Type inference

Subject: Type inference

Readings: Pierce, chapter 22.

The only explicit type annotations in the typed lambda calculus extended with natural numbers and Booleans are on the variables of abstractions.

What if we erase these, and then attempt **reconstruction**?

Idea: replace the erased types with **type variables** chosen from A, B, . . . (our original unrestricted types).

$$\lambda x : \mathsf{Nat}\ x \quad \rightsquigarrow \quad \lambda x : A\ x$$

$$\lambda x : \mathsf{Nat}\ (x + 1) \quad \rightsquigarrow \quad \lambda x : A\ (x + 1)$$

A **type substitution** σ maps type variables to types.

Example:
$\sigma = \{A \mapsto \mathsf{Nat}, B \mapsto (A \to \mathsf{Bool})\}$.

$\mathit{dom}(\sigma) = \{A, B\}$,
$\mathit{range}(\sigma) = \{\mathsf{Nat}, A \to \mathsf{Bool}\}$.

Applying a substitution to a type:

$$\sigma(X) = T \text{ if } (X \mapsto T) \in \sigma$$
$$\sigma(X) = X \text{ if } X \notin \mathit{dom}(\sigma)$$
$$\sigma(\mathsf{Nat}) = \mathsf{Nat}\ ,\ \sigma(\mathsf{Bool}) = \mathsf{Bool}$$
$$\sigma(T_1 \to T_2) = \sigma(T_1) \to \sigma(T_2)$$

We apply substitutions to type environments and terms in the obvious way.

Composition of two substitutions:

$$\begin{aligned}\sigma \circ \gamma = &\{X \mapsto \sigma(T) \mid (X \mapsto T) \in \gamma\} \\ &\cup \{X \mapsto T \mid (X \mapsto T) \in \sigma, \\ &\qquad X \notin \mathit{dom}(\gamma)\}\end{aligned}$$

Type judgments behave nicely with respect to type substitutions.

Thm: (Pierce, 22.1.2) If $\Gamma \vdash t : T$, then $\sigma\Gamma \vdash \sigma t : \sigma T$.

There are (at least) two questions we can ask about a lambda-calculus term t using only unrestricted types, with an associated context Γ.

Are all substitutions well-typed?

For every σ, does $\sigma\Gamma \vdash \sigma t : T$ for some T?

This is the case, for example, for well-typed ML expressions that involve type variables.

(Parametric polymorphism.)

A different question: is some substitution well-typed?

Given (Γ, t), is there a solution (σ, T) such that $\sigma\Gamma \vdash \sigma t : T$?

This is **type inference** or **type reconstruction**.

Idea for typing an untyped t: create Γ associating each abstraction variable with a **fresh** type variable, then find σ, T such that $\sigma\Gamma \vdash \sigma t : T$.

Algorithms work by structural recursion on the term e.

Idea 1: Recursively find substitution that types subterms, then add to it as a result of the relationship between subterms. (Milner)

Idea 2: accumulate **constraints** (equations) that describe relationships between type variables, then solve the whole set. (Wand)

Idea 1 is fairly intuitive, and was historically first.

Idea 2 works for ascription, is easier to prove correct, easier to optimize, easier to extend.

Idea 2 is the one described in Pierce.

We will cover both ideas, starting with Idea 1.

Algorithm W

Algorithm W consumes a type environment Γ and a term t, and produces a solution (σ, T), that is, a substitution σ and type T such that $\sigma\Gamma \vdash \sigma t : T$.

Structural recursion on the term t
has three cases:
variable, abstraction, application.

Variable:
$W(\Gamma, x) = \langle \phi, T \rangle$, where $x : T \in \Gamma$.

Abstraction:
$W(\Gamma, \lambda x.t') = \langle \sigma, \sigma X \to T \rangle$, where $\langle \sigma, T \rangle = W((\Gamma, x : X), t')$ and X is a fresh type variable.

Application is trickier.

Consider the example $\lambda x.\lambda y.y\ x$.

Unification

A **unifier** of types T_1 and T_2 is a substitution σ such that $\sigma T_1 = \sigma T_2$.

It solves the type equation $T_1 = T_2$.

Example:
$T_1 = (\text{Int} \to T_3)$, $T_2 = (T_4 \to \text{Bool})$.

Algorithm U solves unification. It's what we need to complete our description of Algorithm W.

Algorithm U consumes two type expressions and produces a substitution (or signals an error if one cannot be found).

It uses structural recursion on the type expressions.

