

Foundations of Software Fall 2023

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 Γ , t and T , check whether $\Gamma \vdash t : T$

► **Type reconstruction:**

Given Γ 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

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

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

case t of

x : $\{\Gamma(x) \hat{=} T\}$

$\lambda x. t_1$: let a, b fresh in

$\{(a \rightarrow b) \hat{=} T\} \cup$

$TP(\Gamma, x : a \vdash t_1 : b)$

$t_1 \ t_2$: let a fresh in

$TP(\Gamma \vdash t_1 : a \rightarrow T) \cup$

$TP(\Gamma \vdash t_2 : a)$

3

Example

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

Then twice gives rise to the following equations (see blackboard).

4

Soundness and Completeness I

0.1 Definition: In general, a type reconstruction algorithm \mathcal{A} assigns to an environment Γ 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

0.2 Theorem: TP is sound and complete. Specifically:

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

where

$$a \text{ is a new type variable}$$

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

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

Here, FV 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

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

into an 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:

$$a_j \notin \text{FV}(T'_k) \text{ for } j = 1, \dots, n, k = j, \dots, n$$

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 $a \triangleq T$ with a not in $\text{FV}(T)$.

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.

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

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

8

A Unification Algorithm

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

```

mgu          : (Type  $\hat{=}$  Type)  $\rightarrow$  Subst  $\rightarrow$  Subst
mgu(T  $\hat{=}$  U)s  = mgu'(sT  $\hat{=}$  sU)s
mgu'(a  $\hat{=}$  a)s = s
mgu'(a  $\hat{=}$  T)s = s  $\cup$  {a  $\mapsto$  T}   if a  $\notin$  FV(T)
mgu'(T  $\hat{=}$  a)s = s  $\cup$  {a  $\mapsto$  T}   if a  $\notin$  FV(T)
mgu'(T1  $\rightarrow$  T2  $\hat{=}$  U1  $\rightarrow$  U2)s = (mgu(T2  $\hat{=}$  U2)  $\circ$  mgu(T1  $\hat{=}$  U1))s
mgu'(T  $\hat{=}$  U)s = error           in all other cases

```

9

Soundness and Completeness of Unification

0.3 Definition: A substitution u is a *unifier* of a set of equations $\{T_i \hat{=} U_i\}_{i=1, \dots, m}$ 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 $u' = s \circ u$.

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

10

From Judgements to Substitutions

```

TP : Judgement  $\rightarrow$  Subst  $\rightarrow$  Subst
TP( $\Gamma \vdash t : T$ ) =
  case t of
    x      : mgu( $\Gamma(x) \hat{=}$  T)
     $\lambda x. t_1$  : let a, b fresh in
                mgu((a  $\rightarrow$  b)  $\hat{=}$  T)  $\circ$ 
                TP( $\Gamma, x : a \vdash t_1 : b$ )
    t1 t2 : let a fresh in
                TP( $\Gamma \vdash t_1 : a \rightarrow T$ )  $\circ$ 
                TP( $\Gamma \vdash t_2 : a$ )

```

11

Soundness and Completeness II

One can show by comparison with the previous algorithm:

0.5 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( $\Gamma \vdash t : a$ )( $\emptyset$ )
  r is a substitution on  $FV(s(a)) \setminus FV(s(\Gamma))$ 

```

12

Polymorphism

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

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

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) then 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:

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \quad (\text{T-TABS})$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1 [T_2] : [X \mapsto T_2]T_{12}} \quad (\text{T-TAPP})$$

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)) then 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 $S ::= T \mid \forall X. S$

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

$$\begin{array}{c}
 \frac{x \notin \text{dom}(\Gamma')}{\Gamma, x : S, \Gamma' \vdash x : S} \quad (\text{T-VAR}) \\
 \frac{\Gamma, X \vdash t : T_1 \quad X \notin \text{FV}(\Gamma)}{\Gamma \vdash t : \forall X. T_1} \quad (\text{T-TABS}) \\
 \frac{\Gamma \vdash t : \forall X. T_1 \quad \Gamma \vdash t : \forall X. T_1}{\Gamma \vdash t : [X \mapsto T_2]T_1} \quad (\text{T-TAPP}) \\
 \frac{\Gamma \vdash t_1 : S \quad \Gamma, x : S \vdash t_2 : T}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T} \quad (\text{T-LET})
 \end{array}$$

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

$$\begin{array}{c}
 \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x. t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS}) \\
 \frac{\Gamma \vdash t_1 : T_2 \rightarrow T \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 t_2 : T} \quad (\text{T-APP})
 \end{array}$$

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.

$TP : Judgement \rightarrow Subst \rightarrow Subst$

$TP(\Gamma \vdash t : T)(s) =$
 $\text{case } t \text{ of}$
 \dots
 $\text{let } x = t_1 \text{ in } t_2 : \text{let } a, b \text{ fresh in}$
 $\text{let } s_1 = TP(\Gamma \vdash t_1 : a) \text{ in}$
 $TP(\Gamma, x : \text{gen}(s_1(\Gamma), s_1(a)) \vdash t_2 : b)(s_1)$

where $\text{gen}(\Gamma, T) = \forall X_1. \dots \forall X_n. T$ with $X_i \in FV(T) \setminus FV(\Gamma)$.

21

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:

$\text{newInstance}(\forall X_1. \dots \forall X_n. S) =$
 $\text{let } b_1, \dots, b_n \text{ fresh in}$
 $[X_1 \mapsto b_1, \dots, X_n \mapsto b_n]S$
 $TP(\Gamma \vdash t : T) =$
 $\text{case } t \text{ of}$
 $x : \{ \text{newInstance}(\Gamma(x)) \triangleq T \}$
 \dots

22

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) then nil
  else cons (f (head xs)) (map f (tail xs))
...
// names: List[String]
// nums : List[Int]
// map  : ∀X.∀Y.(X → Y) → List[X] → List[Y]
...
map toUpperCase names
map increment nums
```

23

Limitations of Hindley/Milner

Hindley/Milner still does not allow parameter types to be polymorphic. For example,

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

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

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

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

$(\lambda A. \lambda x. (\forall B. B \rightarrow B). x [A \rightarrow A] (x [A])) (\lambda C. \lambda y. C. y)$

24

The Essence of let

We regard

$\text{let } x = t_1 \text{ in } t_2$

as a shorthand for

$[x \mapsto t_1]t_2$

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

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

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash [x \mapsto t_1]t_2 : T}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T} \quad (\text{T-LET}')$$

25

Equivalence of the two systems

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

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

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

$$\text{let } x = t_1 \text{ in } t_2 \rightarrow [x \mapsto t_1]t_2$$

Then

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

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

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

26

Principal Types

0.9 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 $S \leq T$.

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

$$S' \leq T \implies S \leq T$$

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

0.11 Definition: A type scheme S is *principal* (or: *most general*) for Γ and t iff

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

27

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

0.13 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.

28