The Original Refinement Types

"Refinement Types for ML"

水野 雅之

2015 年 5 月 20 日

Outline

1. Motivation
2. Introduction
3. The Language
4. From Rectype Declarations to Datatype Lattices
5. From Datatype Lattice to Function Types
6. Polymorphism
7. The Type Inference Algorithm

Outline

1. Motivation
2. Introduction
3. The Language
4. From Rectype Declarations to Datatype Lattices
5. From Datatype Lattice to Function Types
6. Polymorphism
7. The Type Inference Algorithm

Motivation

Nowadays refinement types are popular

$$\Lambda\pi : \tau.\ e \qquad \{x : T \mid P(x)\}$$

- Isn't it dependent types?
- Too many extensions to see through the essence

Try to read original paper ("Refinement Types for ML")

Outline

1. Motivation
2. Introduction
3. The Language
4. From Rectype Declarations to Datatype Lattices
5. From Datatype Lattice to Function Types
6. Polymorphism
7. The Type Inference Algorithm

Refinement Types?

"We describe a refinement of ML's type system allowing the specification of recursively defined subtypes of user-defined datatypes."

- Preserve desirable properties of ML's type system
  - Decidability of type inference
  - Every well-typed expression has a principal type
- Allow more errors to be detected at compile-time

Opportunity to Improve

```
datatype 'a list = nil | cons of 'a * 'a list
fun lastcons (last as cons(hd,nil)) = last
  | lastcons (cons(hd, tl)) = lastcons tl
```

- undefined when called on nil

```
case lastcons y of
     cons(x,nil) => print x
```

- ML type system does not distinguish singleton lists
- Get compiler warning

Solution

```
datatype 'a list = nil | cons of 'a * 'a list
rectype 'a singleton = cons('a, nil)
```

$$
\begin{array}{rlcll}
\mathtt{cons}: & (\alpha * \alpha\ ?\mathtt{nil}) & \rightarrow & \alpha\ \mathtt{singleton} & \wedge \\
 & (\alpha * \alpha\ \mathtt{singleton}) & \rightarrow & \alpha\ \mathtt{list} & \wedge \\
 & (\alpha * \alpha\ \mathtt{list}) & \rightarrow & \alpha\ \mathtt{list} &
\end{array}
$$

- rectype declaration stand for subtypes
- α singleton $= \{\mathtt{cons}(\alpha, \mathtt{nil}) \mid a \in \alpha\} \subset \alpha$ list
- Abstract interpretation over lattice (p. 2)
- Intersection type (only refinement types)

Expressivity

There are examples which cannot be specified as refinement types

- List without repeated elements
- Closed term in λ-calculus

`rectype` specify so-called regular tree sets

Outline

1. Motivation
2. Introduction
3. The Language
4. From Rectype Declarations to Datatype Lattices
5. From Datatype Lattice to Function Types
6. Polymorphism
7. The Type Inference Algorithm

Rectype Declarations

$$
\begin{array}{lcl}
\textit{rectype} & ::= & \texttt{rectype}\ \textit{rectypedecl} \\
\textit{rectypedecl} & ::= & <\textit{mltypevar}>\ \ \textit{reftyname} = \textit{recursivety}\ \ <\texttt{and}\ \textit{rectypedecl}> \\
\textit{recursivety} & ::= & \textit{recursivety} \mid \textit{recursivety} \\
 & & \textit{mlty} \rightarrow \textit{recursivety} \\
 & & \textit{constructor recursivetyseq} \\
 & & \textit{mltypevar} \\
 & & <\textit{mltypevar}>\ \ \textit{reftyname}
\end{array}
$$

Each rectype declaration must be consistent with the ML datatype

- e.g. $\texttt{rectype}\ \alpha\ \texttt{bad} = \texttt{nil}(\texttt{nil})$

Recursive type's definition must have the same type variable argument

Refinement Types

$$
\begin{array}{rcl}
\textit{refty} & ::= & \textit{refty} \wedge \textit{refty} \\
& & \textit{refty} \vee \textit{refty} \\
& & \textit{refty} \rightarrow \textit{refty} \\
& & \perp \\
& & < \textit{refty} > \textit{mltyname} \\
& & < \textit{refty} > \textit{reftyname} \\
& & < \textit{refty} > \textit{reftyname} \\
& & \textit{reftyvar} :: \textit{mltyvar}
\end{array}
$$

Refinement type variable is bounded by an ML type variable

Ranges only over the refinements of an ML type

An Example

Representation of natural numbers in binary

```
datatype bitstr =
   e | z of bitstr | o of bitstr
```

We would like to guarantee that zero does not appear in MSB (standard form, `std`)

```
rectype std = e | stdpos
and stdpos = o(e) | z(stdpos) | o(stdpos)
```

```
fun add e m = m
  | add n e = n
  | add (z n) (z m) = z (add n m)
  | add (o n) (z m) = o (add n m)
  | add (z n) (o m) = o (add n m)
  | add (o n) (o m) = z (add (add (o e) n) m)
```

`Infered type (See p. 4)`

Outline

1. Motivation
2. Introduction
3. The Language
4. From Rectype Declarations to Datatype Lattices
5. From Datatype Lattice to Function Types
6. Polymorphism
7. The Type Inference Algorithm

From Rectype Declarations to Datatype Lattices

```
datatype bitstr =
    e | z of bitstr | o of bitstr
rectype std = e | stdpos
and stdpos = o(e) | z(stdpos) | o(stdpos)
```

Manipulating regular tree grammers, we can infer:

- These refinement types are closed under intersection and union (See p. 5)
- They form lattice

