1 Type inference

Java and ML are statically typed languages, meaning every binding has a type that is determined at compile time—that is, before any part of the program is executed. The type-checker is a compile-time procedure that either accepts or rejects a program. By contrast, JavaScript and Ruby are dynamically-typed languages; the type of a binding is not determined ahead of time and computations like binding 42 to \( x \) and then treating \( x \) as a string result in run-time errors. We will spend a later lecture comparing the advantages and disadvantages of static versus dynamic typing.

Unlike Java, ML is implicitly typed, meaning programmers rarely need to write down the types of bindings. This is often convenient, especially with higher-order functions. (Although some people disagree as to whether it makes code easier or harder to read). But implicit typing in no way changes the fact that ML is statically typed. Rather, the type-checker has to be more sophisticated because it must infer what the type annotations “would have been” had the programmers written all of them. In principle, type inference and type checking could be separate steps (the inferencer could figure out the types then the checker could determine whether the program is well-typed), but in practice they are often merged.

Whether type inference for a particular programming language is easy, hard, or impossible (in the halting-problem sense of CSci 3313) is often hard to determine. It is not proportional to how permissive the type system is. For example, the “extreme” type systems that “accept everything” and “accept nothing” both make type inference very easy.

2 ML type inference

ML was rather cleverly designed so that type inference is a straightforward algorithm. At a very high level, that algorithm works as follows:

- Determine the types of bindings in order, using the types of earlier bindings to infer the types of later ones. (This is why you cannot use later bindings in a file.)
- For each \texttt{val} or \texttt{fun} binding, analyze the binding to determine constraints about its type. For example, if the inferencer sees \texttt{x+1}, it concludes that \( x \) must have type \texttt{int}. It gathers similar constraints for function calls, pattern matches, etc.
- Use those constraints to solve for the type of the binding.

The ML type inference algorithm is guaranteed never to reject a program that could type-check, if the programmer had written down types. It’s also guaranteed never to accept a program that cannot possibly type check. So explicit type annotations really are optional. The one exception to that is record selectors like \( \#1 \), which complicate type inference (and forced us to write down types on homework 1).

Since it would be verbose to keep writing “the ML type inference algorithm,” we’ll call the algorithm HM. We choose that name because it was independently invented by Roger Hindley and Robin Milner.

3 Gathering constraints: Examples

To gather the constraints for a binding, HM does the following:
• Assign a preliminary type to every (sub)expression in the binding, as well as to the binding itself. For known operations and constants, such as + and 3, use the type that is already known for it. For anything else, use a new type variable that hasn’t been used anywhere else.

• Use the “shape” of the expressions to generate constraints. For example, if an expression involves applying a function to an expression, then generate a constraint requiring the type of the expression to be the same as the function’s argument type.

Example 1.

- fun g(x) = 5 + x;
val g = fn : int -> int

1. Assign preliminary types.

Recall that 5+x is really syntactic sugar for +(5,x), because + is an infix operator.

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>R</td>
</tr>
<tr>
<td>+(5,x)</td>
<td>S</td>
</tr>
<tr>
<td>+</td>
<td>int * int -&gt; int</td>
</tr>
<tr>
<td>(5,x)</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
</tr>
<tr>
<td>x</td>
<td>U</td>
</tr>
</tbody>
</table>

Note that we’re using upper-case italic letters to represent these type variables rather than the usual ML syntax of ‘a, etc. That’s because they’re not quite the same as ML type variables, but the exact difference doesn’t really matter for now.

Also note that x appears twice as a subexpression—once as the function’s argument (g(x)) and once in the function’s body (5 + x). We assign it only a single preliminary type U, rather than assigning each appearance of it a distinct preliminary type.

2. Generate constraints.

• For function bindings, constraints are generated according to this rule:

  *Function binding:* In binding fun f x = e, if the type of f is A, and the type of x is B, and the type of e is C, then \( A = B \rightarrow C \).

  That rule is applicable to our example. The constraint it generates is \( R = U \rightarrow S \).

• For function applications, constraints are generated according to the following rule:

  *Function application:* If the type of f is A, and the type of e is B, and the type of f e is C, then \( A = B \rightarrow C \).

  That rule is applicable to subexpression +(5,x). The constraint it generates is \( \text{int} \times \text{int} \rightarrow \text{int} = T \rightarrow S \).

• For tuple constructors, constraints are generated according to the following rule:

  *Tuple constructor:* If the type of (e1,e2) is A, and the type of e1 is B, and the type of e2 is C, then \( A = B \times C \).

