refers to the Hindley/Milner system extended with newtypes.

Theorem 1 (Well-Typed). *Let* $\Gamma \vdash^{HM} e : t$ *and* e *is defunctionalized to* e'*. Then* $[\![\Gamma]\!] \vdash^{HM} e' : [\![t]\!]$*.*

We can also state completeness. That is, all well-typed programs can be defunctionalized.

Theorem 2 (GADTless Defunctionalization Completeness). *Let $\Gamma \vdash^{HM} e : t$. Then $\Gamma \vdash^{DF} e : t \leadsto e'$ for some e'.*

Proofs of the above results are given in Appendix B.

6 Further GADTless Programming Examples

The GADT examples we have seen so far could straightforwardly be encoded via newtypes. In this section, we observe that for many other GADT examples the explicit coercion encoding approach appears to be more suitable.

Let's attempt an encoding of the trie example found in [4]. A trie is a finite map from keys to values whose structure depends on the type of keys, here encoded as products and sums in GADT variants:

```
data Either a b where
   Left  :: a -> Either a b
   Right :: b -> Either a b
data Trie k v where
  TUnit ::
    Maybe v                -> Trie () v
  TSum  :: forall k1 k2.
    Trie k1 v -> Trie k2 v -> Trie (Either k1 k2) v
  TProd :: forall k1 k2.
    Trie k1 (Trie k2 v)    -> Trie (k1, k2) v
```

A trie for a unit type is maybe one value, a trie for a sum is a product of tries, and a trie for a product is a composition of tries. An important operation on tries is the merging of two maps with the same domain and co-domain.

```
merge :: (v -> v -> v)
      -> Trie k v -> Trie k v -> Trie k v
merge c (TUnit Nothing ) (TUnit Nothing  ) =
  TUnit Nothing
merge c (TUnit Nothing ) (TUnit (Just v')) =
  TUnit (Just v')
merge c (TUnit (Just v)) (TUnit Nothing  ) =
  TUnit (Just v)
merge c (TUnit (Just v)) (TUnit (Just v')) =
  TUnit (Just (c v v'))
merge c (TSum ta tb)     (TSum ta' tb')    =
  TSum (merge c ta ta') (merge c tb tb')
merge c (TProd ta)       (TProd ta')       =
  TProd (merge (merge c) ta ta')
```

The second two last function clauses are interesting. The patterns of the first and second argument constrain `k` to `Either k1 k2` and `Either k1' k2'`, respectively. Hence, we have

$$\texttt{Either k1 k2} = \texttt{k} = \texttt{Either k1}' \texttt{ k2}'$$

from which we can follow $\texttt{k1} = \texttt{k1}'$ and $\texttt{k2} = \texttt{k2}'$.

The point is that in the GADT type system we can deduce $t_1 = t'_1$,...,$t_n = t'_n$ from $T\ t_1\ ...t_n = T\ t'_1...t'_n$ for any n-ary type constructor T. This is the standard "decomposition" law for Herbrand type constructors. In terms of the encoding schemes described in Section 4, this means that we need to decompose the proof term associated to $T\ t_1\ ...t_n = T\ t'_1...t'_n$ into a proof term associated to $t_i = t'_i$. But it seems impossible to define such a decomposition law in Haskell 98 using newtypes.

The explicit coercion encoding approach seems more flexible when it comes to decomposition. For simplicity, we only give parts of the definition of the decomposition law for the `Either a b` data type.

```
decomp1 :: (Either a b -> Either c d) -> (a->c)
decomp1 f  = \ a -> case (f (Left a)) of
                          Left c -> c
```

We inject the `a` value into the `Either a b` data type, apply the incoming coercing function and then extract the `c` value. Notice that if (input) function `f` behaves like the identity function, the resulting (output) function behaves like the identity function as well. We can therefore argue that decomposition for `Either a b` is (correctly) definable in Haskell 98. Thus, we can rewrite the "trie" example in an equivalent form which is accepted by Haskell 98. Due to space limitations we provide the details in Appendix A.

