

# Type Reconstruction and Polymorphism

Week 9

based on slides by Martin Odersky

1

## Type Checking and Type Reconstruction

We now come to the question of type checking and type reconstruction.

**Type checking:** Given  $\Gamma$ ,  $t$  and  $T$ , check whether  $\Gamma \vdash t : T$

**Type reconstruction:** Given  $\Gamma$  and  $t$ , find a type  $T$  such that  $\Gamma \vdash t : T$

Type checking and reconstruction seem difficult since parameters in lambda calculus do not carry their types with them.

Type reconstruction also suffers from the problem that a term can have many types.

**Idea:** : We construct all type derivations in parallel, reducing type reconstruction to a unification problem.

2

## From Judgements to Equations

$TP : \text{Judgement} \rightarrow \text{Equations}$

$TP(\Gamma \vdash t : T) =$

**case  $t$  of**

$x$  :  $\{\Gamma(x) \doteq T\}$

$\lambda x.t'$  : **let**  $a, b$  **fresh in**

$\{(a \rightarrow b) \doteq T\} \cup$

$TP(\Gamma, x : a \vdash t' : b)$

$t \ t'$  : **let**  $a$  **fresh in**

$TP(\Gamma \vdash t : a \rightarrow T) \cup$

$TP(\Gamma \vdash t' : a)$

3

## Example

Let  $\text{twice} = \lambda f.\lambda x.f(f(x))$ .

Then  $\text{twice}$  gives rise to the following equations (see blackboard).

4

## Soundness and Completeness I

**Definition:** In general, a type reconstruction algorithm  $\mathcal{A}$  assigns to an environment  $\Gamma$  and a term  $t$  a set of types  $\mathcal{A}(\Gamma, t)$ .

The algorithm is **sound** if for every type  $T \in \mathcal{A}(\Gamma, t)$  we can prove the judgement  $\Gamma \vdash t : T$ .

The algorithm is **complete** if for every provable judgement  $\Gamma \vdash t : T$  we have that  $T \in \mathcal{A}(\Gamma, t)$ .

5

**Theorem:**  $TP$  is sound and complete. Specifically:

$$\Gamma \vdash t : T \quad \text{iff} \quad \exists \bar{b}. [T/a]EQNS$$

where

$a$  is a new type variable

$$EQNS = TP(\Gamma \vdash t : a)$$

$$\bar{b} = tv(EQNS) \setminus tv(\Gamma)$$

Here,  $tv$  denotes the set of free type variables (of a term, and environment, an equation set).

6

## Type Reconstruction and Unification

**Problem:** : Transform set of equations

$$\{T_i \hat{=} U_i\}_{i=1, \dots, m}$$

into equivalent substitution

$$\{a_j \mapsto T_j'\}_{j=1, \dots, n}$$

where type variables do not appear recursively on their right hand sides (directly or indirectly). That is:

$$a_j \notin tv(T_k') \quad \text{for } j = 1, \dots, n, k = j, \dots, n$$

7

## Substitutions

A **substitution**  $s$  is an idempotent mapping from type variables to types which maps all but a finite number of type variables to themselves.

We often represent a substitution  $s$  as a set of equations  $a \hat{=} T$  with  $a$  not in  $tv(T)$ .

Substitutions can be generalized to mappings from types to types by defining

$$s(T \rightarrow U) = sT \rightarrow sU$$

Substitutions are idempotent mappings from types to types, i.e.

$$s(s(T)) = s(T). \quad (\text{why?})$$

The  $\circ$  operator denotes composition of substitutions (or other functions):  $(f \circ g) x = f(gx)$ .

8

## A Unification Algorithm

We present an incremental version of Robinson's algorithm (1965).

$$\begin{aligned}
 mgu & : (Type \doteq Type) \rightarrow Subst \rightarrow Subst \\
 mgu(T \doteq U) s & = mgu'(sT \doteq sU) s \\
 mgu'(a \doteq a) s & = s \\
 mgu'(a \doteq T) s & = s \cup \{a \mapsto T\} \quad \text{if } a \notin tv(T) \\
 mgu'(T \doteq a) s & = s \cup \{a \mapsto T\} \quad \text{if } a \notin tv(T) \\
 mgu'(T \rightarrow T' \doteq U \rightarrow U') s & = (mgu(T' \doteq U') \circ mgu(T \doteq U)) s \\
 mgu'(T \doteq U) s & = error \quad \text{in all other cases}
 \end{aligned}$$

9

## Soundness and Completeness of Unification

**Definition:** A substitution  $u$  is a **unifier** of a set of equations  $\{T_i \doteq U_i\}_{i=1, \dots, m}$  if  $uT_i = uU_i$ , for all  $i$ . It is a **most general unifier** if for every other unifier  $u'$  of the same equations there exists a substitution  $s$  such that  $u' = s \circ u$ .

**Theorem:** Given a set of equations  $EQNS$ . If  $EQNS$  has a unifier then  $mgu\ EQNS\ \{\}$  computes the most general unifier of  $EQNS$ . If  $EQNS$  has no unifier then  $mgu\ EQNS\ \{\}$  fails.

