Foundations of Software
Fall 2023

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, $\mathtt{t}$ and $\mathtt{T}$, check whether $\Gamma \vdash \mathtt{t} : \mathtt{T}$
- **Type reconstruction:**
  Given Γ and $\mathtt{t}$, find a type $\mathtt{T}$ such that $\Gamma \vdash \mathtt{t} : \mathtt{T}$

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

$$
\begin{array}{l}
\mathsf{TP} : \mathit{Judgement} \longrightarrow \mathit{Equations} \\
\mathsf{TP}(\Gamma \vdash \mathtt{t} : \mathtt{T}) = \\
\quad \mathtt{case\ t\ of} \\
\qquad \begin{array}{lcl}
\mathtt{x} & : & \{\Gamma(\mathtt{x}) \hat{=} \mathtt{T}\} \\
\lambda\mathtt{x}.\mathtt{t}_1 & : & \text{let } a, b \text{ fresh in} \\
 & & \{(a \to b) \hat{=} \mathtt{T}\} \quad \cup \\
 & & \mathsf{TP}(\Gamma, \mathtt{x} : a \vdash \mathtt{t}_1 : b) \\
\mathtt{t}_1\ \mathtt{t}_2 & : & \text{let } a \text{ fresh in} \\
 & & \mathsf{TP}(\Gamma \vdash \mathtt{t}_1 : a \to \mathtt{T}) \quad \cup \\
 & & \mathsf{TP}(\Gamma \vdash \mathtt{t}_2 : a)
\end{array}
\end{array}
$$

Example

Let $\texttt{twice} = \lambda\texttt{f}.\lambda\texttt{x}.\texttt{f (f x)}$.
Then `twice` gives rise to the following equations (see blackboard).

Soundness and Completeness I

0.1 Definition: In general, a type reconstruction algorithm $\mathcal{A}$ assigns to an environment Γ and a term $\mathtt{t}$ a set of types $\mathcal{A}(\Gamma, \mathtt{t})$.
The algorithm is *sound* if for every type $\mathtt{T} \in \mathcal{A}(\Gamma, \mathtt{t})$ we can prove the judgement $\Gamma \vdash \mathtt{t} : \mathtt{T}$.
The algorithm is *complete* if for every provable judgement $\Gamma \vdash \mathtt{t} : \mathtt{T}$ we have that $\mathtt{T} \in \mathcal{A}(\Gamma, \mathtt{t})$.

0.2 Theorem: TP is sound and complete. Specifically:

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

$$\begin{array}{l} \text{where} \\ \qquad a \text{ is a new type variable} \\ \qquad EQNS = \mathrm{TP}(\Gamma \vdash \texttt{t} : a) \\ \qquad \overline{b} = FV(EQNS) \setminus FV(\Gamma) \end{array}$$

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

Type Reconstruction and Unification

Problem: Transform set of equations

$$\{\mathtt{T}_i \hat{=} \mathtt{U}_i\}_{i=1,\,\ldots,\,m}$$

into an equivalent substitution

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

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

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

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 $a \hat{=} \mathtt{T}$ with a not in $FV(\mathtt{T})$.

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

$$s(\mathtt{T} \to \mathtt{U}) \quad = \quad s\mathtt{T} \to s\mathtt{U}$$

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

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

A Unification Algorithm

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

$$
\begin{array}{lcll}
\mathrm{mgu} & : & (\mathit{Type} \hat{=} \mathit{Type}) \to \mathit{Subst} \to \mathit{Subst} & \\
\mathrm{mgu}(\mathtt{T} \hat{=} \mathtt{U})s & = & \mathrm{mgu}'(s\mathtt{T} \hat{=} s\mathtt{U})s & \\
\mathrm{mgu}'(a \hat{=} a)s & = & s & \\
\mathrm{mgu}'(a \hat{=} \mathtt{T})s & = & s \cup \{a \mapsto \mathtt{T}\} & \text{if } a \notin \mathit{FV}(\mathtt{T}) \\
\mathrm{mgu}'(\mathtt{T} \hat{=} a)s & = & s \cup \{a \mapsto \mathtt{T}\} & \text{if } a \notin \mathit{FV}(\mathtt{T}) \\
\mathrm{mgu}'(\mathtt{T}_1 \to \mathtt{T}_2 \hat{=} \mathtt{U}_1 \to \mathtt{U}_2)s & = & (\mathrm{mgu}(\mathtt{T}_2 \hat{=} \mathtt{U}_2) \circ \mathrm{mgu}(\mathtt{T}_1 \hat{=} \mathtt{U}_1))s & \\
\mathrm{mgu}'(\mathtt{T} \hat{=} \mathtt{U})s & = & \mathit{error} & \text{in all other cases}
\end{array}
$$

