Subtyping with Strengthening Type Invariants

KOZSIK Tamás
Eötvös Loránd University, Budapest

Diederik VAN ARKEL, Rinus PLASMEIJER
"Clean group", University of Nijmegen,
The Netherlands

Supported by OTKA T037742.

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 simploe properties)
  - into the run-time environment
    reason about programs during run-time
    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: indecidable (infinite run)
    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

Intro to Clean

- Functional programming language
  - lazy, pure, polymorphic, higher-order
  - semantics based on Term Graph Rewriting Systems
- Program = collection of function definitions + an expression to evaluate (khmm...)
- No assignment, no "imperative variables", only "mathematical" ones
  - variable: sg. that can hold an arbitrary value of a certain type
- Program execution: evaluation of the Start expression

Why is it good?

- A program is an executable specification
- Just maths...
- Easy to learn FP, easy to do FP

- Referential transparency: no side effects
  - less error-prone
  - better quality software: understand/modify/reuse
- Easy to reason about programs formally
  - mathematical proofs use referential transparency

Clean is much more than that

- High-level language constructs
- High expressive power
- Fancy syntax (?)
- **Efficient**
- Large libraries
- Integrated Development Environment, etc.

- Suitable for writing real-world apps

Some features

- Predefined type constructs: lists, tuples, arrays, records, functions
- Functions are first-class citizens
  - higher-order
- Flexible type system: algebraic types, parametric polymorphism, type classes, type constructors (higher-order types)
- Strictness annotations (evaluation order)
- Uniqueness attributes (destructive updates)

Some more...

- Strong type checking
- Type inference
- Modules
- Block structure
- Abstract data types
- Generic programming
- Dynamic typing
- Object IO for the devel. of graphical apps

Example: quicksort

```
module qsort

qsort [] = []
qsort [x:xs] =
        qsort [a \\ a <- xs | a < x]
        ++ [x] ++
        qsort [a \\ a <- xs | a > x]

Start = qsort [42, 33, 100, 15]
```

Type declaration

```
qsort :: [a] -> [a]  |  < a
qsort [] = []
qsort [x:xs] =
        qsort [a \\ a <- xs | a < x]
        ++ [x] ++
        qsort [a \\ a <- xs | x > a]
```

What am I doing?

- Modify the type system of Clean
- Add subtyping with type invariants
- Clean 2.0 compiler offered by KUN
  - source code is available
  - ... in Clean ... :-)
- Theory + implementation
- Hoping to do sg. useful, practical

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)
```

- Here the program aborts for negative numbers
- Things can be worse
  (do harmful computation)

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

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 -> Nat
abs :: Int -> Nat

fac (abs x)    is not dangerous
```

Later

- Generate code that checks type invariants run-time, namely before and after evaluating a function (several examples...)
- Use a proof system to argue about type invariants
  - Special proof system (dedicated to Clean): Sparkle (formerly *C*lean *P*rover *S*ystem)
    - reason about Clean progs, no transformation
    - integrated with IDE

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)

Things not addressed here

- Subtype assertions for algebraic data constructor symbols

  [] :: <S>[a]

- Multiple "standard" types (monomorphic)

  plus :: Int Int -> Int

  plus :: <N>Int <N>Int -> <N>Int

- Polymorphic subtype marks

  plus :: <N a>Int <N a>Int -> <N a>Int

Implementation difficulties

- The Clean compiler is written in Clean
- The front-end is about 50.000 lines (2.500.000 characters)
- Clean programs are shorter than corresponding C programs
  - Rinus says: only one tenth
- Actually, it is not a very nice code... (hacking, not too much abstraction, no comments, no documentation)

How I do the implementation

- I need to change about 10 modules heavily
- 10 more modules only a little bit

- I do not know what they do...

Main activities

- Scanning
- Parsing
- Collect info
  - syntax tree
  - symbol tables
- Check visibility, etc.
- Type checking / inferencing (unification)

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

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
  – e.g. obtained from Internet
  – currently type-checks are being implemented by the Clean group - we want more!

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**