$U(b, b) = \phi$ (same base type)

$U(b, b') =$ **error**
(different base types)

$U(b, T_1 \to T_2) =$ **error**
(base type isn't function)

(same for $U(T_1 \to T_2, b)$)

$U(T_1 \to T_2, T_3 \to T_4) = \sigma_2 \circ \sigma_1$
where $\sigma_1 = U(T_1, T_3)$ and
$\sigma_2 = U(\sigma_1 T_2, \sigma_1 T_4))$

$U(X, X) = \phi$ (same type variable)

$U(X, T) =$ **error** if X **occurs** in T (circularity)

$U(X, T) = \{X \mapsto T\}$ otherwise

(same for $U(T, X)$)

The **occurs check** in unification is responsible for the error raised by OCaml on programs such as:

```
# let rec len lst =
    match lst with
      []     -> 0
    | [f::r] -> 1 + len r;;
```

```
Error: This expression has type 'a list
       but an expression was expected
       of type 'a list list
       The type variable 'a
       occurs inside 'a list
```

We can now finish the case of applications.

$$W(\Gamma, M\ N) = \langle \sigma_3 \circ \sigma_2 \circ \sigma_1, \sigma_3\ X \rangle$$

where X is a fresh type variable and
$\langle \sigma_1, T_1 \rangle = W(\Gamma, M)$
$\langle \sigma_2, T_2 \rangle = W(\sigma_1 \Gamma, N)$
$\sigma_3 = U(\sigma_2 T_1, T_2 \to X)$.

We can extend the inference/reconstruction algorithm to other language constructs in much the same way we did for type checking.

Milner (1978) specified Algorithm W and proved its soundness. There were earlier equivalents, notably Hindley (1969).

This is typically called Hindley-Milner type inference.

Damas (1982) proved completeness and that Algorithm W produces the **principal type** of a term.

Any other valid typing is equal to a substitution applied to the principal type.

The production of a principal type for each subexpression makes it easier to usefully signal type errors.

But the interleaving of recursion and unification makes proofs of desirable properties harder.

The algorithm described in Pierce (idea 2) first accumulates type constraints and then finds a substitution satisfying them.

We will now go over the constraint-based algorithm (and the associated proofs) in detail.

Pierce includes Bool and Nat in his presentation.

We will concentrate on the simply-typed lambda calculus with uninterpreted base types.

A **constraint set** C is a set of equations $\{S_i = T_i\}$.

A substitution σ **unifies** C if for all i, $\sigma S_i = \sigma T_i$.

We define inference rules for the constraint typing relation $\Gamma \vdash t : S \mid_\chi C$.

χ is a set of fresh type variables, omitted when possible.

There are three rules for the constraint typing relation, corresponding to the three cases for forming a term.

These give us a structurally recursive algorithm for computing S, C given Γ, t.

For variables:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid_{\phi} \{\}}$$

For abstractions:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid_\chi C}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2 \mid_\chi C}$$

For applications:

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \mid \chi_1 C_1 \quad \Gamma \vdash t_2 : T_2 \mid \chi_2 C_2 \\ \chi_1 \cap \chi_2 = \chi_1 \cap FV(T_2) = \chi_2 \cap FV(T_1) = \phi \\ X \notin \chi_1, \chi_2, T_1, T_2, C_1, C_2, t_1, t_2 \\ C' = C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}\end{array}}{\Gamma \vdash t_1\ t_2 : X \mid_{\chi_1 \cup \chi_2 \cup \{X\}} C'}$$

If $\Gamma \vdash t : S \mid_\chi C$, then
a solution for (Γ, t, S, C)
is a pair (σ, T) such that
σ unifies C and $\sigma S = T$.

The soundness theorem tells us that the constraint approach will not give us a wrong answer: a solution to the constraints gives us a solution to the original problem.

Soundness (22.3.5):
If $\Gamma \vdash t : S \mid_\chi C$ and (σ, T) is a solution for (Γ, t, S, C), it is also a solution for (Γ, t), that is, $\sigma\Gamma \vdash t : T$.

The proof is by induction on the depth of a derivation of $\Gamma \vdash t : S \mid_{\chi} C$.

There are three cases, depending on the last rule used.

For variables:

$$\frac{x : S \in \Gamma}{\Gamma \vdash x : S \mid \{\}}$$

(σ, T) is a solution for $(\Gamma, x : S, S, C)$, so $\sigma S = T$. Since $x : S \in \Gamma$,$x : T \in \sigma\Gamma$, and $\sigma\Gamma \vdash x : T$.

