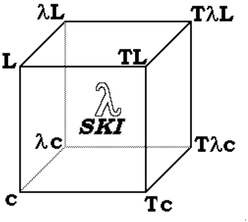
Basic Type Inference

Alex Vondrak

ajvondrak@csupomona.edu

October 6, 2010



Motivation

Example (Ever Get Sick of This?)

```
Map<String, AbstractFoo<? super C>> fooMap =
  new HashMap<String, AbstractFoo<? super C>>();
```

- Don't want to write so many type declarations. . .
- . . . But want compile-time safety guarantees
- Enter type inference

The Idea

Before

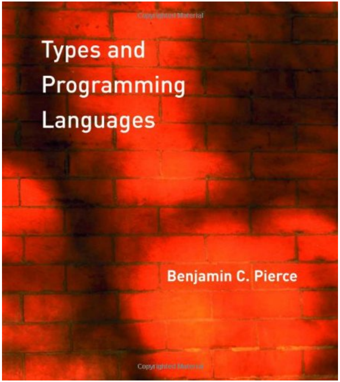
```
public int add(int x, int y) {
  return x + y;
}
```

After

```
public add(x, y) {
  return x + y;
}
```

- Compiler still infers the `int` types

Outline



Ch. 22 Type Reconstruction

- Ch. 2 Mathematical Preliminaries
- Ch. 3 Untyped Arithmetic Expressions
- Ch. 8 Typed Arithmetic Expressions
- Ch. 9 Simply Typed Lambda-Calculus

Ch. 11 ~~Simple Extensions~~

1. Simply-Typed λ-Calculus

2. Constraint Generation

3. Unification

Untyped λ-Calculus

- Minimal system of function definition and application
- Defines anonymous functions
- Treats functions as first-class values

Examples

Algebra	λ-Calculus
$f(x) = x$	$f = \lambda x\ .\ x$
$f(5)$	$(f\ 5)$ OR $((\lambda x\ .\ x)\ 5)$
$g(h) = h(5)$	$g = \lambda h\ .\ (h\ 5)$
$g(f) \twoheadrightarrow f(5) \twoheadrightarrow 5$	$((\lambda h\ .\ (h\ 5))\ (\lambda x\ .\ x)) \twoheadrightarrow ((\lambda x\ .\ x)\ 5) \twoheadrightarrow 5$

Simply-Typed λ-Calculus

- Untyped λ-calculus + declarations
- Since we have types, extend with corresponding values
  - Booleans (`true`, `false`) of type `Bool`
  - Natural numbers $(0, 1, 2, \ldots)$ of type `Nat`
- Function types have the form $T \rightarrow T$

Examples

Untyped	Typed
$\lambda x\ .\ x$	$(\lambda x\colon \mathtt{Nat}\ .\ x)\colon \mathtt{Nat} \rightarrow \mathtt{Nat}$
$\lambda h\ .\ (h\ 5)$	$(\lambda h\colon \mathtt{Nat} \rightarrow \mathtt{Nat}\ .\ (h\ 5))\colon (\mathtt{Nat} \rightarrow \mathtt{Nat}) \rightarrow \mathtt{Nat}$
$\lambda h\ .\ (h\ 5)$	$(\lambda h\colon \mathtt{Nat} \rightarrow \mathtt{Bool}\ .\ (h\ 5))\colon (\mathtt{Nat} \rightarrow \mathtt{Bool}) \rightarrow \mathtt{Bool}$

Type Variables

- Variables ($x, y, z, \ldots$) range over values ($\texttt{true}, \texttt{false}, 0, 1, 2, \ldots$)
- Type variables (A, B, C, $\ldots$) range over concrete types ($\texttt{Nat}$ & $\texttt{Bool}$)

Examples

Untyped	Typed (with type variables)
$\lambda x \ . \ x$	$(\lambda x \colon T \ . \ x) \colon T \to T$
$\lambda h \ . \ (h \ 5)$	$(\lambda h \colon \texttt{Nat} \to T \ . \ (h \ 5)) \colon (\texttt{Nat} \to T) \to T$

Making Things Interesting

- Extend simply-typed λ-calculus with a few “built-ins”
- Note: not the extensions from Chapter 11 of TAPL

Definitions

$$(\texttt{succ}\ n) \equiv n + 1$$

$$(\texttt{pred}\ n) \equiv \begin{cases} n - 1 & (n > 0) \\ 0 & (n = 0) \end{cases}$$

$$(\texttt{iszero}\ n) \equiv \begin{cases} \texttt{true} & (n = 0) \\ \texttt{false} & (n \neq 0) \end{cases}$$

Also, `if` … `then` … `else` … works as you'd expect

Running Example

- Actually, the idea is not to erase type declarations
- Instead, let them be variables

```
f = λ g: A → B . if (iszero (g (succ 0)))
                 then false
                 else true
```

- What are the values (concrete types) of A and B?
- What type does f have?
- How do we figure it out algorithmically?

