Type Reconstruction and Polymorphism

Week 9

based on slides by Martin Odersky

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.

From Judgements to Equations

$$
\begin{array}{l}
TP : \mathit{Judgement} \to \mathit{Equations} \\
TP(\Gamma \vdash t : T) = \\
\quad \textbf{case } t \textbf{ of} \\
\quad\quad \begin{array}{lll}
x & : & \{\Gamma(x) \hat{=} T\} \\
\lambda x.t' & : & \textbf{let } a, b \text{ fresh } \textbf{in} \\
 & & \{(a \to b) \hat{=} T\} \quad \cup \\
 & & TP(\Gamma, x : a \vdash t' : b) \\
t\ t' & : & \textbf{let } a \text{ fresh } \textbf{in} \\
 & & TP(\Gamma \vdash t : a \to T) \quad \cup \\
 & & TP(\Gamma \vdash t' : a)
\end{array}
\end{array}
$$

Example

Let `twice` $= \lambda f.\lambda x.f(f(x))$.

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

Soundness and Completeness I

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 $T \in \mathcal{A}(\Gamma, t)$.

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

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

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

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

Type Reconstruction and Unification

Problem: : Transform set of equations

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

into equivalent substitution

$$\{a_j \mapsto T'_j\}_{j=1,\ldots,n}$$

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

$$a_j \notin tv(T'_k) \quad \text{for } j = 1, \ldots, n, k = j, \ldots, n$$

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 definining

$$\begin{array}{lcl} s(T \to U) & = & sT \to sU \\ s(K[T_1, \ldots, T_n]) & = & K[sT_1, \ldots, sT_n] \end{array}$$

Substitutions are idempotent mappings from types to types, i.e. $s(s(T)) = s(T)$. (why?)

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

A Unification Algorithm

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

$$
\begin{array}{lcll}
mgu & : & (Type \mathrel{\hat=} Type) \to Subst \to Subst & \\
mgu(T \mathrel{\hat=} U)\ s & = & mgu'(sT \mathrel{\hat=} sU)\ s & \\
mgu'(a \mathrel{\hat=} a)\ s & = & s & \\
mgu'(a \mathrel{\hat=} T)\ s & = & s \cup \{a \mapsto T\} & \textbf{if } a \notin tv(T) \\
mgu'(T \mathrel{\hat=} a)\ s & = & s \cup \{a \mapsto T\} & \textbf{if } a \notin tv(T) \\
mgu'(T \to T' \mathrel{\hat=} U \to U')\ s & = & (mgu(T' \mathrel{\hat=} U') \circ mgu(T \mathrel{\hat=} U))\ s & \\
\multicolumn{4}{l}{mgu'(K[T_1, \ldots, T_n] \mathrel{\hat=} K[U_1, \ldots, U_n])\ s} \\
 & = & (mgu(T_n \mathrel{\hat=} U_n) \circ \ldots \circ mgu(T_1 \mathrel{\hat=} U_1))\ s & \\
mgu'(T \mathrel{\hat=} U)\ s & = & error & \text{in all other cases}
\end{array}
$$

Soundness and Completeness of Unification

Definition: A substitution u is a unifier of a set of equations $\{T_i \mathrel{\hat=} U_i\}_{i=1,\ldots,m}$ 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.

From Judgements to Substitutions

$$
\begin{array}{l}
TP : Judgement \to Subst \to Subst \\
TP(\Gamma \vdash t : T) = \\
\quad \textbf{case } t \textbf{ of} \\
\qquad \begin{array}{lcl}
x & : & \mathsf{mgu}(newInstance(\Gamma(x)) \mathrel{\hat=} T) \\
\lambda x.t' & : & \textbf{let } a, b \text{ fresh } \textbf{in} \\
 & & \mathsf{mgu}((a \to b) \mathrel{\hat=} T) \quad \circ \\
 & & TP(\Gamma, x : a \vdash t' : b) \\
t\ t' & : & \textbf{let } a \text{ fresh } \textbf{in} \\
 & & TP(\Gamma \vdash t : a \to T) \quad \circ \\
 & & TP(\Gamma \vdash t' : a)
\end{array}
\end{array}
$$