There are many other examples which can be translated using the explicit coercion approach. A comprehensive list of examples can be found under [3]

`http://users.ox.ac.uk/~wolf2335/projects/translate-gadt/`

In fact, it almost seems that all practical examples can be encoded. Though, not every decomposition function is definable. Here is the (contrived) critical example.

```
data Foo a  where
   K :: Foo a
data Erk a b c where
   I :: c -> Erk a a c
f :: Erk (Foo a) (Foo Int) a -> a
f (I x) = x + 1
```

First, we convince ourselves that the above program is well-typed in the GADT system. The pattern `I x` in combination with the type annotation implies that `Foo a` = `Foo Int`. By decomposition, we conclude that `a` = `Int`. Thus, the program text `x+1` can be given type `Int`. Hence, the above is well-typed. To translate the above, we need to define a function of type `EQ (Foo a) (Foo Int) -> EQ a Int`. We claim it is impossible to define such a function with satisfies the invariant. It suffices to show that a function

```
decompFoo :: (Foo a->Foo Int)->(a->Int)
```

[3] Examples are also part of the technical report version [27].

with the property that `decompFoo (\ x->x)` evaluates to `\ x->x` is not definable.

The problem here is that a value of type `a` cannot be injected into a value of type `Foo a`. So, clearly the incoming function of type `Foo a->Foo Int` is useless. Effectively, we could omit the function parameter altogether. Parametricity tells us that any function of type `a->Int` must be a constant function. Hence, `decompFoo` applied to any function of type `Foo a->Foo Int` yields a constant function. Hence, an encoding of the above critical example is impossible.

The above suggests that in order to represent the decomposition law via the explicit coercion approach, (1) we must demand that for each data type $T\ a_1...a_n$, all type parameters a_1,...,a_n appear in the argument type of at least one constructor. This immediately rules out examples such as the above phantom type *Foo a* and also abstract types such as *IO a*. [4] In addition, (2) we require that type parameters a_1, ..., a_n appear in a positive position in the argument of a constructor. We say that a data type $T\ a_1...a_n$ is *decomposable* iff the above conditions (1) and (2) are satisfied.

Theorem 3 (GADTless Programming via Explicit Coercions). *Every GADT program which only uses decomposable types can be encoded via explicit coercions.*

A proof sketch of the above result is given in Appendix C.

7 Conclusion and Related Work

The main result of this paper is that Pottier's and Gauthier's [19] GADT-based polymorphic typed defunctionalization can actually be expressed in Haskell 98 using newtypes to encode GADTs (Section 5.2). Thus, a type-preserving polymorphic defunctionalization becomes possible for many systems without having to extend the typed-intermediate language with GADTs (or something equivalent).

The idea of using newtypes to encode GADTs originates from the work by Baars and Swierstra [1], Hinze and Cheney [3], and possibly many others. But many GADT programs can only be encoded using explicit coercion functions. The idea of using explicit coercion functions to mimic GADT style program appears first in the work by Yang [29]. His method has been re-invented later by Chen, Zhu and Xi [2]. In our second main result, we establish for the first time sufficient conditions under which an encoding of GADTs with explicit coercions is possible in Haskell 98 (Section 6). The disadvantage of the explicit coercion compared to the newtypes approach is that explicit coercions impose a run-time penalty. Furthermore, there are (albeit contrived) GADT programs which cannot seem to be encoded via explicit coercions. This happens if one cannot "decompose" type equations (respectively their associated proof terms/coercions).

Interestingly, the "decomposition" problem also arises when translating type class programs [7].

[4] A phantom type has a type parameter which does not appear as an argument of a constructor. The constructors of an abstract type are not visible to the user.

```
class Foo a where foo :: a->Int
instance Foo a => Foo [a] where
    foo [] = 1
    foo _  = 2
bar :: Foo [a] => a->Int
bar = foo
```

Based on the System F-style translation scheme described in [7], we are unable to translate function `bar`. The program text demands a dictionary for `Foo a` but the annotation only supplies a dictionary for `Foo [a]`. This is the wrong way around. The instance declaration tells us how to construct `Foo [a]` given `Foo a` but the other direction does not hold in general.