10

## From Judgements to Substitutions

$TP : Judgement \rightarrow Subst \rightarrow Subst$

$TP(\Gamma \vdash t : T) =$

**case  $t$  of**

$x$  :  $mgu(\Gamma(x) \doteq T)$

$\lambda x.t'$  : **let**  $a, b$  **fresh in**

$mgu((a \rightarrow b) \doteq T)$   $\circ$

$TP(\Gamma, x : a \vdash t' : b)$

$t\ t'$  : **let**  $a$  **fresh in**

$TP(\Gamma \vdash t : a \rightarrow T)$   $\circ$

$TP(\Gamma \vdash t' : a)$

11

## Soundness and Completeness II

One can show by comparison with the previous algorithm:

**Theorem:**  $TP$  is sound and complete. Specifically:

$\Gamma \vdash t : T$  iff  $T = r(s(a))$

**where**

$a$  is a new type variable

$s = TP(\Gamma \vdash t : a)\ \{\}$

$r$  is a substitution on  $tv(s\ a) \setminus tv(s\ \Gamma)$

12

## Polymorphism

In the simply typed lambda calculus, a term can have many types. But a variable or parameter has only one type.

Example:

$$(\lambda x.xx)(\lambda y.y)$$

is untypable. But if we substitute actual parameter for formal, we obtain

$$(\lambda y.y)(\lambda y.y) : a \rightarrow a$$

Functions which can be applied to arguments of many types are called **polymorphic**.

13

## Polymorphism in Programming

Polymorphism is essential for many program patterns.

Example: map

```
def map f xs =  
  if (isEmpty xs) nil  
  else cons (f (head xs)) (map (f, tail xs))  
...  
names: List[String]  
nums : List[Int]  
...  
map toUpperCase names  
map increment nums
```

Without a polymorphic type for map one of the last two lines is always illegal!

14

## Forms of Polymorphism

Polymorphism means “having many forms”.

Polymorphism also comes in several forms.

- **Universal polymorphism**, sometimes also called **generic types**: The ability to instantiate type variables.
- **Inclusion polymorphism**, sometimes also called **subtyping**: The ability to treat a value of a subtype as a value of one of its supertypes.
- **Ad-hoc polymorphism**, sometimes also called **overloading**: The ability to define several versions of the same function name, with different types.

We first concentrate on universal polymorphism.

Two basic approaches: **explicit** or **implicit**.

15

## Explicit Polymorphism

We introduce a polymorphic type  $\forall a.T$ , which can be used just as any other type.

We then need to make introduction and elimination of  $\forall$ 's explicit.

Typing rules:

$$(\forall E) \frac{\Gamma \vdash t : \forall a.T}{\Gamma \vdash t[U] : [U/a]T} \quad (\forall I) \frac{\Gamma, a \vdash t : T}{\Gamma \vdash \Lambda a.t : \forall a.T}$$

16

We also need to give all parameter types, so programs become verbose.

**Example:**

```
def map [a][b] (f: a => b) (xs: List[a]) =
  if (isEmpty [a] (xs)) nil [b]
  else cons [b] (f (head [a] xs)) (map [a][b] (f) (tail [a] xs))
...
names: List[String]
nums : List[Int]
...
map [String] [String] toUpperCase names
map [Int] [Int] increment nums
```

17

## Translating to System F

The translation of map into a System-F term is as follows: (See blackboard)

18

## Implicit Polymorphism

Implicit polymorphism does not require annotations for parameter types or type instantiations.

**Idea:** In addition to types (as in simply typed lambda calculus), we have a new syntactic category of **type schemes**. Syntax:

**Type Scheme**  $S ::= T \mid \forall a. S$

Type schemes are not fully general types; they are used only to type named values, introduced by a `val` construct.

The resulting type system is called the **Hindley/Milner system**, after its inventors. (The original treatment uses `let ... in ...` rather than `val ... ; ...`).

19

## Hindley/Milner Typing rules

(VAR)  $\Gamma, x : S, \Gamma' \vdash x : S \quad (x \notin \text{dom}(\Gamma'))$

( $\forall$ E)  $\frac{\Gamma \vdash t : \forall a. T}{\Gamma \vdash t : [U/a]T} \quad (\forall I) \frac{\Gamma, a \vdash t : T \quad a \notin \text{fv}(\Gamma)}{\Gamma \vdash t : \forall a. T}$

(LET)  $\frac{\Gamma \vdash t : S \quad \Gamma, x : S \vdash t' : T}{\Gamma \vdash \text{let } x = t \text{ in } t' : T}$

The other two rules are as in simply typed lambda calculus:

( $\rightarrow$ I)  $\frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash \lambda x. t : T \rightarrow U} \quad (\rightarrow$ E)  $\frac{\Gamma \vdash M : T \rightarrow U \quad \Gamma \vdash N : T}{\Gamma \vdash M N : U}$

20

## Type Reconstruction for Hindley/Milner

Type reconstruction for the Hindley/Milner system works as for simply typed lambda calculus. We only have to add a clause for **let** expressions and refine the rules for variables.