Soundness and Completeness of Unification

0.3 Definition: A substitution u is a *unifier* of a set of equations $\{\mathtt{T}_i \hat{=} \mathtt{U}_i\}_{i=1,\ldots,m}$ if $u\mathtt{T}_i = u\mathtt{U}_i$, 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 $\mathrm{mgu}(EQNS)(\emptyset)$ computes the most general unifier of $EQNS$. If $EQNS$ has no unifier then $\mathrm{mgu}(EQNS)(\emptyset)$ fails.

From Judgements to Substitutions

$$
\begin{array}{l}
\mathsf{TP} : \mathit{Judgement} \to \mathit{Subst} \to \mathit{Subst} \\
\mathsf{TP}(\Gamma \vdash \mathtt{t} : \mathtt{T}) = \\
\quad \mathtt{case\ t\ of} \\
\qquad \begin{array}{lcl}
\mathtt{x} & : & \mathsf{mgu}(\Gamma(x) \hat{=} \mathtt{T}) \\
\lambda\mathtt{x}.\mathtt{t}_1 & : & \text{let } a, b \text{ fresh in} \\
 & & \mathsf{mgu}((a \to b) \hat{=} \mathtt{T}) \quad \circ \\
 & & \mathsf{TP}(\Gamma, \mathtt{x} : a \vdash \mathtt{t}_1 : b) \\
\mathtt{t}_1\ \mathtt{t}_2 & : & \text{let } a \text{ fresh in} \\
 & & \mathsf{TP}(\Gamma \vdash \mathtt{t}_1 : a \to \mathtt{T}) \quad \circ \\
 & & \mathsf{TP}(\Gamma \vdash \mathtt{t}_2 : a)
\end{array}
\end{array}
$$

Soundness and Completeness II

One can show by comparison with the previous algorithm:

0.5 Theorem: TP is sound and complete. Specifically:

$$\Gamma \vdash \mathtt{t} : \mathtt{T} \quad \text{iff} \quad \mathtt{T} = r(s(a))$$

where

- a is a new type variable
- $s = \mathsf{TP}(\Gamma \vdash \mathtt{t} : a)(\emptyset)$
- r is a substitution on $FV(s(a)) \setminus FV(s(\Gamma))$

Polymorphism

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

$$(\lambda \mathtt{x}.\mathtt{x}\ \mathtt{x})(\lambda \mathtt{y}.\mathtt{y})$$

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

$$(\lambda \mathtt{y}.\mathtt{y})(\lambda \mathtt{y}.\mathtt{y}) : a \to a$$

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) 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!

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*.

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:

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{X}.\mathtt{t}_2 : \forall \mathtt{X}.\mathtt{T}_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X}.\mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}} \quad \text{(T-TApp)}$$