For abstractions:

$$\frac{\Gamma, x : T_1 \vdash t_2 : S_2 \mid C}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to S_2 \mid C}$$

(σ, T) is a solution for
$(\Gamma, \lambda x : T_1 . t_2, T_1 \to S_2, C)$,
so σ unifies C and $T = \sigma T_1 \to \sigma S_2$,
and $(\sigma, \sigma S_2)$ is a solution for
$((\Gamma, x : T_1), t_2, S_2, C)$.

By the inductive hypothesis, $(\sigma, \sigma S_2)$ is a solution for $((\Gamma, x : T_1), t_2)$, so $\sigma\Gamma, x : \sigma T_1 \vdash \sigma t_2 : \sigma S_2$.

So we can conclude that $\sigma\Gamma \vdash \lambda x : \sigma T_1.\sigma t_2 : \sigma T_1 \to \sigma S_2$, that is, (σ, T) is a solution for $(\Gamma, \lambda x : T_1.t_2)$.

The application case is similar (two invocations of the inductive hypothesis).

This concludes our discussion of the soundness of constraint typing.

The completeness theorem tells us that the constraint approach will not fail if there is an solution to the original problem, because every such solution can be extended to an solution for the constraints.

To state the completeness theorem, we need a notion of substitutions agreeing on a subset of variables.

$\sigma \setminus \chi$ is the substitution that behaves like σ except it is undefined for the variables in χ.

Completeness (22.3.7):
If $\Gamma \vdash t : S \mid_\chi C$, (σ, T) is a solution for (Γ, t), and $dom(\sigma) \cap \chi = \phi$, then there is a solution σ', T for (Γ, t, S, C) with $\sigma' \setminus \chi = \sigma$.

Once again, the proof is by induction on the depth of a derivation of $\Gamma \vdash t : S \mid_{\chi} C$.

See Pierce for the details.

We now need an algorithm to find a solution to a constraint set.

This is algorithm U generalized to a set of constraints in the obvious way.

$$
\begin{array}{l}
\mathit{unify}(C) = \\
\quad \text{if } C = \phi \text{ then } [\,] \text{ else} \\
\quad \text{let } \{S = T\} \cup C' = C \text{ in} \\
\quad \text{if } S = T \text{ then } \mathit{unify}(C') \\
\quad \text{else if } S = X \text{ and } X \notin \mathit{FV}(T) \\
\qquad \text{then } \mathit{unify}([X \mapsto T]C') \circ [X \mapsto T] \\
\quad \text{else if } T = X \text{ and } X \notin \mathit{FV}(S) \\
\qquad \text{then } \mathit{unify}([X \mapsto S]C') \circ [X \mapsto S] \\
\quad \text{else if } S = S_1 \to S_2 \text{ and } T = T_1 \to T_2 \\
\qquad \text{then } \mathit{unify}(C' \cup \{S_1 = T_1, S_2 = T_2\}) \\
\quad \text{else fail}
\end{array}
$$

We can prove not only that unification gives us the solution we want, but that it is the “best” solution, in some sense.

We say $\sigma \sqsubseteq \sigma'$ (σ is "more general" than σ') if $\sigma' = \gamma \circ \sigma$ for some γ.

A **principal unifier** for C is σ that unifies C such that $\sigma \sqsubseteq \sigma'$ for any σ' unifying C.

Theorem (22.4.5.1):
For all C, *unify*(C) terminates.

Proof: The quantity (m, n), where m is the number of distinct type variables in C and n is the total size of all types in C, is lexicographically decreasing with each recursive application.

Theorem (22.4.5.2):
For all C, if $\mathit{unify}(C) = \sigma$,
then σ unifies C.

Proof: Easy induction on the number of recursive applications of *unify*.

Theorem (22.4.5.3):
For all C, if $\mathit{unify}(C) = \sigma$,
then σ is a principal unifier for C.

Proof: Relatively easy induction on the number of recursive applications of *unify*.

A **principal solution** for (Γ, t) is a solution (σ, T) such that σ is a principal unifier, in which case we call T a **principal type**.

These theorems show that if (Γ, t) has a solution, it has a principal solution, and we have an algorithm to compute it.

Polymorphism

Simple `let`:

$W(\Gamma, \texttt{let } x = E \texttt{ in } B) =$

$\langle \sigma_2 \circ \sigma_1, T_2 \rangle$, where

$\langle \sigma_1, T_1 \rangle = W(\Gamma, E)$,

$\langle \sigma_2, T_2 \rangle = W((\sigma_1(\Gamma), x : T_1), B)$.

This corresponds to the following inference rule:

$$\frac{\Gamma \vdash E : T_1 \qquad \Gamma, \mathtt{x} : T_1 \vdash B : T_2}{\Gamma \vdash \mathtt{let\ x} = E\ \mathtt{in}\ B : T_2}$$

`letrec` can be typed in a similar fashion.