Many other programming language features appear to be (roughly) equivalent in terms of expressive power to GADTs. For example, type classes, as pioneered by Weirich [28], can also be used to encode GADT style behavior. Kiselyov [10] provides numerous examples of ingenious type class encodings of GADT programs. The gist of his idea is to turn each (value) pattern clause into an (type class) instance declaration. A drawback of the type class encoding is that the original GADT program has to go under some substantial rewrites. Furthermore, GADTs are closed whereas type class instances are open. Hence, both concepts seem to complement, rather than substitute, each other.

In conclusion, we could show that some non-trivial GADT-based transformations and programs can actually be expressed in Haskell 98. Our results help to get a better understanding of GADTs and under which conditions we can replace them with features available in Haskell 98.

Acknowledgments

We thank Gregory Duck, Manuel M. T. Chakravarty, James Cheney, Oleg Kiselyov, Simon Peyton Jones and Jeremy Wazny for their comments on previous versions of this paper.

References

1. A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *Proc. of ICF'02*, pages 157–166. ACM Press, 2002.
2. C. Chen, D. Zhu, and H. Xi. Implementing cut elimination: A case study of simulating dependent types in Haskell. In *Proc. of PADL'04*, volume 3057 of *LNCS*, pages 239–254. Springer-Verlag, 2004.
3. J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. of Haskell Workshop'02*, pages 90–104. ACM Press, 2002.
4. J. Cheney and R. Hinze. First-class phantom types. TR 1901, Cornell University, 2003.
5. K. Crary and S. Weirich. Flexible type analysis. In *Proc. of ICFP'99*, pages 233–248. ACM Press, 1999.
6. Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

7. C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.
8. http://hackage.haskell.org/trac/haskell-prime.
9. A. Kennedy and C. V. Russo. Generalized algebraic data types and object-oriented programming. In *Proc. of OOPSLA'05*, pages 21–40. ACM Press, 2005.
10. O. Kiselyov. Typed lambda-expressions without gadts. http://www.haskell.org//pipermail/haskell-cafe/2005-January/008212.html, 2005. Haskell-Cafe Mailing List.
11. J. Lassez, M. Maher, and K. Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.
12. K. Läufer and M. Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 78–91, 1992.
13. Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997.
14. H. Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proc. of ICFP'05*, pages 54–65. ACM Press, 2005.
15. M. Odersky, M. Sulzmann, and M Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
16. E. Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health & Science University, OGI School of Science & Engineering, September 2004.
17. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
18. S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. of ICFP'06*, pages 50–61. ACM Press, 2006.
19. F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *Proc. of POPL'04*, pages 89–98. ACM Press, January 2004.
20. F. Pottier and Y. Régis-Gianas. Towards efficient, typed LR parsers. In *ACM Workshop on ML*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 155–180, 2006.
21. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM Press.
22. John C. Reynolds. Definitional interpreters revisited. *Higher Order Symbol. Comput.*, 11(4):355–361, 1998.
23. Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proc. of POPL'02*, pages 217–232. ACM Press, 2002.
24. T. Sheard. Languages of the future. *SIGPLAN Not.*, 39(10):116–119, 2004.
25. J.R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.
26. M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Proc. of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 53–66. ACM Press, 2007.
27. M. Sulzmann and M. Wang. A systematic translation of guarded recursive data types to existential types. Technical Report TR22/04, The National University of Singapore, 2004.
28. S. Weirich. Type-safe cast: (functional pearl). In *Proc. of ICFP'00*, pages 58–67. ACM Press, 2000.
29. Z. Yang. Encoding types in ML-like languages. In *Proc. of ICFP '98*, pages 289–300. ACM Press, 1998.