Soundness and Completeness II

One can show by comparison with the previous algorithm:

Theorem: TP is sound and complete. Specifically:

$$
\begin{array}{l}
\Gamma \vdash t : T \quad \text{iff} \quad T = r(s(a)) \\
\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\quad a \text{ is a new type variable} \\
\qquad\qquad\qquad\quad s = TP\ (\Gamma \vdash t : a)\ \{\} \\
\qquad\qquad\qquad\quad r \text{ is a substitution on } tv(s\ a) \backslash tv(s\ \Gamma)
\end{array}
$$

Strong Normalization

Question: Can Ω be given a type?

$$\Omega \quad = \quad (\lambda x.xx)(\lambda x.xx) :?$$

What about Y?

Self-application is not typable!

In fact, we have more:

Theorem: (Strong Normalization) If $\vdash t : T$, then there is a value V such that $t \to^* V$.

Corollary: Simply typed lambda calculus is not Turing complete.

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

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

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

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!

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:

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

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 [a]
  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
```


Translating to System F

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

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:

$$\text{Type Scheme} \quad S \quad ::= \quad T \mid \forall a.S$$

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 ... ; ...`).

Hindley/Milner Typing rules

$$(\textsc{Var}) \;\; \Gamma, x : S, \Gamma' \vdash x : S \qquad (x \notin dom(\Gamma'))$$

$$(\forall\mathrm{E}) \; \frac{\Gamma \vdash t : \forall a.T}{\Gamma \vdash t : [U/a]T} \qquad (\forall\mathrm{I}) \; \frac{\Gamma \vdash t : T \qquad a \notin tv(\Gamma)}{\Gamma \vdash t : \forall a.T}$$

$$(\textsc{Let}) \; \frac{\Gamma \vdash t : S \qquad \Gamma, x : S \vdash t' : T}{\Gamma \vdash \textbf{let}\ x = t\ \textbf{in}\ t' : T}$$

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

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

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.

$$TP : Judgement \to Subst \to Subst$$

$$\begin{array}{l} TP(\Gamma \ \vdash \ t : T) \ s = \\ \quad \textbf{\textit{case}} \ t \ \textbf{\textit{of}} \\ \quad\quad \ldots \\ \quad\quad \textbf{\textit{let}} \ x = t_1 \ \textbf{\textit{in}} \ t_2 \quad : \quad \textbf{\textit{let}} \ a, b \ \text{fresh} \ \textbf{\textit{in}} \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{\textit{let}} \ s_1 = TP \ (\Gamma \ \vdash \ t_1 : a) \ \textbf{\textit{in}} \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad TP \ (\Gamma, x : \textbf{gen}(s_1 \ \Gamma, s_1 \ a) \ \vdash \ t_2 : b) \ s_1 \end{array}$$

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

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:

$$\begin{array}{l} newInstance(\forall a_1, \ldots, a_n.S) = \\ \quad \textbf{\textit{let}} \ b_1, \ldots, b_n \ \text{fresh} \ \textbf{\textit{in}} \\ \quad [b_1/a_1, \ldots, b_n/a_n]S \\ TP(\Gamma \ \vdash \ t : T) = \\ \quad \textbf{\textit{case}} \ t \ \textbf{\textit{of}} \\ \quad\quad x \quad : \quad \{newInstance(\Gamma(x)) \ \hat{=} \ T\} \\ \quad\quad \ldots \end{array}$$

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
```

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:

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

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

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

The Essence of let

We regard

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

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:

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

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

$$\textbf{let } x = t \textbf{ in } t' \quad \to \quad [t/x]t'$$

Then

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

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

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

Principal Types

Definition: A type T is a generic instance of a type scheme $S = \forall\alpha_1 \ldots \forall\alpha_n.T'$ if there is a substitution s on $\alpha_1, \ldots, \alpha_n$ 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

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

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

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

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

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 Γ 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.

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.