  That rule is applicable to subexpression (5,x). The constraint it generates is \( T = \text{int} \times U \).
The set of constraints thus generated is

\[ R = U \rightarrow S, \]
\[ \text{int} \times \text{int} \rightarrow \text{int} = T \rightarrow S, \]
\[ T = \text{int} \times U. \]

We’ll later examine in detail how to solve constraint sets. For now, it’s easy enough to work your way through a solution to this particular set: starting from the last constraint, we know \( T \) must be \( \text{int} \times U \). Substituting that into the second constraint, we get that \( \text{int} \times \text{int} \rightarrow \text{int} \) must equal \( \text{int} \times U \rightarrow S \), so \( S = U = \text{int} \). Substituting for \( S \) and \( U \) in the first constraint, we get that \( R = \text{int} \rightarrow \text{int} \). So the inferred type of \( g \) is \( \text{int} \rightarrow \text{int} \).

Note that humans doing type inference “in their head” often take shortcuts—just like humans doing long division in their head—but the point is there is an algorithm that methodically goes through the code gathering constraints and putting them together to get the answer.

**Example 2.**

```haskell
- fun apply(f,x) = f x;
val apply = fn : ('a -> 'b) * 'a -> 'b
```

1. **Assign preliminary types.**

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply (f,x)</td>
<td>( R )</td>
</tr>
<tr>
<td>f</td>
<td>( S )</td>
</tr>
<tr>
<td>x</td>
<td>( T )</td>
</tr>
<tr>
<td>f x</td>
<td>( U )</td>
</tr>
<tr>
<td></td>
<td>( V )</td>
</tr>
</tbody>
</table>

2. **Generate constraints.**

- By the function binding constraint rule, we have that \( R = S \rightarrow V \).
- By the function application constraint rule, we have that \( T = U \rightarrow V \).
- By the tuple constructor constraint rule, we have that \( S = T \times U \).

The set of constraints thus generated is

\[ R = S \rightarrow V, \]
\[ T = U \rightarrow V, \]
\[ S = T \times U. \]

Starting with the last constraint, we have that \( S \) is \( T \times U \). Substituting for \( S \) in the first constraint, we have that \( R = (T \times U) \rightarrow V \). From the second constraint, we have that \( T \) is \( U \rightarrow V \). Substituting that into the first constraint, we have that \( R = (U \rightarrow V) \times U \rightarrow V \). There are no further substitutions to be made, so we’re done solving the constraints. So the type of \( \text{apply} \) is \( (U \rightarrow V) \times U \rightarrow V \). If we replace \( U \) with \( 'a \) and \( V \) with \( 'b \), we get that the type of \( \text{apply} \) is \( ('a \rightarrow 'b) \times 'a \rightarrow 'b \). And that’s exactly what the REPL infers.
Example 3.

- apply(g,3);
val it = 8 : int

One way to determine the type of this example would, of course, be to have the run-time evaluate the expression, then determine what the type of the resulting value is. But ML actually infers the type of the expression before evaluating it—which is good, because it ensures type errors are found at compile time, rather than run time.

1. Assign preliminary types.

In this running example, the inference for \( g \) and \( \text{apply} \) have already been done, so we can fill in their types as “known”, much like the type of + is already known.

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply(g,3)</td>
<td>( R )</td>
</tr>
<tr>
<td>apply</td>
<td>( (U \rightarrow V) \times U \rightarrow V )</td>
</tr>
<tr>
<td>(g,3)</td>
<td>( S )</td>
</tr>
<tr>
<td>g</td>
<td>( \text{int} \rightarrow \text{int} )</td>
</tr>
<tr>
<td>3</td>
<td>( \text{int} )</td>
</tr>
</tbody>
</table>

2. Generate constraints.

- By the function application constraint rule, we have that \( (U \rightarrow V) \times U \rightarrow V = S \rightarrow R \).
- By the tuple constructor constraint rule, we have that \( S = (\text{int} \rightarrow \text{int}) \times \text{int} \).

The set of constraints thus generated is

\[
(U \rightarrow V) \times U \rightarrow V = S \rightarrow R, \\
S = (\text{int} \rightarrow \text{int}) \times \text{int}.
\]

Starting with the last constraint, we have that \( S \) is \( (\text{int} \rightarrow \text{int}) \times \text{int} \). Substituting that into the first constraint, we have that \( (U \rightarrow V) \times U \rightarrow V = (\text{int} \rightarrow \text{int}) \times \text{int} \rightarrow R \). So \( U = V = R = \text{int} \). The inferred type of \( \text{apply}(g,3) \) is therefore \( \text{int} \).

Example 4.