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
```

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 instantations.

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

$$\textit{Type Scheme} \quad \mathtt{S} \quad ::= \quad \mathtt{T} \mid \forall \mathtt{X.S}$$

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 ... ; ...`).

Hindley/Milner Typing rules

$$\frac{\mathtt{x} \notin \mathit{dom}(\Gamma')}{\Gamma, \mathtt{x} : \mathtt{S}, \Gamma' \vdash \mathtt{x} : \mathtt{S}} \quad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t} : \mathtt{T}_1 \qquad \mathtt{X} \notin \mathit{FV}(\Gamma)}{\Gamma \vdash \mathtt{t} : \forall \mathtt{X}.\mathtt{T}_1} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \forall \mathtt{X}.\mathtt{T}_1}{\Gamma \vdash \mathtt{t} : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_1} \quad \text{(T-TApp)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{S} \qquad \Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{t}_2 : \mathtt{T}}{\Gamma \vdash \mathtt{let}\ \mathtt{x}\ =\ \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}} \quad \text{(T-Let)}$$

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

$$\frac{\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}} \quad \text{(T-App)}$$

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.

$$\mathrm{TP} : \mathit{Judgement} \to \mathit{Subst} \to \mathit{Subst}$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{TP}(\Gamma \vdash \mathtt{t} : \mathtt{T})(s) =} \\
\quad \mathtt{case\ t\ of} & & \\
\quad\quad \ldots & & \\
\quad\quad \mathrm{let}\ \mathrm{x} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 & : & \mathrm{let}\ a, b\ \mathrm{fresh\ in} \\
 & & \mathrm{let}\ s_1 = \mathrm{TP}(\Gamma \vdash \mathtt{t}_1 : a)\ \mathrm{in} \\
 & & \mathrm{TP}(\Gamma, \mathrm{x} : \mathrm{gen}(s_1(\Gamma), s_1(a)) \vdash \mathtt{t}_2 : b)(s_1)
\end{array}
$$

where $\mathrm{gen}(\Gamma, \mathtt{T}) = \forall \mathtt{X}_1 . \cdots \forall \mathtt{X}_n . \mathtt{T}$ with $\mathtt{X}_i \in \mathit{FV}(\mathtt{T}) \setminus \mathit{FV}(\Gamma)$.

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:

$$
\begin{array}{l}
\mathsf{newInstance}(\forall \mathtt{X}_1 \,.\, \cdots \mathtt{X}_n \,.\, \mathtt{S}) = \\
\qquad \mathsf{let}\ b_1, \ldots, b_n\ \mathsf{fresh\ in} \\
\qquad [\mathtt{X}_1 \mapsto b_1, \ldots, \mathtt{X}_n \mapsto b_n]\mathtt{S} \\
\mathsf{TP}(\Gamma \vdash \mathtt{t} : \mathtt{T}) = \\
\qquad \mathtt{case\ t\ of} \\
\qquad\quad \mathtt{x} \quad : \quad \{\, \mathsf{newInstance}(\Gamma(\mathtt{x})) \hat{=} \mathtt{T} \} \\
\qquad\quad \ldots
\end{array}
$$

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) 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
```

Limitations of Hindley/Milner

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

$$(\lambda \texttt{x.x x})(\lambda \texttt{y.y})$$

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

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

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

$$(\Lambda \texttt{A}.\lambda \texttt{x:}(\forall \texttt{B: B} \rightarrow \texttt{B}).\ \texttt{x [A} \rightarrow \texttt{A] (x [A]))}(\Lambda \texttt{C}.\lambda \texttt{y:C.y})$$

The Essence of **let**

We regard

$$\texttt{let } \texttt{x} = \texttt{t}_1 \texttt{ in } \texttt{t}_2$$

as a shorthand for

$$[\texttt{x} \mapsto \texttt{t}_1]\texttt{t}_2$$

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:

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \qquad \Gamma \vdash [\texttt{x} \mapsto \texttt{t}_1]\texttt{t}_2 : \texttt{T}}{\Gamma \vdash \texttt{let } \texttt{x} = \texttt{t}_1 \texttt{ in } \texttt{t}_2 : \texttt{T}} \qquad \text{(T-LET')}$$

Equivalence of the two systems

0.7 Theorem: $\Gamma \vdash_{\mathsf{HM}} \mathtt{t} : \mathtt{S}$ iff $\Gamma \vdash_{\mathsf{HM}'} \mathtt{t} : \mathtt{S}$
The theorem establishes the following connection between the Hindley/Milner system and the simply typed lambda calculus F_1:

0.8 Corollary: Let $\mathtt{t}^*$ be the result of expanding all `let`'s in t according to the rule

$$\mathtt{let}\ \mathtt{x} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 \quad \to \quad [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2$$

Then

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

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

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

Principal Types

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

0.10 Definition: A type scheme $\mathtt{S}'$ is a generic instance of a type scheme $\mathtt{S}$ iff for all types $\mathtt{T}$

$$\mathtt{S}' \leq \mathtt{T} \Longrightarrow \mathtt{S} \leq \mathtt{T}$$

We write in this case $\mathtt{S} \leq \mathtt{S}'$.

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

- $\Gamma \vdash \mathtt{t} : \mathtt{S}$
- $\Gamma \vdash \mathtt{t} : \mathtt{S}'$ implies $\mathtt{S} \leq \mathtt{S}'$

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

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.

Reading for next week

- Chapter 15 – Subtyping, up to section 15.5 included
- Chapter 16 – Metatheory of Subtyping