λ Calculus: Lecture 7

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

PLC, 27 March 2017

Type inference as equation solving

What is the type of `twice f x = f (f x)`?

- Generically, `twice :: a -> b -> c`
- We then reason as follows

`a`	=	`d -> e`	(because `f` is a function)
`b`	=	`d`	(because `f` is applied to `x`)
`e`	=	`d`	(because `f` is applied to `(f x)`)
`c`	=	`e`	(because output of `twice` is `f (f x)`)

- Thus `b` = `c` = `d` = `e` and `a` = `b -> b`
- Most general type is `twice :: (b -> b) -> b -> b`

Unification

- Start with a system of equations over terms
- Find a substitution for variables that satisfies the equation
- Least constrained solution : most general unifier (mgu)

Unification algorithm

1. $t = X$, t is not a variable $\rightsquigarrow$ $X = t$.
2. Erase equations of form $X = X$.
3. Let $t = t'$ where $t = f(\ldots)$, $t' = f'(\ldots)$
   - $f \neq f' \rightsquigarrow$ terminate : unification not possible
   - Otherwise, $f(t_1, t_2, \ldots, t_k) = f(t'_1, t'_2, \ldots, t'_k)$

     Replace by k new equations

     $$t_1 = t'_1, t_2 = t'_2, \ldots, t_k = t'_k$$
4. $X = t$, X occurs in $t \rightsquigarrow$ terminate: unification not possible
5. $X = t$, X does not occur in t, X occurs in other equations
   $\rightsquigarrow$ Replace all occurrence of X in other equations by t.

Unification algorithm : Examples

$$\begin{array}{lcl} f(X) & = & f(f(a)) \\ g(Y) & = & g(Z) \end{array}$$

$$\begin{array}{lcl} X & = & f(a) \\ g(Y) & = & g(Z) \end{array}$$

$$\begin{array}{lcl} X & = & f(a) \\ Y & = & Z \end{array}$$

mgu is $\{X \leftarrow f(a), Z \leftarrow Y\}$

Unification algorithm : Examples . . .

$$
\begin{array}{lcl}
g(Y) & = & X \\
f(X, h(X), Y) & = & f(g(Z), W, Z)
\end{array}
$$

$$
\begin{array}{lcl}
X & = & g(Y) \\
f(X, h(X), Y) & = & f(g(Z), W, Z)
\end{array}
$$

$$
\begin{array}{lcl}
X & = & g(Y) \\
X & = & g(Z) \\
h(X) & = & W \\
Y & = & Z
\end{array}
$$

$$
\begin{array}{lcl}
g(Z) & = & g(Y) \\
X & = & g(Z) \\
h(g(Z)) & = & W \\
Y & = & Z
\end{array}
$$

Unification algorithm : Examples . . .

$$
\begin{array}{lcl}
Z & = & Y \\
X & = & g(Z) \\
h(g(Z)) & = & W \\
Y & = & Z
\end{array}
$$

$$
\begin{array}{lcl}
Z & = & Z \\
X & = & g(Z) \\
h(g(Z)) & = & W \\
Y & = & Z
\end{array}
$$

$$
\begin{array}{lcl}
X & = & g(Z) \\
W & = & h(g(Z)) \\
Y & = & Z
\end{array}
$$

Equations : $g(Y) = X,\ f(X, h(X), Y) = f(g(Z), W, Z)$

mgu : $\{X \leftarrow g(Z), W \leftarrow h(g(Z)), Y \leftarrow Z\}$

Unification algorithm : Correctness

- The algorithm terminates
  - Rules 1–4 can be used only a finite number of times without using Rule 5
  - Rule 5 can be used at most once for each variable
- When the algorithm terminates, all equations are of the form $X_i = t_i$. This defines a substitution

$$\{X_1 \leftarrow t_1, X_2 \leftarrow t_2, \ldots, X_n \leftarrow t_n\}$$

- This substitution is a unifier
  - Every transformation preserves the set of unifiers
- This substitution is an mgu
  - More complicated, omit

