Programming Language Concepts: Lecture 20

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 20, 06 April 2009

“Simply typed” λ-calculus

A separate set of variables Var_s for each type s

Define Λ_s, expressions of type s, by mutual recursion

- For each type s, every variable $x \in Var_s$ is in Λ_s
- If $M \in \Lambda_t$ and $x \in Var_s$ then $(\lambda x.M) \in \Lambda_{s \to t}$.
- If $M \in \Lambda_{s \to t}$ and $N \in \Lambda_s$ then $(MN) \in \Lambda_t$.
  - Note that application must be well typed

"Simply typed" λ-calculus

A separate set of variables Var_s for each type s

Define Λ_s, expressions of type s, by mutual recursion

- For each type s, every variable $x \in Var_s$ is in Λ_s
- If $M \in \Lambda_t$ and $x \in Var_s$ then $(\lambda x.M) \in \Lambda_{s \to t}$.
- If $M \in \Lambda_{s \to t}$ and $N \in \Lambda_s$ then $(MN) \in \Lambda_t$.
  - Note that application must be well typed

β rule as usual

- $(\lambda x.M)N \to_\beta M\{x \leftarrow N\}$
- We must have $\lambda x.M \in \Lambda_{s \to t}$ and $N \in \Lambda_s$ for some types s, t
- Moreover, if $\lambda x.M \in \Lambda_{s \to t}$, then $x \in Var_s$, so x and N are compatible

"Simply typed" λ-calculus ...

- Extend $\rightarrow_\beta$ to one-step reduction $\rightarrow$, as usual
- The reduction relation $\rightarrow^*$ is Church-Rosser
- In fact, $\rightarrow^*$ is strongly normalizing
  - M is normalizing : M has a normal form.
  - M is strongly normalizing : every reduction sequence leads to a normal form
- No infinite computations!

Type checking

- Syntax of simply typed λ-calculus permits only well-typed terms
- Converse question; Given an arbitrary term, is it well-typed?

Theorem

The type-checking problem for the simply typed λ-calculus is decidable

Type checking

- Syntax of simply typed λ-calculus permits only well-typed terms
- Converse question; Given an arbitrary term, is it well-typed?

Theorem

The type-checking problem for the simply typed λ-calculus is decidable

- Principal type scheme of a term M — unique type s such that every other valid type is an "instance" of s

Theorem

We can always compute the principal type scheme for any well-typed term in the simply typed λ-calculus.

System F

- Add type variables, a, b, ...
- Use i, j, ... to denote concrete types
- Type schemes

$$s ::= a \mid i \mid s \to s \mid \forall a.s$$

System F

Syntax of second order polymorphic lambda calculus

- Every variable and (type) constant is a term.
- If M is a term, x is a variable and s is a type scheme, then $(\lambda x \in s.M)$ is a term.
- If M and N are terms, so is (MN).
  - Function application does not enforce type check
- If M is a term and a is a type variable, then $(\Lambda a.M)$ is a term.
  - Type abstraction
- If M is a term and s is a type scheme, (Ms) is a term.
  - Type application

System F

Example A polymorphic identity function

$$\Lambda a.\lambda x \in a.x$$

Two β rules, for two types of abstraction

- $(\lambda x \in s.M)N \to_\beta M\{x \leftarrow N\}$
- $(\Lambda a.M)s \to_\beta M\{a \leftarrow s\}$

System F

- System F is also strongly normalizing
- . . . but type inference is undecidable!
  - Given an arbitrary term, can it be assigned a sensible type?

Type inference in System F

Notation

If A is a list of assumptions, $A + \{x : s\}$ is the list where

- Assumption for x in A (if any) is overridden by the new assumption $x : s$.
- For any variable $y \neq x$, assumption does not change

$$\frac{A + \{x : s\} \vdash M : t}{A \vdash (\lambda x \in s.M) : s \to t}$$

$$\frac{A \vdash M : s \to t, \quad A \vdash N : s}{A \vdash (MN) : t}$$

$$\frac{A \vdash M : s}{A \vdash (\Lambda a.M) : \forall a.s}$$

$$\frac{A \vdash M : \forall a.s}{A \vdash Mt : s\{a \leftarrow t\}}$$

Type inference in System F

- Type inference is undecidable for System F
- ...but we have type-checking algorithms for Haskell, ML, ...!
- Haskell etc use a restricted version of polymorphic types
  - All types are universally quantified at the top level
- When we write `map :: (a -> b) -> [a] -> [b]`, we mean that the type is

$$\text{map} \;::\; \forall a, b.\ (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

- Also called shallow typing
- System F permits deep typing

$$\forall a.\ [(\forall b.\ a \rightarrow b) \rightarrow a \rightarrow a]$$

Type inference as equation solving

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

- Generically, `twice :: a -> b -> c`

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)

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`)

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)`)

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)`)

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`

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

Unification

- Start with a system of equations over terms
- Find a substitution for variables that satisfies the equation

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)

Terms

- Fix a set of function symbols and constants : signature
  - Each function symbol as an arity
  - Constants are functions with arity 0

Terms

- Fix a set of function symbols and constants : signature
  - Each function symbol as an arity
  - Constants are functions with arity 0
- Terms are well formed expressions, including variables

Terms

- Fix a set of function symbols and constants : signature
  - Each function symbol as an arity
  - Constants are functions with arity 0
- Terms are well formed expressions, including variables
  - Every variable is a term.