By adding lists, we can write functions such as `foldr`.

Algorithm W will type them as OCaml does.

But there is a problem.

Algorithm W will type $\lambda x.x$ as $X_i \to X_i$.

But using this might cause specialization.

$\texttt{let}\ id = \lambda x.x$
$\texttt{in if}\ id\ true\ \texttt{then}\ id\ 1\ \texttt{else}\ 0$

This is not typable using the simple-let approach.

Uses of a polymorphic function force it to be specialized, and different uses may be in conflict.

Idea 1: perform the substitution (works, is expensive, see Pierce for details).

Idea 2: type *id* as $\forall X.X \to X$.

Every time the polymorphic function is used, instantiate its type as $X_i \to X_i$ (where X_i is a fresh type variable).

We permit quantifiers at the very left (top) of a type expression.

We introduce such quantifiers when a polymorphic function is given a name via `let`.

With type environment Γ, to type

$\texttt{let}\ x = E\ \texttt{in}\ B$

we let $\langle \sigma_1, T_1 \rangle = W(\Gamma, E)$, as before. But we form T_1' from T_1 by possibly adding quantifiers.

To describe quantifier creation, we make the following definition:

A type variable occurs **free** in a type environment if it occurs in a type expression in the environment and it is not quantified in that expression.

A type variable can occur free in a type environment if, for example, it is introduced while dealing with an application.

Such variables may later be given a meaning through unification, and shouldn’t be quantified at this point.

To form T_1', we quantify with $\forall$ any type variable in T_1 not occurring free in $\sigma_1(\Gamma)$.

Let-polymorphism:

$W(\Gamma, \text{let } x = E \text{ in } B) = \langle \sigma_2 \circ \sigma_1, T_2 \rangle$,
where
$\langle \sigma_1, T_1 \rangle = W(\Gamma, E)$,
$\langle \sigma_2, T_2 \rangle = W((\sigma_1(\Gamma), x : T_1'), B)$

T_1' is T with variables not occurring free in $\sigma_1(\Gamma)$ quantified.

This corresponds to the inference rule:

$$\frac{\Gamma \vdash E : T_1 \qquad \Gamma, \mathtt{x} : \forall \alpha . T_1 \vdash B : T_2}{\Gamma \vdash \mathtt{let\ x\ =}\ E\ \mathtt{in}\ B : T_2}$$

where α is all type variables not free in Γ.

For `let`-polymorphism, quantifier elimination occurs when an identifier bound to a polymorphic function is looked up in the type environment.

Quantifiers are removed from the result, and the quantified variables replaced with fresh type variables.

This corresponds to the inference rule:

$$\frac{x : \forall \alpha . T \in \Gamma}{\Gamma \vdash x : [\alpha_i \mapsto X_i] T}$$

where each X_i is a fresh type variable.

Now our example is typable.

$\texttt{let}\ id = \lambda x.x$
$\texttt{in if}\ id\ true\ \texttt{then}\ id\ 1\ \texttt{else}\ 0$

Let-polymorphism works well in practice, though one can construct pathological examples that take exponential time to typecheck (Pierce, 22.7, see errata).

Because of a potential hole in the type system, ML imposes the **value restriction**: generalization through quantification only occurs for syntactic values (constants, variables, abstractions).

The hole occurs when polymorphism is combined with references.

```
let r = ref (fun x -> x) in
(r := fn x=>x+1 ; (!r) true)
```

Under our rules, `r` would be given the type $\forall X.\mathit{Ref}\ (X \to X)$ in the first line.

The use of `r` in the second line will, by our specialization rule, give it type $\mathit{Ref}\ (X_1 \to X_1)$.

The RHS of the assignment has type Int $\to$ Int, so X_1 will be unified with Int.

The use of `r` in the third line will give it type $\mathit{Ref}\ (X_2 \to X_2)$.

The dereference and application to `true` will cause X_2 to be unified with Bool.

The example will typecheck, even though it shouldn’t.

If the program is run, a type error should occur when the third line is executed.

The value restriction forbids both uses of `r`, so this example will no longer compile.

But the value restriction also forbids reasonable programs, even ones that do not use mutation.

A simple way to force OCaml to invoke the value restriction:

```
let id = fun x -> x in
let id2 = id id in
if id2 true then id2 0
            else id2 1
```

This will not typecheck.

Here's a fix:

```
let id = fun x -> x in
let id2 = fun x -> id id x in
if id2 true then id2 0
            else id2 1
```

OCaml can weaken the value restriction in certain circumstances.

Next: Haskell