Subtyping with Strengthening Type Invariants

KOZSIK Tamás

Eötvös Loránd University, Budapest

Diederik VAN ARKEL, Rinus PLASMEIJER

"Clean group", University of Nijmegen,

The Netherlands

Motivation

- Development of safety critical applications
- Integration of
  - programming (coding)
  - proof of correctness (reasoning about the code)
- Make it in a usable way
  - easy to use
  - efficient

Vision

- Integrate a proof tool in the Clean environment
  - into the programming environment (IDE)
    prove properties while writing the program
    (these are often very simple properties)
  - into the run-time environment
    test properties of programs during run-time,
    e.g. enhance reliability of mobile code

Problem of efficiency

- A proof tool is very resource consuming e.g. takes a lot of time to complete a proof
- Sometimes a proof can be obtained with the help of the type system
  - Very simple: very fast
  - More complex: undecidable - dependent types
  - Everything in between

Key idea

- Program properties expressed as type invariants

  x: Natural   x: Integer with x >= 0

- Propagation of properties: verified by type system
  - If I add two Natural numbers, the result is also a Natural number
- Polymorphism is gained with subtyping
  - Natural is a special Integer, that is Natural $\leq$ Integer

Why (Concurrent) Clean?

- Functional language
  - referential transparency => simple maths
- Concurrency (?)
- Integrated Development Environment
  - Integrated proof tool for Clean progs (Sparkle)
- Efficient
  - Strictness annotations (evaluation order)
  - Uniqueness attributes (destructive updates)

What am I doing?

- Modify the type system of Clean
  - Add subtyping with type invariants
- Formalization + implementation
  - Clean 2.0 compiler offered by KUN

What are these subtypes for?

```
fac :: Int → Int
fac 0 = 1
fac n = n * fac (n-1)
```

What are these subtypes for?

```
fac :: Int → Int      // only for non-negative arg.
fac 0 = 1
fac n = n * fac (n-1)
```

What are these subtypes for?

fac :: Int → Int **fac :: Nat → Nat**

fac 0 = 1

fac n = n * fac (n-1)

- ... but there is no such type in Clean...

What are these subtypes for?

fac :: Int → Int

fac 0 = 1

fac n = n * fac (n-1)

fac :: Nat → Nat

- ... but there is no such type in Clean...
- Add a subtype mark!

fac :: <N> Int → <N> Int

// N(x) = (x>=0)

Subtype marks

- Notations to indicate some properties (type invariants, extra restrictions)
- The type system should work with them
- "Just" notations, not much more...
- Still, they can be used to derive/prove properties of code
- Especially propagation of type invariants
  - e.g. the identity function preserves any type invariants...

First-order logic in semantics

- We could assign logical formulas to these subtype marks

  N(x) = (x >= 0)

- This is not the business of the type system
- For the type system, subtype marks do not have such meaning: "just notations"
- Handle formulas:
  - proof system (mathematical proof of correctness)
  - run-time system
    (run-time check, like in Alphard or Eiffel)

Currently

- Just the type system, no logical formulas
- They are still good for certain things
  - localize dangerous code

fac :: Nat $\rightarrow$ Nat

abs :: Int $\rightarrow$ Nat

fac (abs x)   is not dangerous

One day…

- Generate code that checks type invariants run-time, namely before and after evaluating a function
- Use a proof system to argue about type invariants

Believe-me marks

- Believe me, that this property holds. What else can you guarantee based on this?
- Maybe prove (sub)type correctness of other functions...
- Later those believe-me marks should be investigated by a proof system or a run-time check

For example, sorting...

```
insert ::    a   <S>[a]   →   <S!>[a]     | < a
insert e [] =  [e]
insert e [x:xs] =   if (e <= x)  [e,x:xs]
                                 [x: insert e xs]


sort ::    [a]   →   <S>[a]   | < a
sort [] = []
sort [x:xs] = insert x (sort xs)
```

Subtype assertions for algebraic data constructor symbols

- In non-pattern expressions (composing)

$$[] \quad :>: \quad <S>[a]$$

$$[:] \quad :>: \quad a \quad [a] \quad \rightarrow \quad [a]$$

- In pattern expressions (decomposing)

$$[] \quad :<: \quad [a]$$

$$[:] \quad :<: \quad a \quad <S>[a] \quad \rightarrow \quad <S>[a]$$

$$[:] \quad :<: \quad a \quad [a] \quad \rightarrow \quad [a]$$

Polymorphic subtype marks

- Multiple "standard" types (monomorphic)

plus :: Int Int → Int
plus :: <N>Int <N>Int → <N>Int

- Polymorphic subtype marks meaning the same

plus :: <N a>Int <N a>Int → <N a>Int

Interfere with other things

- Overloading polymorphism (type classes)
- Synonym types
- Uniqueness typing
- Built-in type constructors
- Existentially and universally quantified types
- Dynamic types
- Syntactic sugar
- Module system, ADT-s

Theory already done

- Formalization of subtyping
  - Like uniqueness “subtyping”
  - Data constructor assertions, restrictions
- Properties of the type system
  - Subject reduction, principal typing
  - Without believe-me marks

Ideas about implementation

- Type derivation with interaction from the programmer
- Aspect-oriented approach to add subtypes to the program
  - turn on / turn off
    - in editor
    - in compiler
  - like turning on/off the run-time checks

Future plans

- Not only first-order logic in describing properties, but also temporal logic
  - argue about safety and progress properties
  - verify concurrent/distributed applications
- Checking mobile code run-time: dynamics
  - e.g. obtained from Internet
  - currently type-checks are more or less ready
  - proof checks: prototype

Plans for me

- Finish this implementation (catch up with theory)
- Increase expressive power
- Eliminate interference with other language concepts not addressed in theory
- Develop large examples (case studies)
- Integrate with proof tool, do run-time checks
- **Get the PhD**