A GADT Encoding Examples via Explicit Coercions

```
type EQ a b = (a->b,b->a)

sym :: EQ a b -> EQ b a
sym (f,g) = (g,f)

data Trie k v where
  TUnit ::
    EQ k () -> Maybe v -> Trie k v
  TSum  :: forall k1 k2.
    EQ k (Either k1 k2) -> Trie k1 v -> Trie k2 v -> Trie k v
  TProd :: forall k1 k2.
    EQ k (k1,k2) -> Trie k1 (Trie k2 v)    -> Trie k v

merge :: (v -> v -> v) -> Trie k v -> Trie k v -> Trie k v
merge c (TUnit (g1,h1) Nothing ) (TUnit (g2,h2) Nothing  ) =
  TUnit (g1,h1) Nothing
merge c (TUnit (g1,h1) Nothing ) (TUnit (g2,h2) (Just v')) =
  TUnit (g1,h1) (Just v')
merge c (TUnit (g1,h1) (Just v)) (TUnit (g2,h2) Nothing  ) =
  TUnit (g1,h1) (Just v)
merge c (TUnit (g1,h1) (Just v)) (TUnit (g2,h2) (Just v')) =
  TUnit (g1,h1) (Just (c v v'))
merge c (TSum (g1,h1) ta tb)     (TSum (g2,h2) ta' tb')    =
  TSum (g1,h1) (merge c ta ((comp ((decompE1 (g1.h2)),(decompE1 (g2.h1)))) ta'))
               (merge c tb ((comp ((decompE2 (g1.h2)),(decompE2 (g2.h1)))) tb'))
merge c (TProd (g1,h1) ta)       (TProd (g2,h2) ta')       =
  let k1'k1 = ((decompP1 (g1.h2)),(decompP1 (g2.h1)))
      k2'k2 = ((decompP2 (g1.h2)),(decompP2 (g2.h1)))
  in  TProd (g1,h1) (merge (merge c) ta (comp2 k1'k1 k2'k2 ta'))

comp :: EQ a b -> Trie a v -> Trie b v
comp (f',g') (TUnit (f,g) v)    = TUnit (f.g', f'.g) v
comp (f',g') (TSum (f,g) t1 t2) = TSum (f.g', f'.g) t1 t2
comp (f',g') (TProd (f,g) t1)   = TProd (f.g', f'.g) t1

comp' :: EQ v v' -> Trie k v -> Trie k v'
comp' (f',g') (TUnit eq v) = TUnit eq (compM f' v)
comp' eq' (TSum eq t1 t2)  = TSum eq (comp' eq' t1) (comp' eq' t2)
comp' eq' (TProd eq t1)    = TProd eq (comp' ((comp' eq'), (comp' (sym eq'))) t1)

comp2 :: EQ a b -> EQ a' b' -> Trie a (Trie a' v) -> Trie b (Trie b' v)
comp2 eq1 eq t = comp' (comp eq, comp (sym eq)) (comp eq1 t)

compM :: (a->b) -> Maybe a -> Maybe b
compM f Nothing  = Nothing
compM f (Just x) = Just (f x)

decompP1 :: ((a,b) -> (c,d)) -> (a->c)
```

```
decompP1 f  = \ a -> case (f (a,undefined)) of
                       (c,_) -> c

decompP2 :: ((a,b) -> (c,d)) -> (b->d)
decompP2 f  = \ b -> case (f (undefined,b)) of
                       (_,d) -> d

decompE1 :: (Either a b -> Either c d) -> (a->c)
decompE1 f  = \ a -> case (f (Left a)) of
                       Left c -> c

decompE2 :: (Either a b -> Either c d) -> (b->d)
decompE2 f  = \ b -> case (f (Right b)) of
                       Right d -> d
```

B Defunctionalization Proofs

Theorem 1 follows straightforwardly from the two lemmas below. We use $\sigma_{apply'}$ as an abbreviation of function `apply'`'s type $Arrow'\ a_1\ a_2 \rightarrow a_1 \rightarrow a_2$. We assume types of X', *refl*, *to* and *from* are known and are omitted from the environment.