Types for the constructors are calculated as

$$
\begin{aligned}
\mathtt{e} &: \ ?\mathtt{e} \\
\mathtt{o} &: \ ?\mathtt{e} \to \mathtt{stdpos} \wedge \mathtt{stdpos} \to \mathtt{stdpos} \\
\mathtt{z} &: \ \mathtt{stdpos} \to \mathtt{stdpos}
\end{aligned}
$$

Outline

1. Motivation
2. Introduction
3. The Language
4. From Rectype Declarations to Datatype Lattices
5. From Datatype Lattice to Function Types
6. Polymorphism
7. The Type Inference Algorithm

Normal Form

After apply the following rewrite rules,

$$
\begin{array}{lcl}
\rho \wedge (\sigma \vee \tau) & \Rightarrow & (\rho \wedge \sigma) \vee (\rho \wedge \tau) \\
(\rho \vee \sigma) \to \tau & \Rightarrow & (\rho \to \tau) \wedge (\sigma \to \tau)
\end{array}
$$

The refinement types will fit the grammar

$$
\begin{array}{lcl}
\mathit{unf} & ::= & \mathit{inf} \\
 & & \mathit{unf} \vee \mathit{unf} \\
\mathit{inf} & ::= & < \mathit{unf} > \ \ \mathit{reftyname} \\
 & & \mathit{inf} \wedge \mathit{inf} \\
 & & \mathit{inf} \to \mathit{unf} \\
 & & \mathit{reftyvar} :: \mathit{mltyvar}
\end{array}
$$

Subtyping Rule for Arrow

"$\to$" is contravariant in its first argument and covariant in its second argument

$$\frac{\tau_1 \leq \tau_2 \qquad \sigma_2 \leq \sigma_1}{\sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2}\ \textsc{S-Arrow}$$

Datatype constructor may also be covariant or contracariant in their arguments

- `stdpos list` $\leq$ `std list`

Subtyping Rule for Union

$$\frac{\text{If for each } \sigma_i \text{ there is a } \sigma'_j \text{ such that } \sigma_i \leq \sigma'_j}{\sigma_1 \vee \sigma_2 \vee \ldots \vee \sigma_n \leq \sigma'_1 \vee \sigma'_2 \leq \ldots \vee \sigma'_m} \textsc{S-Union}$$

σ_i, σ'_j : inf refinement types

Refinement Type of Application

Function has type $\sigma = (\rho_1 \to \tau_1) \wedge (\rho_2 \to \tau_2) \wedge \ldots \wedge (\rho_n \to \tau_n)$
Argument has type ρ
Their application $\mathtt{apptype}(\sigma, \rho)$

$$\mathtt{apptype}(\sigma, \rho) = \bigwedge_{\{i \mid \rho \leq \rho_i\}} \tau_i$$

Outline

1 Motivation

2 Introduction

3 The Language

4 From Rectype Declarations to Datatype Lattices

5 From Datatype Lattice to Function Types

6 Polymorphism

7 The Type Inference Algorithm

Polymorphism

The domain of type variable is restricted to range over subtypes of given bound (bounded quantification)

$$\begin{array}{rcl} \textit{reftyscheme} & ::= & \textit{inf} \\ & & \forall\alpha.\ \textit{reftyscheme} \\ & & \forall r\alpha :: \alpha.\ \textit{reftyscheme} \end{array}$$

Refinement type for identity function id:

$$\forall\alpha.\forall r\alpha :: \alpha.\ r\alpha \rightarrow r\alpha$$

Instantiate α to bitstr and instantiate the refinement type quantifier

$$\begin{array}{ccccc} \texttt{bitstr} & \rightarrow & \texttt{bitstr} & \wedge \\ \texttt{stdpos} & \rightarrow & \texttt{stdpos} & \wedge \\ \texttt{std} & \rightarrow & \texttt{std} & \wedge \\ \texttt{?e} & \rightarrow & \texttt{?e} & \wedge \\ \bot & \rightarrow & \bot & \end{array}$$

Outline

1. Motivation
2. Introduction
3. The Language
4. From Rectype Declarations to Datatype Lattices
5. From Datatype Lattice to Function Types
6. Polymorphism
7. The Type Inference Algorithm

Target Language

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \mathit{variable} \\
 & & \mathit{exp}\ \mathit{exp} \\
 & & \lambda\ \mathit{variable}.\ \mathit{exp} \\
 & & \mathit{exp} : \mathit{refty} \\
 & & \texttt{let}\ \mathit{variable} = \mathit{exp}\ \texttt{in}\ \mathit{exp} \\
 & & \texttt{fix}\ \mathit{variable}.\mathit{exp}
\end{array}
$$

Typing derivation	$\Gamma \vdash e : D :: L$
Meaning of metavariables	See p. 7

The Type Inference Algorithm

type inference by unification just like ML types

Typing rules : See Figure 2

Eliminate all of the refinement, corresponds to ML types

Theorem 1 (preservation)

For all valid type environments Γ and expressions e, if e evaluates to v and $\Gamma \vdash e : D :: L$ then $\Gamma \vdash v : D' :: L$ for some $D' \leq D$.

Proof.

By induction on the structure of the definition of the "evaluates to" relation □

Manipulating Case Statement

Implicitly define a new constant CASE_*datatype*

```
case E of
      e => E1
   | o(m) => E2
   | z(m) => E3
```

```
CASE_bitstr E
   (fn () => E1)
   (fn (m) => E2)
   (fn (m) => E3)
```

Type for `CASE_bitstr`: See Figure 1