- apply(not,false);
val it = true : bool

By essentially the same reasoning as in example 3, HM can infer that the type of this expression is \( \text{bool} \). This illustrates the polymorphism of \( \text{apply} \): because the type \( (U \rightarrow V) \times U \rightarrow V \) of \( \text{apply} \) contains type variables, the function can be applied to any arguments, so long as those arguments’ types can be consistently substituted for the type variables.

4 Gathering constraints: Algorithm

We now present an algorithm that generates constraints. This algorithm is a precise description of how constraint gathering works in the examples we discussed above. The algorithm is not exactly what HM does, because HM actually performs type checking at the same time as type inference. However, the resulting types are the same, and separating inference from checking hopefully will give you a clearer idea of how inference itself works.
We’ll consider only a very small subset of ML expressions here: variables \( x \), anonymous functions \( \text{fn} \ x \Rightarrow e \), and function call \( e_1 \ e_2 \). (So there are no integers, no pattern matching, etc.) This subset has a name: the lambda calculus. It’s a remarkable fact that every (full) ML program can actually written in just the lambda calculus. It’s kind of the “assembly language” of functional programming—small, completely expressive, and rather unpleasant to write real programs in. So it’s sufficient to show how to do type inference just for the lambda calculus, even though in real language implementations of ML we’d want to do type inference for the entire language.

The algorithm takes as input a lambda calculus expression \( e \). We’ll assume that every function \( \text{fn} \ x \Rightarrow e_1 \) in that expression has a variable with a different name. (If not, our algorithm could make a pre-pass to rename variables. This is easy because of lexical scope.) The output of the algorithm is a set of constraints.

The first thing the algorithm does is to assign a unique type variable (e.g., \( R \), \( S \), \( T \)),

- one to each variable \( x \) bound by a function in \( e \), and
- one to each occurrence of each subexpression of \( e \).

Call the type variable assigned to \( x \) in the former clause \( D(x) \), and call the type variable assigned to occurrence of a subexpression \( e' \) in the latter clause \( U(e') \).

Next, our algorithm generates the following constraints:

- \( U(x) = D(x) \) for each occurrence of a variable \( x \),
- \( U(e_1) = U(e_2) \Rightarrow U(e_1 \ e_2) \) for each occurrence of a subexpression \( e_1 \ e_2 \),
- \( U(\text{fn} \ x \Rightarrow e) = D(x) \Rightarrow U(e) \) for each occurrence of a subexpression \( \text{fn} \ x \Rightarrow e \).

The latter two cases essentially implement the function application and function binding rules from our earlier examples. Our earlier examples didn’t use the first case because we streamlined (i.e., simplified) the examples not to use it.

The result is a set of constraints, which is the output of the algorithm. It’s not too hard to implement this algorithm as a recursive function over a tree representing the syntax of \( e \). We used such trees in the lectures on datatypes earlier in the semester.

**Example.** Given expression \( \text{fn} \ x \Rightarrow (\text{fn} \ y \Rightarrow x) \), a type variable \( R \) is associated with variable \( x \), and \( S \) with variable \( y \); and \( T \) with the occurrence of \( \text{fn} \ x \Rightarrow (\text{fn} \ y \Rightarrow x) \), and \( X \) with the occurrence of \( (\text{fn} \ y \Rightarrow x) \), and \( Y \) with the occurrence of \( x \). (Note that the names we’ve chosen for the type variables are completely arbitrary.) The constraints generated are \( T = R \Rightarrow X \), and \( X = S \Rightarrow Y \), and \( Y = R \).

## 5 Solving constraints: Algorithm

What does it mean to solve a set of constraints? To answer this question, we define *type substitutions*. A type substitution is a map from a type variable to a type. For example, we’ll write \( (Y := \text{int} \Rightarrow \text{int}) \) for the substitution that maps type variable \( Y \) to \( \text{int} \Rightarrow \text{int} \). The way a substitution \( S \) operates on a type can be defined recursively:

\[
S(X) = \begin{cases} 
\text{if } S = (X := t) \text{ then } t \text{ else } X \\
S(t_1 \Rightarrow t_2) = S(t_1) \Rightarrow S(t_2)
\end{cases}
\]

A substitution \( S \) can be applied to a constraint \( t = t' \); the result \( S(t = t') \) is defined as \( S(t) = S(t') \). And a substitution can be applied to a set \( C \) of constraints; the result \( S(C) \) is the result of applying \( S \) to each of the individual constraints in \( C \).
Given two substitutions $S$ and $S'$, we write $S \circ S'$ for their composition: $(S \circ S')(t) = S(S'(t))$.