Lemma 1. *Let $\Gamma \vdash^{HM} e : t$ and $\Gamma \vdash^{DF} e : t \rightsquigarrow e'$. Then $[\![\Gamma]\!] \cup apply' : \sigma_{apply'} \vdash^{HM} e' : [\![t]\!]$.*

Proof. By structural induction on the derivation of $\Gamma \vdash^{DF} e : t \rightsquigarrow e'$.
∘ *Case* (Abs). The rule is

$$\frac{\begin{array}{c}\bar{y} = fv(\lambda x.e) \quad \bar{y} = y_1, ..., y_n \quad \Gamma \vdash^{HM} y_i : t_i \quad \text{for } i = 1, .., n \\ \Gamma \vdash^{HM} X' : (a_1 :=: [\![t_1]\!]) \rightarrow (a_2 :=: [\![t_2]\!]) \rightarrow [\![\bar{t}]\!] \rightarrow Arrow\ a_1\ a_2 \\ \Gamma \cup \{x : a_1, ep_1 : a_1 :=: t_1, ep_2 : a_2 :=: t_2\} \vdash^{DF} e : t_2 \rightsquigarrow e'\end{array}}{\Gamma \vdash^{DF} \lambda x.e : t_1 \rightarrow t_2 \rightsquigarrow X'\ refl\ refl\ \bar{y}}$$

Given $\Gamma \vdash^{HM} refl : a :=: a$, by rule (App), we have $\Gamma \vdash^{HM} X'\ refl\ refl\ \bar{y} : Arrow'\ [\![t_1]\!]\ [\![t_2]\!]$. Since $[\![t_1 \rightarrow t_2]\!] = Arrow'\ [\![t_1]\!]\ [\![t_2]\!]$, we conclude $\Gamma \vdash^{HM} X'\ refl\ refl\ \bar{y} : [\![t_1 \rightarrow t_2]\!]$.
∘ *Case* (App). The rule is

$$\frac{\Gamma \vdash^{DF} e_1 : t_2 \rightarrow t \rightsquigarrow e_1' \quad \Gamma \vdash^{DF} e_2 : t_2 \rightsquigarrow e_2'}{\Gamma \vdash^{DF} e_1\ e_2 : t \rightsquigarrow apply'\ e_1'\ e_2'}$$

By induction, we have

$$[\![\Gamma]\!].apply' : \sigma_{apply'} \vdash^{HM} e_1' : Arrow'\ [\![t_2]\!]\ [\![t]\!]$$

$$[\![\Gamma]\!].apply' : \sigma_{apply'} \vdash^{HM} e_2' : [\![t_2]\!]$$

By rule (App), we conclude $[\![\Gamma]\!].apply' : \sigma_{apply'} \vdash^{HM} apply'\ e_1'\ e_2' : [\![t]\!]$.

Lemma 2. $apply' : \sigma_{apply'} \vdash^{HM} \lambda f.\lambda arg.\mathsf{case}\ f\ \mathsf{of}\ \overline{patcl_x} : \sigma_{apply'}$.

Proof. Let's consider a lambda abstraction whose defunctionalization derivation ends with

$$\frac{\begin{array}{c}\bar{y} = fv(\lambda x.e) \quad \bar{y} = y_1, ..., y_n \quad \Gamma \vdash^{HM} y_i : t_i \quad \text{for } i = 1, .., n \\ \Gamma \vdash^{HM} X' : (a_1 :=: [\![t_1]\!]) \to (a_2 :=: [\![t_2]\!]) \to [\![\bar{t}]\!] \to Arrow\ a_1\ a_2 \\ \Gamma \cup \{x : a_1, ep_1 : a_1 :=: t_1, ep_2 : a_2 :=: t_2\} \vdash^{DF} e : t_2 \rightsquigarrow e'\end{array}}{\Gamma \vdash^{DF} \lambda x.e : t_1 \to t_2 \rightsquigarrow X'\ refl\ refl\ \bar{y}}$$

Applying Lemma 1 to the premise yields

$$[\![\Gamma]\!] \cup \{apply' : t_{apply'}, x : a_1, ep_1 : a_1 :=: [\![t_1]\!], ep_2 : a_2 :=: [\![t_2]\!]\} \vdash^{HM} e' : [\![t_2]\!]$$

Applying rule (App), we have

$$\begin{array}{c}[\![\Gamma]\!] \cup \{apply' : t_{apply'}, x : a_1, ep_1 : a_1 :=: [\![t_1]\!], ep_2 : a_2 :=: [\![t_2]\!]\} \vdash^{HM} \\ (to\ eq_2)\ e' : a_2\end{array}$$

Note that $patcl_x$ is of the form `X' ep`$_1$ `ep`$_2$ $\bar{y}$ `-> let x=arg in (to eq`$_2$`)` e'. Given the lambda variable `arg`'s type a_1. Pattern matching of `X'` produces the bindings $ep_1 : a_1 :=: [\![t_1]\!]$ and $ep_2 : a_2 :=: [\![t_2]\!]$. The let introduction gives us $x : a_1$.

