Design Principles of Programming Languages

Type Reconstruction

Zhenjiang Hu, Haiyan Zhao, Yingfei Xiong

Peking University, Spring Term, 2016

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

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

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

Unification

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

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

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

By typing rules

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

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

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

Type Variables

- New Syntactic Rule

  t ::= …

  $\lambda x. t$ // 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

$$
\begin{aligned}
\sigma(\mathsf{X}) &= \begin{cases} \mathsf{T} & \text{if } (\mathsf{X} \mapsto \mathsf{T}) \in \sigma \\ \mathsf{X} & \text{if X is not in the domain of } \sigma \end{cases} \\
\sigma(\mathsf{Nat}) &= \mathsf{Nat} \\
\sigma(\mathsf{Bool}) &= \mathsf{Bool} \\
\sigma(\mathsf{T}_1 \to \mathsf{T}_2) &= \sigma\mathsf{T}_1 \to \sigma\mathsf{T}_2
\end{aligned}
$$

Composition of Type Substitution

If σ and γ are substitutions, we write $\sigma \circ \gamma$ for the substitution formed by composing them as follows:

$$\sigma \circ \gamma = \left[\begin{array}{ll} \mathsf{X} \mapsto \sigma(\mathsf{T}) & \text{for each } (\mathsf{X} \mapsto \mathsf{T}) \in \gamma \\ \mathsf{X} \mapsto \mathsf{T} & \text{for each } (\mathsf{X} \mapsto \mathsf{T}) \in \sigma \text{ with } \mathsf{X} \notin \mathit{dom}(\gamma) \end{array} \right]$$

Note that $(\sigma \circ \gamma)\mathsf{S} = \sigma(\gamma\mathsf{S})$. □

Preservation of Typing under Type Substitution

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

- 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 $\Gamma = \mathtt{f}{:}X, \mathtt{a}{:}Y$ and $\mathtt{t} = \mathtt{f}\ \mathtt{a}$. Then

$$([X \mapsto Y{\to}\mathtt{Nat}],\ \mathtt{Nat}) \qquad ([X \mapsto Y{\to}Z],\ Z)$$
$$([X \mapsto Y{\to}Z,\ Z \mapsto \mathtt{Nat}],\ Z) \qquad ([X \mapsto Y{\to}\mathtt{Nat}{\to}\mathtt{Nat}],\ \mathtt{Nat}{\to}\mathtt{Nat})$$
$$([X \mapsto \mathtt{Nat}{\to}\mathtt{Nat},\ Y \mapsto \mathtt{Nat}],\ \mathtt{Nat})$$

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

- 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.
- A most general substitution leads to a principle type

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

$$([X \mapsto Y \to Nat],\ Nat) \qquad ([X \mapsto Y \to Z],\ Z)$$
$$([X \mapsto Y \to Z,\ Z \mapsto Nat],\ Z) \qquad ([X \mapsto Y \to Nat \to Nat],\ Nat \to Nat)$$
$$([X \mapsto Nat \to Nat,\ Y \mapsto Nat],\ Nat)$$

are all solutions for (Γ, t).

- Which are most general substituions?

Principal Types

- Substitution σ is more general than σ', written $\sigma \sqsubseteq \sigma'$ iff $\sigma' = \gamma \circ \sigma$ for some γ.
- A most general substitution leads to a principle type

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

$$([X \mapsto Y\to Nat],\ Nat) \qquad ([X \mapsto Y\to Z],\ Z)$$
$$([X \mapsto Y\to Z,\ Z \mapsto Nat],\ Z) \qquad ([X \mapsto Y\to Nat\to Nat],\ Nat\to Nat)$$
$$([X \mapsto Nat\to Nat,\ Y \mapsto Nat],\ Nat)$$

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

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

* 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

$$
\begin{aligned}
\mathit{unify}(C) \quad = \quad & \text{if } C = \emptyset, \text{ then } [\,] \\
& \text{else let } \{\mathsf{S} = \mathsf{T}\} \cup C' = C \text{ in} \\
& \quad \text{if } \mathsf{S} \text{ is } \mathsf{T} \\
& \qquad \text{then } \mathit{unify}(C') \\
& \quad \text{else if } \mathsf{S} \text{ is } \mathsf{X} \text{ and } \mathsf{X} \notin \mathit{FV}(\mathsf{T}) \\
& \qquad \text{then } \mathit{unify}([\mathsf{X} \mapsto \mathsf{T}]C') \circ [\mathsf{X} \mapsto \mathsf{T}] \\
& \quad \text{else if } \mathsf{T} \text{ is } \mathsf{X} \text{ and } \mathsf{X} \notin \mathit{FV}(\mathsf{S}) \\
& \qquad \text{then } \mathit{unify}([\mathsf{X} \mapsto \mathsf{S}]C') \circ [\mathsf{X} \mapsto \mathsf{S}] \\
& \quad \text{else if } \mathsf{S} \text{ is } \mathsf{S}_1 \to \mathsf{S}_2 \text{ and } \mathsf{T} \text{ is } \mathsf{T}_1 \to \mathsf{T}_2 \\
& \qquad \text{then } \mathit{unify}(C' \cup \{\mathsf{S}_1 = \mathsf{T}_1, \mathsf{S}_2 = \mathsf{T}_2\}) \\
& \quad \text{else} \\
& \qquad \mathit{fail}
\end{aligned}
$$

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

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

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \ \mid_{\chi}\ C}{\Gamma \vdash \texttt{let } x{=}t_1 \texttt{ in } t_2 : T_2 \ \mid_{\chi}\ C}$$

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→X)→X →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
   - $(X \to X) \to X \to X$
2. Introduce quantification for type variables not used in Γ
   - double: $\forall X.(X \to X) \to X \to X$
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