A substitution unifies a constraint $t_1 = t_2$ if $S(t_1) = S(t_2)$. A substitution $S$ unifies a set $C$ of constraints if $S$ unifies every constraint in $C$. For example, substitution $S = (Y := \text{int} \to \text{int}) \circ (X := \text{int})$ unifies constraint $X \to (X \to \text{int}) = \text{int} \to Y$, because

$$S(X \to (X \to \text{int})) = \text{int} \to \text{int} \to \text{int} = S(\text{int} \to Y).$$

To solve a set of constraints $C$, we need to find a substitution that unifies $C$. If there are no substitutions that satisfy $C$, where $C$ is the constraints generated from expression $e$, then $e$ is not typeable.

To find a substitution that unifies $C$, we use an algorithm appropriately called the unification algorithm. It is defined as follows:

- $\text{unify}(\emptyset) =$ the empty substitution
- $\text{unify}(\{t = t'\} \cup C) =$
  - if $t = X = t'$, then return $\text{unify}(C)$.
  - if $t = X$ and $X \not\in t'$, then let $S = (X := t')$, and return $\text{unify}(S(C)) \circ S$.
  - if $t' = X$ and $X \not\in t$, then let $S = (X := t)$, and return $\text{unify}(S(C)) \circ S$.
  - if $t = t_0 \to t_1$ and $t' = t'_0 \to t'_1$, then return $\text{unify}(C \cup \{t_0 = t'_0, t_1 = t'_1\})$.
  - otherwise, fail. There is no possible unifier.

In the second and third subcases, “$X \not\in t$” means that $X$ should not occur in $t$. It ensures that the algorithm doesn’t produce a cyclic substitution—for example, $(X := X \to X)$.

It’s possible to prove that the unification algorithm always terminates, and that it produces a result if and only if a unifier actually exists—that is, if and only if the set of constraints has a solution. Moreover, the solution the algorithm produces is the most general unifier, in the sense that if $S = \text{unify}(C)$ and $S'$ unifies $C$, then there must exist some $S''$ such that $S' = S'' \circ S$.

If $R$ is the type variable assigned to represent the type of the entire expression $e$, and if $S$ is the substitution produced by the algorithm, then $S(R)$ is the type inferred for $e$ by HM type inference. Call that type $t$. It’s possible to prove $t$ is the principal type for the expression, meaning that if $e$ also has type $t'$ for any other $t'$, then there exists a substitution $S$ such that $t' = S(t)$. So HM actually infers the most lenient type that is possible for any expression.

6 Hindley-Milner type inference

The core of HM type inference is essentially constraint generation followed by unification. However, we’re still missing one key component of HM: our lambda calculus was missing let expressions. Adding them in is actually a little tricky.

Consider the following code:

```ml
let
  val double = fn f => fn z => f (f z)
  val a = double (fn x => x+1) 1
  val b = double (fn x => not x) false
in
  ...
end
```
The inferred type for \( f \) in `double` would be \((X \rightarrow X)\). The use of `double` in the binding of `a` would produce the constraint \( X = \text{int} \), and the use of `double` in the binding of `b` would produce the constraint \( X = \text{bool} \). Those constraints are contradictory, causing unification to fail! The program is not typeable, given how we’ve defined HM so far.

There is a very nice solution to this called *let-polymorphism*, which is what ML actually uses. Let-polymorphism enables a polymorphic function bound by a `let` expression behave as though it has multiple types. The essential idea is to allow each usage of a polymorphic function to have its own instantiation of the type variables, so that contradictions like the one above can’t happen.

Other language features that complicate type inference are records (as we’ve already seen) and mutability (which we haven’t).

HM is usually a very efficient algorithm—you’ve probably never had to wait for the REPL to print the inferred types of your programs. In practice, it runs in approximately linear time. But in theory, there are some very strange programs that can cause its running-time to blow up. (Technically, it’s DEXPTIME-complete.) For fun, try typing the following code in the REPL:

```plaintext
val b = true
val f0 = fn x => x+1
val f = fn x => if b then f0 else fn y => x y
val f = fn x => if b then f0 else fn y => x y
... (* keep repeating that line *)
```

You’ll see the types get longer and longer...

**The history of HM.** HM has been rediscovered many times by many people. Curry used it informally in the 1950’s (perhaps even the 1930’s). He wrote it up formally in 1967 (published 1969). Hindley discovered it independently in 1969; Morris in 1968; and Milner in 1978. In the realm of logic, similar ideas go back perhaps as far as Tarski in the 1920’s. Commenting on this history, Hindley wrote,

> There must be a moral to this story of continual re-discovery; perhaps someone along the line should have learned to read. Or someone else learn to write.