Design Principles of Programming Languages

Type Reconstruction

Zhenjiang Hu, Haiyan Zhao, Yingfei Xiong

Peking University, Spring Term, 2014

Type Reconstruction

- A controversial feature
  - Pros: less typing, yeah!
    - Map<String, List<Pair<Token, SrcInfo>>>
  - Cons: more difficult to debug type errors
    - No type declaration as central concept

- Yet worth studying
  - Understanding the potentials of compilers
  - Closely related to polymorphism

An Example

- $g = \lambda a. \lambda f. iszero\ (f\ a)$
- $h = g\ 10$

- What is the types of a, f, g, h?

Introducing Type Variables

- $g = \lambda a{:}X.\lambda f{:}Y.iszero\,(f\ a)$
- $h = g\ 10$

Generating Constraints

- $g = \lambda a{:}X.\lambda f{:}Y.iszero\ (f\ a)$
- $h = g\ 10$

- $Y = X \rightarrow Z_0$
- $Nat = Z_0$
- $X \rightarrow Y \rightarrow Bool = Nat \rightarrow Z_1$

Unification

- $g = \lambda a{:}X.\lambda f{:}Y.\,iszero\,(f\ a)$
- $h = g\ 10$

- $Y = X \rightarrow Z_0$
- $Nat = Z_0$
- $X \rightarrow Y \rightarrow Bool = Nat \rightarrow Z_1$

- $X = Nat, Y = Nat \rightarrow Nat, Z_0 = Nat, Z_1 = (Nat \rightarrow Nat) \rightarrow Bool$

By typing rules

- $g = \lambda a{:}X.\lambda f{:}Y.iszero\ (f\ a)$
- $h = g\ 10$

- $Y = X \rightarrow Z_0$
- $Nat = Z_0$
- $X \rightarrow Y \rightarrow Bool = Nat \rightarrow Z_1$

- $X = Nat, Y = Nat \rightarrow Nat, Z_0 = Nat, Z_1 = (Nat \rightarrow Nat) \rightarrow Bool$
- $g{:}Nat \rightarrow (Nat \rightarrow Nat) \rightarrow Bool$
- $h{:}(Nat \rightarrow Nat) \rightarrow Bool$

Type Variables

- New Syntactic Rule

```
t ::= …
        λx. t            // untyped lambda abstraction
T ::= …
        X                // type variables
```

Type Substitution

- A finite mapping from type variables to types
  - $\sigma = [X \mapsto Bool, Y \mapsto Nat \to Nat]$
  - Note the difference between $\mapsto$ and $\to$
- Application of substituation σ

$$
\begin{array}{lcl}
\sigma(\mathtt{X}) & = & \begin{cases} \mathtt{T} & \text{if } (\mathtt{X} \mapsto \mathtt{T}) \in \sigma \\ \mathtt{X} & \text{if X is not in the domain of } \sigma \end{cases} \\
\sigma(\mathtt{Nat}) & = & \mathtt{Nat} \\
\sigma(\mathtt{Bool}) & = & \mathtt{Bool} \\
\sigma(\mathtt{T_1 \to T_2}) & = & \sigma\mathtt{T_1} \to \sigma\mathtt{T_2}
\end{array}
$$

Preservation of Typing under Type Substitution

$$\frac{\sigma : \text{any type substitution} \qquad \Gamma \vdash t : T}{\sigma\Gamma \vdash \sigma t : T}$$

- Proof: By induction on typing rules

Solution

- A solution for (Γ, t) is a pair (σ, T) such that $\sigma\Gamma \vdash \sigma t : T$

EXAMPLE: Let $\Gamma = \texttt{f}{:}X, \texttt{a}{:}Y$ and $\texttt{t} = \texttt{f a}$. Then

$$
\begin{array}{ll}
([X \mapsto Y\to\text{Nat}],\ \text{Nat}) & ([X \mapsto Y\to Z],\ Z) \\
([X \mapsto Y\to Z,\ Z \mapsto \text{Nat}],\ Z) & ([X \mapsto Y\to\text{Nat}\to\text{Nat}],\ \text{Nat}\to\text{Nat}) \\
([X \mapsto \text{Nat}\to\text{Nat},\ Y \mapsto \text{Nat}],\ \text{Nat}\to\text{Nat}) &
\end{array}
$$

are all solutions for $(\Gamma, \texttt{t})$.

- Problem: which one is better?