21

$TP : \text{Judgement} \rightarrow \text{Subst} \rightarrow \text{Subst}$

$TP(\Gamma \vdash t : T) s =$

**case  $t$  of**

...

**let  $x = t_1$  in  $t_2$  : let  $a, b$  fresh in**

**let  $s_1 = TP(\Gamma \vdash t_1 : a)$  in**

**$TP(\Gamma, x : \text{gen}(s_1 \Gamma, s_1 a) \vdash t_2 : b) s_1$**

where  $\text{gen}(\Gamma, T) = \forall \text{tv}(T) \backslash \text{tv}(\Gamma). T$ .

22

## Variables in Environments

When comparing with the type of a variable in an environment, we have to make sure we create a new instance of their type as follows:

$\text{newInstance}(\forall a_1, \dots, a_n. S) =$

**let  $b_1, \dots, b_n$  fresh in**

$[b_1/a_1, \dots, b_n/a_n]S$

$TP(\Gamma \vdash t : T) =$

**case  $t$  of**

$x : \{ \text{newInstance}(\Gamma(x)) \doteq T \}$

...

23

## Hindley/Milner in Programming Languages

Here is a formulation of the map example in the Hindley/Milner system.

```
let map = λf.λxs in
  if (isEmpty xs) nil
  else cons (f (head xs)) (map f (tail xs))
...
// names: List[String]
// nums : List[Int]
// map : ∀a.∀b.(a → b) → List[a] → List[b]
...
map toUpperCase names
map increment nums
```

24

## Limitations of Hindley/Milner

Hindley/Milner still does not allow parameter types to be polymorphic.  
I.e.

$$(\lambda x.xx)(\lambda y.y)$$

is still ill-typed, even though the following is well-typed:

$$\text{let } id = \lambda y.y \text{ in } id \ id$$

With explicit polymorphism the expression could be completed to a well-typed term:

$$(\Lambda a.\lambda x : (\forall b : b \rightarrow b). x[a \rightarrow a](x[a]))(\Lambda c.\lambda y : c. y)$$

25

## The Essence of let

We regard

$$\text{let } x = t \text{ in } t'$$

as a shorthand for

$$[t/x]t'$$

We use this equivalence to get a revised Hindley/Milner system.

**Definition:** Let  $HM'$  be the type system that results if we replace rule (LET) from the Hindley/Milner system  $HM$  by:

$$(LET') \frac{\Gamma \vdash t : T \quad \Gamma \vdash [t/x]t' : U}{\Gamma \vdash \text{let } x = t \text{ in } t' : U}$$

26

**Theorem:**  $\Gamma \vdash_{HM} t : S$  iff  $\Gamma \vdash_{HM'} t : S$

The theorem establishes the following connection between the Hindley/Milner system and the simply typed lambda calculus  $F_1$ :

**Corollary:** Let  $t^*$  be the result of expanding all **let**'s in  $t$  according to the rule

$$\text{let } x = t \text{ in } t' \rightarrow [t/x]t'$$

Then

$$\Gamma \vdash_{HM} t : T \Rightarrow \Gamma \vdash_{F_1} t^* : T$$

Furthermore, if every **let**-bound name is used at least once, we also have the reverse:

$$\Gamma \vdash_{F_1} t^* : T \Rightarrow \Gamma \vdash_{HM} t : T$$

27

## Principal Types

**Definition:** A type  $T$  is a **generic instance** of a type scheme  $S = \forall \alpha_1 \dots \forall \alpha_n. T'$  if there is a substitution  $s$  on  $\alpha_1, \dots, \alpha_n$  such that  $T = sT'$ . We write in this case  $S \leq T$ .

**Definition:** A type scheme  $S'$  is a **generic instance** of a type scheme  $S$  iff for all types  $T$

$$S' \leq T \Rightarrow S \leq T$$

We write in this case  $S \leq S'$ .

28

**Definition:** A type scheme  $S$  is *principal* (or: *most general*) for  $\Gamma$  and  $t$  iff

- $\Gamma \vdash t : S$
- $\Gamma \vdash t : S'$  implies  $S \leq S'$

**Definition:** A type system  $TS$  has the *principal typing property* iff, whenever  $\Gamma \vdash_{TS} t : S$  then there exists a principal type scheme for  $\Gamma$  and  $t$ .

**Theorem:**

1.  $HM'$  without *let* has the p.t.p.
2.  $HM'$  with *let* has the p.t.p.
3.  $HM$  has the p.t.p.

Proof sketch: (1.): Use type reconstruction result for the simply typed lambda calculus. (2.): Expand all *let*'s and apply (1.). (3.): Use equivalence between  $HM$  and  $HM'$ .

These observations could be used to come up with a type reconstruction algorithm for  $HM$ . But in practice one takes a more direct approach.