By rules (Let) and (Clause), we have

$$\begin{array}{c}[\![\Gamma]\!] \cup \{apply' : t_{apply'}, arg : a_1\} \vdash^{HM} \\ X'\ ep_1\ ep_2\ \bar{y} \to \mathsf{let}\ x = arg\ \mathsf{in}\ (to\ eq_2)\ e' : Arrow'\ a_1\ a_2 \to a_2\end{array}$$

Since $\bar{y} = fv(\lambda x.e)$, we have

$$\begin{array}{c}\{apply' : t_{apply'}, arg : a_1\} \vdash^{HM} \\ X'\ ep_1\ ep_2\ \bar{y} \to \mathsf{let}\ x = arg\ \mathsf{in}\ (to\ eq_2)\ e' : Arrow'\ a_1\ a_2 \to a_2\end{array}$$

Because this holds for every lambda abstraction, we conclude

$$apply' : \sigma_{apply'} \vdash^{HM} \lambda f.\lambda arg.\mathsf{case}\ f\ \mathsf{of}\ \overline{patcl_x} : \sigma_{apply'}$$

Proof of Theorem 2.

Proof. By structural induction on the derivation of $\Gamma \vdash^{HM} e : t$.
∘ *Case* (Abs). The rule is

$$\frac{\Gamma \cup \{x : t_1\} \vdash^{HM} e : t_2}{\Gamma \vdash^{HM} \lambda x.e : t_1 \to t_2}$$

By induction, we have $\Gamma \cup \{x : t_1\} \vdash^{DF} e : t_2 \rightsquigarrow e'$. Since $\{ep_1, ep_2\} \cap fv(e) = \emptyset$, we have $\Gamma \cup \{x : t_1, ep_1 : a_1 :=: t_1, ep_2 : a_2 :=: t_2\} \vdash^{DF} e : t_2 \rightsquigarrow e'$.

If we apply rule (Eq$_1$) with eq_1 to every occurrence of x in e, we obtain

$$\Gamma \cup \{x : a_1, ep_1 : a_1 :=: t_1, ep_2 : a_2 :=: t_2\} \vdash^{DF} e : t_2 \rightsquigarrow e''$$

where e'' is e' with every occurrence of x substituted by `(from eq`$_1$`)` x. Thus, by rule (Abs), we conclude $\Gamma \vdash^{DF} \lambda x.e : t_1 \to t_2 \rightsquigarrow X'\ refl\ refl\ \bar{y}$.

C GADTless Explicit Coercion Proofs

The essence to GADTless encoding is the mimicking of GADT typing rule (Eq) with explicit coercions. A measurement of expressiveness of an encoding approach is how much valid (i.e. equivalent to identity) proof witnesses can be constructed comparing to constraint implication $\models$ (defined below) used in the GADT type system.

$$\text{(Sym)}\frac{t_1 = t_2 \in C}{C \models t_2 = t_1} \quad \text{(DCompA)}\frac{C \models t_1 \rightarrow t_3 = t_2 \rightarrow t_4}{C \models t_1 = t_2 C \models t_3 = t_4}$$

$$\text{(Arrow)}\frac{\begin{array}{c} C \models t_1 = t_2 \\ C \models t_3 = t_4 \end{array}}{C \models t_1 \rightarrow t_3 = t_2 \rightarrow t_4} \quad \text{(Comp)}\frac{\begin{array}{c} C \models t_i = t_i' \\ \text{for } i = 1, ..., n \end{array}}{C \models T\ t_1 ... t_n = T\ t_1' ... t_n'}$$

$$\text{(Trans)}\frac{\begin{array}{c} C \models t_1 = t_2 \\ C \models t_2 = t_3 \end{array}}{C \models t_1 = t_3} \quad \text{(DCompT)}\frac{C \models T\ t_1 ... t_n = T\ t_1' ... t_n'}{C \models t_i = t_i' \quad \text{for } i = 1, ..., n}$$