Principal Types

- ~~Substitution σ is more general than σ', written $\sigma \sqsubseteq \sigma'$ iff $\sigma' = \gamma \circ \sigma$ for some γ.~~
- Substitution σ is more general than σ' for term t, written $\sigma \sqsubseteq_t \sigma'$ iff $\sigma' t = \gamma(\sigma t)$ for some γ.
- A most general substitution leads to a principle type

EXAMPLE: Let Γ = `f`:X, `a`:Y and `t` = `f a`. Then

$$\begin{array}{ll}
([\mathsf{X} \mapsto \mathsf{Y}\to\mathsf{Nat}],\ \mathsf{Nat}) & ([\mathsf{X} \mapsto \mathsf{Y}\to\mathsf{Z}],\ \mathsf{Z}) \\
([\mathsf{X} \mapsto \mathsf{Y}\to\mathsf{Z},\ \mathsf{Z} \mapsto \mathsf{Nat}],\ \mathsf{Z}) & ([\mathsf{X} \mapsto \mathsf{Y}\to\mathsf{Nat}\to\mathsf{Nat}],\ \mathsf{Nat}\to\mathsf{Nat}) \\
([\mathsf{X} \mapsto \mathsf{Nat}\to\mathsf{Nat},\ \mathsf{Y} \mapsto \mathsf{Nat}],\ \mathsf{Nat}\to\mathsf{Nat}) &
\end{array}$$

are all solutions for $(\Gamma, \mathtt{t})$.

- Which are most general substituions?

Principal Types

- Substitution σ is more general than σ' for term t, written $\sigma \sqsubseteq_t \sigma'$ iff $\sigma' t = \gamma(\sigma t)$ for some γ.
- A most general substitution leads to a principle type

EXAMPLE: Let Γ = `f`:X, `a`:Y and `t` = `f a`. Then

$$\begin{array}{ll}
([X \mapsto Y{\to}Nat],\ Nat) & ([X \mapsto Y{\to}Z],\ Z) \\
([X \mapsto Y{\to}Z,\ Z \mapsto Nat],\ Z) & ([X \mapsto Y{\to}Nat{\to}Nat],\ Nat{\to}Nat) \\
([X \mapsto Nat{\to}Nat,\ Y \mapsto Nat],\ Nat{\to}Nat) &
\end{array}$$

are all solutions for (Γ, `t`).

- Which one is a most general one?
  1. Replacing less variables
  2. Replacing with less specific types

Constraint Set

- A constraint set is a set of equations $\{S_i = T_i\}$.

- σ satisfy C=$\{S_i = T_i\}$ when $\sigma S_i = \sigma T_i$ for all i.

Constraint Typing Rules

- $\dfrac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T \mid \{\}}$
- $\dfrac{\Gamma, x{:}T_1 \vdash t_2{:}T_2 \mid C}{\Gamma \vdash \lambda x{:}T_1.t_2 {:} T_1 \rightarrow T_2 \mid C}$
- $\dfrac{\begin{array}{c} X \text{ is a fresh type variable} \\ \Gamma, x{:}X \vdash t{:}T \mid C \end{array}}{\Gamma \vdash \lambda x.t {:} X \rightarrow T \mid C}$
- $\dfrac{\begin{array}{c} \Gamma \vdash t_1{:}T_1 \mid C_1 \qquad \Gamma \vdash t_2{:}T_2 \mid C_2 \\ X \text{ is a fresh type variable} \end{array}}{\Gamma \vdash t_1\ t_2 {:} X \mid C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow X\}}$
- $\dfrac{}{\Gamma \vdash 0{:}Nat \mid \{\}}$
- $\dfrac{}{\Gamma \vdash true{:}Bool \mid \{\}}$
- $\dfrac{}{\Gamma \vdash false{:}Bool \mid \{\}}$
- $\dfrac{\Gamma \vdash t_1{:}T \mid C}{\Gamma \vdash succ\ t_1{:}Nat \mid C \cup \{T = Nat\}}$
- $\dfrac{\Gamma \vdash t_1{:}T \mid C}{\Gamma \vdash pred\ \ t_1{:}Nat \mid C \cup \{T = Nat\}}$
- $\dfrac{\Gamma \vdash t_1{:}T \mid C}{\Gamma \vdash iszero\ t_1{:}Nat \mid C \cup \{T = Nat\}}$
- $\dfrac{\begin{array}{c} \Gamma \vdash t_1{:}T_1 \mid C_1 \quad \Gamma \vdash t_2{:}T_2 \mid C_2 \quad \Gamma \vdash t_3{:}T_3 \mid C_3 \\ C' = C_1 \cup C_2 \cup C_3 \cup \{T_1 = Bool, T_2 = T_3\} \end{array}}{\Gamma \vdash if\ t_1\ then\ t_2\ else\ t_3 {:} T_2 \mid C'}$

* In the book, the freshness of type variables are also treated formally in the rules

Exercise

- Deduce constraints for
  - $\lambda f.\lambda n.if\ iszero\ n\ then\ f\ n\ else\ n$

Soundness and Completeness

- Suppose that $\Gamma \vdash t : S \mid C$. A solution for (Γ, t, S, C) is a pair (σ, T) such that σ satisfies C and $\sigma S = T$.
- Soundness
  - If (σ, T) is a solution for (Γ, t, S, C), then it is also a solution for (Γ, t).
  - Proof: by induction on constraint typing rules
- Completeness
  - If (σ, T) is a solution for (Γ, t), then there exists solution (σ', T) for (Γ, t, S, C) where σ and σ' are the same for any type variables in t.
  - Proof: by induction on constraint typing rules