1. Simply-Typed λ-Calculus
2. Constraint Generation
3. Unification

Constraint Typing Relation

$$\Gamma \vdash t : T \mid_{\mathcal{X}} C$$

Γ	Typing context
t	Simply-typed λ-calculus term
T	Type of t
$\mathcal{X}$	Set of type variables
C	Constraint set

Axioms and the Structure of Typing Rules

- "0 is a Nat"

$$\textsc{Zero}\;\frac{}{\Gamma \vdash \mathtt{0} : \mathtt{Nat} \mid_{\varnothing} \varnothing}$$

- "true is a Bool"

$$\textsc{True}\;\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{Bool} \mid_{\varnothing} \varnothing}$$

- "false is a Bool"

$$\textsc{False}\;\frac{}{\Gamma \vdash \mathtt{false} : \mathtt{Bool} \mid_{\varnothing} \varnothing}$$

Variables and the Typing Context, Γ

- The typing context keeps a record of the types of variables

$$\textsc{Var}\,\frac{x\colon T \in \Gamma}{\Gamma \quad \vdash \quad x \quad : \quad T \quad \mid_{\varnothing} \quad \varnothing}$$

- So Γ is a set of pairs, $x\colon T$
- Later, we use the syntax

$$\Gamma, x\colon T$$

to mean something like

$$\Gamma \cup \{x\colon T\}$$

Built-in Functions and the Constraint Set, C

- "`iszero` results in a `Bool` and should be applied to a `Nat`"

$$\textsc{IsZero}\;\frac{\Gamma \vdash t : T \mid_{\mathcal{X}} C}{\Gamma \vdash (\texttt{iszero}\ t) : \texttt{Bool} \mid_{\mathcal{X}} C \cup \{T = \texttt{Nat}\}}$$

- "`succ` results in a `Nat` and should be applied to a `Nat`"

$$\textsc{Succ}\;\frac{\Gamma \vdash t : T \mid_{\mathcal{X}} C}{\Gamma \vdash (\texttt{succ}\ t) : \texttt{Nat} \mid_{\mathcal{X}} C \cup \{T = \texttt{Nat}\}}$$

- "`pred` results in a `Nat` and should be applied to a `Nat`"

$$\textsc{Pred}\;\frac{\Gamma \vdash t : T \mid_{\mathcal{X}} C}{\Gamma \vdash (\texttt{pred}\ t) : \texttt{Nat} \mid_{\mathcal{X}} C \cup \{T = \texttt{Nat}\}}$$

Function Application and the Type Variable Set, $\mathcal{X}$

$$\textsc{App}\,\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \mid_{\mathcal{X}_1} C_1 \\ \Gamma \vdash t_2 : T_2 \mid_{\mathcal{X}_2} C_2 \\ \mathcal{X}_1 \cap \mathcal{X}_2 = \mathcal{X}_1 \cap FV(T_2) = \mathcal{X}_2 \cap FV(T_1) = \varnothing \\ X \notin \mathcal{X}_1, \mathcal{X}_2, T_1, T_2, C_1, C_2, \Gamma, t_1, \text{ or } t_2\end{array}}{\Gamma \vdash (t_1\ t_2) : X \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{X\}} C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}}$$

If Statements

$$\textsc{If}\;\frac{\begin{array}{cccccc}\Gamma & \vdash & t_1 & : & T_1 & \mid_{\mathcal{X}_1} C_1 \\ \Gamma & \vdash & t_2 & : & T_2 & \mid_{\mathcal{X}_2} C_2 \\ \Gamma & \vdash & t_3 & : & T_3 & \mid_{\mathcal{X}_3} C_3 \\ \multicolumn{6}{c}{\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \text{ non-overlapping}}\end{array}}{\Gamma \vdash (\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3) : T_2 \mid_{\bigcup \mathcal{X}_i} \bigcup C_i \cup \{ T_1 = \texttt{Bool}, T_2 = T_3 \}}$$

Note

We type `if` ... `then` ... `else` ... conservatively by enforcing that the `then` and `else` have the same types.

Function Definition

$$\textsc{Abs}\frac{\Gamma, x : T_1 \quad \vdash \quad t_2 \quad : \quad T_2 \quad |_{\mathcal{X}} \quad C}{\Gamma \quad \vdash \quad (\lambda\, x : T_1 \,.\, t_2) \quad : \quad T_1 \to T_2 \quad |_{\mathcal{X}} \quad C}$$

- We can "pull out" a variable from the typing context…
- …Then "wrap" the original term in a λ abstraction
- This introduces no new constraints

1. Simply-Typed λ-Calculus
2. Constraint Generation
3. Unification

Finding a Solution

- The constraint set gave us equations of type variables
- Now we solve them

Definition

A substitution

$$[A \mapsto B]$$

replaces every A with B. $[\ ]$ is the identity substitution.

Example

$$\begin{aligned} [A \mapsto B] \quad & (\lambda x \colon A \,.\, \lambda y \colon B \,.\, \ldots) \\ = \quad & (\lambda x \colon B \,.\, \lambda y \colon B \,.\, \ldots) \end{aligned}$$

Unification Algorithm

```
unify($C$) =
  if    $C$==$\emptyset$
  then  [ ]
  else
      let  $(S = T) \in C$
      let  $C'$ = $C \setminus \{S = T\}$
      if          $S$ == $T$
      then        unify($C'$)
      else if     $S$==$X \wedge X \notin FV(T)$
      then        unify([$X \mapsto T$]$C'$) $\circ$ [$X \mapsto T$]
      else if     $T$==$X \wedge X \notin FV(S)$
      then        unify([$X \mapsto S$]$C'$) $\circ$ [$X \mapsto S$]
      else if     $S$==$S_1 \rightarrow S_2 \wedge T$==$T_1 \rightarrow T_2$
      then        unify($C' \cup \{S_1 = T_1, S_2 = T_2\}$)
      else        fail
```

Conclusion

- We worked through a basic derivation using Hindley-Milner type inference
- More complex “in the wild”
    - Haskell
    - ML family (SML, OCaml, etc.)
- Idea is still simple
    - Set up equations
    - Solve equations
- See *Types and Programming Languages* for more