To prove the claim of Theorem 3, it is sufficient to show that all the above rules are expressible via explicit coercions given only decomposable types are used in decomposition. The following corresponding coercion functions constructed are equivalent to the identity operationally assuming the input functions are identity.

```
sym :: EQ a b -> EQ b a
sym (f,g) = (g,f)

trans :: EQ a b -> EQ b c -> EQ a c
trans (f₁,g₁) (f₂,g₂) = (f₂.f₁,g₁.g₂)

arrow :: EQ a₁ b₁ -> EQ a₂ b₂ -> EQ (a₁->a₂) (b₁->b₂)
arrow (f₁,g₁) (f₂,g₂) = (\ g -> f₂.g.g₁, \ g -> g₂.g.f₁)
```

W.l.o.g we assume each data constructor takes exactly one argument. No phantom variables are allowed given the decomposable requirement.

```
T ā = T₁ a₁
    | T₂ a₂
    ...
    | Tₙ aₙ

comp :: EQ a₁ t₁ -> EQ a₂ t₂ -> ... -> EQ aₙ tₙ -> EQ (T ā) (T t̄)
comp (f₁,g₁) (f₂,g₂) ... (fₙ,gₙ) = (f₁'.f₂'.....fₙ'), (g₁'.g₂'.....gₙ')
   where f₁' :: T a₁ ... aₙ -> T t₁ a₂ ... aₙ
         f₁' (T₁ x₁) = T₁ (f₁ x₁)
         f₁' x = x
         ...
         f_n' :: T a_1 a_2 ... a_n -> T a_1 ... a_{n-1} t_n
         f_n' (T_n x_n) = T_n (f_n x_n)
         f_n' x = x
         g_1' :: T t_1 a_2 ... a_n -> T a_1 ... a_n
         g_1' (T_1 x_1) = T_1 (g_1 x_1)
         g_1' x = x
         ...
         g_n' :: T a_1 ... a_{n-1} t_n -> T a_1 ... a_n
         g_n' (T_n x_n) = T_n (g_n x_n)
         g_n' x = x
```

Given the decomposable assumption, we only consider parameters at positive positions.

```
dcompA :: EQ (a_1->a_2) (b_1->b_2) -> EQ a_2 b_2
dcompA (f,g) = (\ x -> f (\ y -> x) undefined, \ x -> g (\ y -> x) undefined)
```

Due to laziness, evaluations of the above coercion functions do not diverge. However, it does not work in a strict language.

Decomposition can be defined as well by: 1. injecting the value into a data type whose conversion functions are available; 2. coercing the data type 3. projecting the data type which gives back the original value with a different type.

```
decompT_1 :: EQ (T $\bar{a}$) (T $\bar{t}$) -> EQ a_1 t_1
decompT_1 (f,g) = (\ x -> project_1 (f (inject_1 x), \ x -> project_1 (g (inject_1 x))
   where project_1 :: T a_1 ... a_n -> a_1
         project_1 (T_1 x_1) = x_1
         inject_1 :: a_1 -> T a_1 ... a_n
         inject_1 x_1 = (T_1 x_1)
...
decompT_n :: EQ (T $\bar{a}$) (T $\bar{t}$) -> EQ a_n t_n
decompT_n (f,g) = (\ x -> project_n (f (inject_n x), \ x -> project_n (g (inject_n x))
   where project_n :: T a_1 ... a_n -> a_n
         project_n (T_n x_n) = x_n
         inject_n :: a_n -> T a_1 ... a_n
         inject_n x_n = (T_n x_n)
```

Note that though a projection function $project_i$ is partial, it always works with data constructed by the corresponding injection function $inject_i$. Thus, coercion functions constructed above are total.