Type inference with shallow types

Syntax

- Built-in types $i, j, k, \ldots$
- A set of constants C_i for each built-in type i
  - e.g., $i = \texttt{Char}$, $C_i = \{\texttt{'a'}, \texttt{'b'}, \ldots\}$
- λ-terms

$$\Lambda = c \mid x \mid \lambda x.M \mid MN$$

Type inference with shallow types

- $M = c \in C_i \rightsquigarrow M :: i$
- $M = x \rightsquigarrow M :: \alpha$ for a fresh type variable α
- $M = \lambda x.M' \rightsquigarrow M :: \alpha \to \beta$ for fresh type variables α, β.
  - Inductively, $x :: \gamma$ in M'
  - Add equation $\alpha = \gamma$
- $M = M'N' \rightsquigarrow M :: \beta$ for fresh type variables β.
  - Inductively, $M' :: \alpha \to \beta$, $N' :: \gamma$
  - Add equation $\alpha = \gamma$

Type inference with shallow types

Consider

```
applypair f x y = (f x,f y)
```

Is the following expression well typed, where `id z = z`?

```
applypair id 7 'c' = (id 7, id 'c') = (7,'c')
```

We have to unify the following set of constraints

```
id  :: a -> a
7   :: Int
'c' :: Char
a = Int              (from id 7)
a = Char             (from id 'c')
```

Not possible! Haskell compiler says

```
applypair :: (a -> b) -> a -> a -> (b,b)}
```

Type inference with shallow types

In the λ-calculus, we have

$$\lambda fxy.pair\ (fx)(fy), \text{ where } pair \equiv \lambda xyz.(zxy)$$

When we pass a value for f, it has to unify with types of both x and y

- Every argument must have the same type across all copies

Suppose, we write, instead

```
applypair x y = (f x,f y) where f z = z
```

Now, we have

```
applypair :: a -> b -> (a,b)
```

What's going on?

Type inference with shallow types

Extend λ-calculus with "local" definitions, like `where`

$$\Lambda = C_i \mid x \mid \lambda x.M \mid MN \mid \text{let } f = e \text{ in } M$$

Here is the λ-term for the second version of `applypair`

$$\text{let } f = \lambda z.z \text{ in } \lambda xy.pair\ (fx)(fy)$$

In fact, Haskell allows both

```
let f z = z in applypair x y = (f x,f y)
```

and

```
applypair x y = (f x,f y) where f z = z
```

Type inference with shallow types

- let $f = e$ in $\lambda x.M$ and $(\lambda fx.M)e$ are equivalent with respect to β-reduction
- . . . but type inference works differently for the two
- One may be typeable while the other is not
  - $(\lambda I.(II))(\lambda x.x)$
  - let $I = \lambda x.x$ in (II)

Type inference with shallow types

Type inference for $M = \text{let } f = e \text{ in } M'$

First attempt

- Assume $f :: t$ where $\alpha, \beta, \ldots$ are type variables occurring in t
- Make a separate copy of type variables for each instance of f in M'

Example

- let $f = \lambda z.z$ in $\lambda xy.pair\ (fx)(fy)$
- First instance of f has type $\alpha_1 \to \beta_1$
- Second instance of f has type $\alpha_2 \to \beta_2$

Type inference with shallow types

A subtle problem

```
applypair2 w x y = ((tag x),(tag y))
  where
    tag      = pair w
    pair s t = (s,t)
```

- `applypair2 w x y` $\rightsquigarrow$ `((w,x),(w,y))`
- Type should be
  `applypair2 :: a -> b -> c -> ((a,b),(a,c))`

Type inference with shallow types

```
applypair2 w x y = ((tag x),(tag y))
  where
    tag      = pair w
    pair s t = (s,t)
```

Type inference

```
applypair2 :: a -> b -> c -> (d,e)
pair       :: f -> g -> (f,g)
tag        :: h -> (i,h)
```

- `a = i` because `tag` uses input `w` from `applypair2`
- Using `let` rule, two instances of `tag` get different types
  - `d = h1 -> (i1,h1)`
  - `e = h2 -> (i2,h2)`
- End up with
  `applypair2 :: a -> b -> c -> ((i1,b),(i2,c))`
  - The connection `a = i = i1 = i2` is lost!

Type inference with shallow types

- In `tag :: h -> (i,h)`
  - `h` is local to `tag`
  - `i` is unified with type passed directly to main function
- `h` is called a generic variable
  - Should not make copies of non-generic variables!

Correct type inference rule for $M = \text{let } f = e \text{ in } M'$

- Assume $f :: t$ where $\alpha, \beta, \ldots$ are generic type variables occurring in t
- Make a separate copy of these generic type variables for each instance of f in M'
- Non-generic variables retain their name across all copies of f