Unification Algorithm

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

Soundness

- The unification algorithm returns a most general substitution if there is one, or fails otherwise.
  - Proof: Induction on the number of recursive calls

Termination

- Every iteration either
  - drop a constraint from C, or
  - divide a constraint into smaller constraints

Type Reconstruction with Subtyping

- Constraints containing both <: and =
- Every type variable starts with TOP
- Shrink types to satisfy constraints
  - X:Nat, Y:TOP $\overset{X=Y}{\Longrightarrow}$X:Nat, Y:Nat
  - X:Nat, Y:TOP $\overset{X<:Y}{\Longrightarrow}$X:Nat, Y:TOP
  - X:Nat, Y:TOP $\overset{X:>Y}{\Longrightarrow}$X:Nat, Y:Nat
- Until a fixed point is reached
- Termination
  - Types for the variables are always shrink
  - Lower bound exist

Polymorphism

- let double=λf. λa. f (f a) in
  {
    double (λx:Nat. succ (succ x)) 1,
    double (λx:Bool x) false
  }

- Will this program be type checked?

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x{=}t_1\ \texttt{in}\ t_2 : T_2}$$

Three types of polymorphisms

- Polymorphism
  - A single interface to different types
- Adhoc Polymorphism
  - e.g., case…of…, function overloading

```
double f:Nat->Nat a:Nat = f (f a)
double f:Bool->Bool a:Bool = f (f a)
```

- Subtyping

```
interface function {
    Object apply(Object);
    Object doubleApply(Object);
}
```

- Parametric Polymorphism
  - e.g., C++ template

Hindley-Milner Type System

- A simple polymorphism type system deals with the previous case
- Widely-used in some mainstream functional programming languages
    - Ocaml, ML, Haskell98.
- Weaker than System-F to be introduced in the next course
    - Type reconstruction is undecidable for System-F.

Typing Rules in HM-System

$$\cancel{\dfrac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let\ x{=}t}_1\ \mathtt{in\ t}_2 : \mathtt{T}_2}}$$

$$\dfrac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2 \ \ |_{\mathcal{X}}\ C}{\Gamma \vdash \mathtt{let\ x{=}t}_1\ \mathtt{in\ t}_2 : \mathtt{T}_2 \ \ |_{\mathcal{X}}\ C}$$

Exercise

- What is the type of this program?
- let f = λx.x in
  let g = λx.f (f x) in
  {g 5, g true}

Exercise

- What is the type of this program?

-
  ```
  (λf.
   let g = λx.f (f x) in
   {g 5, g true}
  ) (λx.x)
  ```

Exercise

- What is the type of this program?
- let h= λx.x in
  (λf.
   let g = λx.f (f x) in
   {g 5, g true}
) h

Inefficiency of the typing rules

- let double=λf. λa. f (f a) in
  {
    double (λx:Nat. succ (succ x)) 1,
    double (λx:Bool x) false
  }

- The red part is type checked twice

A more efficient algorithm

- let double=λf. λa. f (f a) in
  {
    double (λx:Nat. succ (succ x)) 1,
    double (λx:Bool x) false
  }

1. Type check only the "let" part (red) and get its principle type
   - $(X\to X)\to X \to X$

A more efficient algorithm

- let double=λf. λa. f (f a) in
  {
   double (λx:Nat. succ (succ x)) 1,
   double (λx:Bool x) false
  }

1. Type check only the “let” part and get its principle type
   - $(X\to X)\to X \to X$
2. Introduce quantification for type variables not used in Γ
   - double: $\forall X.(X\to X)\to X \to X$

A more efficient algorithm

- let double=λf. λa. f (f a) in
  {
  double (λx:Nat. succ (succ x)) 1,
  double (λx:Bool x) false
  }

1. Type check only the "let" part and get its principle type
   - (X→X)→X →X
2. Introduce quantification for type variables not used in Γ
   - double: ∀X.(X→X)→X →X
3. Add it to Γ and type check the body, using an additional typing rule

$$\frac{t : \forall X_1 \ldots X_n . T \in \Gamma \quad Y_1 \ldots Y_n \text{ are fresh variables}}{\Gamma \vdash t : [X_1 \mapsto Y_1] \ldots [X_n \mapsto Y_n] T}$$

A more efficient algorithm

- The informal description does not work with the formal system in the text book
  - Need to reformulate all rules to make it formal
- For full formal description, see Wikipedia page of “Hindley-Milner type system”

Ref variables

- What is the type of the following program?
  - let r=ref (λx.x) in
    (r:=(λx:Nat, succ x); (!r)true);

Ref variables

- What is the type of the following program?
  - let r=ref (λx.x) in
    (r:=(λx:Nat, succ x); (!r)true);

- HM-system does not work with ref variables
- Disallow polymorphism when the let definition is of reference type

Homework

- Change the constraint typing rule and the unification algorithm so that the following term can be typed
  - fix (λh. λx:Nat. h)