Terms

- Fix a set of function symbols and constants : signature
  - Each function symbol as an arity
  - Constants are functions with arity 0
- Terms are well formed expressions, including variables
  - Every variable is a term.
  - If f is a k-ary function symbol in the signature and t_1, t_2, ..., t_k are terms, then $f(t_1, t_2, \ldots, t_k)$ is a term.

Terms

- Fix a set of function symbols and constants : signature
  - Each function symbol as an arity
  - Constants are functions with arity 0
- Terms are well formed expressions, including variables
  - Every variable is a term.
  - If f is a k-ary function symbol in the signature and t_1, t_2, . . . , t_k are terms, then $f(t_1, t_2, \ldots, t_k)$ is a term.
- Notation
  - $a, b, c, f, \ldots, x, y, \ldots$ are function symbos
  - $A, B, C, F, \ldots, X, Y, \ldots$ are variables

Unification

Example

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

Unification

Example

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

- Substitution: assigns a term to each variable X, Y, Z

Unification

Example

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

- Substitution: assigns a term to each variable X, Y, Z
- Unifier: substitution that satisfies equations

Unification

Example

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

- Substitution: assigns a term to each variable X, Y, Z
- Unifier: substitution that satisfies equations
- For instance, $\{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\}$

Unification

Example

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

- Substitution: assigns a term to each variable X, Y, Z
- Unifier: substitution that satisfies equations
- For instance, $\{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\} = \theta$

Unification

Example

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

- Substitution: assigns a term to each variable X, Y, Z
- Unifier: substitution that satisfies equations
- For instance, $\{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\} = \theta$
- $t\theta$: apply substitution θ to term t

Unification

Example

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

- Substitution: assigns a term to each variable X, Y, Z
- Unifier: substitution that satisfies equations
- For instance, $\{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\} = \theta$
- $t\theta$: apply substitution θ to term t (not $\theta(t)$!)

Unification

Example

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

- Substitution: assigns a term to each variable X, Y, Z
- Unifier: substitution that satisfies equations
- For instance, $\{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\} = \theta$
- $t\theta$: apply substitution θ to term t (not $\theta(t)$!)
- Apply substitution in parallel
  - $t = g(p(X), q(f(Y)))$

Unification

Example

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

- Substitution: assigns a term to each variable X, Y, Z
- Unifier: substitution that satisfies equations
- For instance, $\{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\} = \theta$
- $t\theta$: apply substitution θ to term t (not $\theta(t)$!)
- Apply substitution in parallel
  - $t = g(p(X), q(f(Y)))$
  - $\gamma = \{X \leftarrow Y, Y \leftarrow f(a)\}$

Unification

Example

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

- Substitution: assigns a term to each variable X, Y, Z
- Unifier: substitution that satisfies equations
- For instance, $\{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\} = \theta$
- $t\theta$: apply substitution θ to term t (not $\theta(t)$!)
- Apply substitution in parallel
  - $t = g(p(X), q(f(Y)))$
  - $\gamma = \{X \leftarrow Y, Y \leftarrow f(a)\}$
  - $t\gamma = g(p(Y), q(f(f(a))))$

Unification

Example

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

- Substitution: assigns a term to each variable X, Y, Z
- Unifier: substitution that satisfies equations
- For instance, $\{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\} = \theta$
- $t\theta$: apply substitution θ to term t (not $\theta(t)$!)
- Apply substitution in parallel
  - $t = g(p(X), q(f(Y)))$
  - $\gamma = \{X \leftarrow Y, Y \leftarrow f(a)\}$
  - $t\gamma = g(p(Y), q(f(f(a))))$
  - $g(p(Y))$ does not become $g(p(f(a)))$!

Unification

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

- Many solutions are possible:
  - $\theta = \{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\}$
  - $\theta' = \{X \leftarrow f(a), Y \leftarrow a, Z \leftarrow a\}$
  - $\theta'' = \{X \leftarrow f(a), Y \leftarrow Z\}$

Unification

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

- Many solutions are possible:
  - $\theta = \{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\}$
  - $\theta' = \{X \leftarrow f(a), Y \leftarrow a, Z \leftarrow a\}$
  - $\theta'' = \{X \leftarrow f(a), Y \leftarrow Z\}$
- θ'' is the "least constrained"

Unification

$$f(X) = f(f(a))$$
$$g(Y) = g(Z)$$

- Many solutions are possible:
  - $\theta = \{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\}$
  - $\theta' = \{X \leftarrow f(a), Y \leftarrow a, Z \leftarrow a\}$
  - $\theta'' = \{X \leftarrow f(a), Y \leftarrow Z\}$
- θ'' is the "least constrained"
- Any solution γ breaks up into two steps, first of which is θ''
  - θ is θ'' followed by $\{Y \leftarrow g(a)\}$

Unification

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

- Many solutions are possible:
  - $\theta = \{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\}$
  - $\theta' = \{X \leftarrow f(a), Y \leftarrow a, Z \leftarrow a\}$
  - $\theta'' = \{X \leftarrow f(a), Y \leftarrow Z\}$
- θ'' is the "least constrained"
- Any solution γ breaks up into two steps, first of which is θ''
  - θ is θ'' followed by $\{Y \leftarrow g(a)\}$
- Least constrained solution: most general unifier

Unification

Obstacles to unification

Unification

Obstacles to unification

- Equations of the form $p(\ldots) = q(\ldots)$
  - Outermost function symbols don't agree
  - No substitution can make the terms equal

Unification

Obstacles to unification

- Equations of the form $p(\ldots) = q(\ldots)$
  - Outermost function symbols don't agree
  - No substitution can make the terms equal
- Equations of the form $X = f(\ldots X \ldots)$
  - Any substitution for X also applies to X nested in f

Unification

Obstacles to unification

- Equations of the form $p(\ldots) = q(\ldots)$
  - Outermost function symbols don't agree
  - No substitution can make the terms equal
- Equations of the form $X = f(\ldots X \ldots)$
  - Any substitution for X also applies to X nested in f
- These are the only two reasons why unification can fail!

A unification algorithm

- Start with equations

$$\begin{array}{rcl} t_1^l & = & t_1^r \\ t_2^l & = & t_2^r \\ & \vdots & \\ t_n^l & = & t_n^r \end{array}$$

- Perform a sequence of transformations on these equations till no more transformations apply

Unification algorithm : transformations

1. $t = X$, t is not a variable $\leadsto$ $X = t$.

Unification algorithm : transformations

1. $t = X$, t is not a variable $\leadsto$ $X = t$.
2. Erase equations of form $X = X$.

Unification algorithm : transformations

1. $t = X$, t is not a variable $\rightsquigarrow$ $X = t$.
2. Erase equations of form $X = X$.
3. Let $t = t'$ where $t = f(\ldots)$, $t' = f'(\ldots)$

Unification algorithm : transformations

1. $t = X$, t is not a variable $\leadsto$ $X = t$.
2. Erase equations of form $X = X$.
3. Let $t = t'$ where $t = f(\ldots)$, $t' = f'(\ldots)$
   - $f \neq f'$ $\leadsto$ terminate : unification not possible

Unification algorithm : transformations

1. $t = X$, t is not a variable $\leadsto X = t$.

2. Erase equations of form $X = X$.

3. Let $t = t'$ where $t = f(\ldots)$, $t' = f'(\ldots)$
   - $f \neq f' \leadsto$ 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

     $$t_1 = t'_1, t_2 = t'_2, \ldots, t_k = t'_k$$

Unification algorithm : transformations

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

Unification algorithm : transformations

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

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

Unification algorithm : Examples

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

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

Unification algorithm : Examples

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

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

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

Unification algorithm : Examples

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

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

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

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

Unification algorithm : Examples . . .

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

Unification algorithm : Examples . . .

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

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

Unification algorithm : Examples . . .

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

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

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

Unification algorithm : Examples . . .

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

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

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

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

Unification algorithm : Examples . . .

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

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

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

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

Unification algorithm : Examples . . .

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

Unification algorithm : Examples . . .

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

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

Unification algorithm : Examples . . .

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

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

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

Unification algorithm : Examples . . .

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

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

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

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

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 : 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

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

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

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

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 $i, j, k, \ldots$

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., $i = \texttt{Char}$, $C_i = \{\texttt{'a'}, \texttt{'b'}, \ldots\}$

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

- $M = c \in C_i \rightsquigarrow M :: i$

Type inference with shallow types

- $M = c \in C_i \rightsquigarrow M :: i$
- $M = x \rightsquigarrow M :: \alpha$ for a fresh type variable α

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.

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'

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 $\alpha = \gamma$

Type inference with shallow types

- $M = c \in C_i \rightsquigarrow M :: i$
- $M = x \rightsquigarrow M :: \alpha$ for a fresh type variable α
- $M = \lambda x.M' \rightsquigarrow M :: \alpha \rightarrow \beta$ 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.

Type inference with shallow types

- $M = c \in C_i \rightsquigarrow M :: i$
- $M = x \rightsquigarrow M :: \alpha$ for a fresh type variable α
- $M = \lambda x.M' \rightsquigarrow M :: \alpha \rightarrow \beta$ 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, $M' :: \alpha \rightarrow \beta$, $N' :: \gamma$

Type inference with shallow types

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

Type inference with shallow types

Consider

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

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')
```

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')
```

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

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

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

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

Suppose, we write, instead

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

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

Suppose, we write, instead

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

Now, we have

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

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`

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

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`

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

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

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

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)