1 The type inference problem

In the syntax for the simply-typed lambda calculus (and its extensions), we included type declarations. These type declarations made it possible to construct typing proofs with a simple recursive type checker derived directly from the typing rules. For example, we had a lambda abstraction term $\lambda x{:}\tau.\, e$ with a corresponding typing rule:

$$\frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau'}$$

Similarly, for extensions such as $\mathbf{let}$ and $\mathbf{rec}$, we exploited type annotations:

$$\frac{\Gamma, y{:}\tau \to \tau', x{:}\tau \vdash e : \tau'}{\Gamma \vdash \mathbf{rec}\ y{:}\tau \to \tau'.\ \lambda x.\, e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash e' : \tau'}{\Gamma \vdash \mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ e' : \tau'}$$

Suppose we didn't have the type annotations. Can we still type-check these terms? We know from programming in ML that it is possible. We can easily write typing rules for the language without type annotations:

$$\frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \to \tau'} \qquad \frac{\Gamma, y{:}\tau \to \tau', x{:}\tau \vdash e : \tau'}{\Gamma \vdash \mathbf{rec}\ y.\ \lambda x.\, e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash e' : \tau'}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau'}$$

But without the type annotations, a type checker doesn't know what type to bind variables like x and y to in the typing context Γ. It seems it must somehow guess.

The problem of type checking code without type annotations is the problem of *type inference*. It is also known as *type reconstruction*, which makes sense if we think of the problem as recovering the type annotations that were omitted.

To see how we might do type inference, consider inferring types by hand in the following example.

```
let d = λz. z+z in
 (λf.λx.λy.
   if (f x y) then
    f (d x) y
   else
    f x (f x y)
```

What is the type of this term? Because * is an operation on integers, clearly $\mathbf{z}$ has type $\mathbf{int}$, and therefore $\mathbf{d}$ has type $\mathbf{int} \to \mathbf{int}$. Since $\mathbf{d}$ is applied to $\mathbf{x}$, we know that $\mathbf{x{:}\ int}$, and the first argument to $\mathbf{f}$ has type $\mathbf{int}$. Because $(\mathbf{f\ x\ y})$ is used as a boolean, the return type of $\mathbf{f}$ must be $\mathbf{bool}$. Since $(\mathbf{f\ x\ y})$ is passed to $\mathbf{f}$ as the second argument, that argument must have type $\mathbf{bool}$, and so must variable $\mathbf{y}$. Therefore $\mathbf{f{:}\ int{\to}bool{\to}bool}$ and the whole term has type $\mathbf{(int{\to}bool{\to}bool){\to}int{\to}bool{\to}bool}$.

We can automate this process by rewriting our type system to include *type variables* that we solve for as part of type checking. We can write a type system that includes type variables T for the unannotated lambda calculus (plus $\mathbf{let}$, $\mathbf{rec}$):

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash b : B}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \tau_0 = \tau_1 \to T}{\Gamma \vdash e_0\ e_1 : T} \qquad \frac{\Gamma, x{:}T \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : T \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2} \qquad \frac{\Gamma, x{:}T_1, y{:}T_1 \to T_2 \vdash e : \tau_2 \quad \tau_2 = T_2}{\Gamma \vdash \mathbf{rec}\ y.\ \lambda x.\, e : T_1 \to T_2}$$

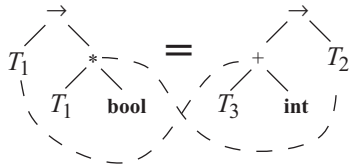
These are really just the same rules as above; the only difference is that some types are named T. Rather than guess the right type, a recursive type checker can create a new type variable for each such type. If the type variables are solved in a way that satisfies each type equation in the rules, then a typing derivation can clearly be constructed using the typing rules above.

Now we just have the problem of solving type equations. For example, consider the following type equation:

$$T_1 \to T_1 * \mathbf{bool} = (T_3 + \mathbf{int}) \to T_2$$

We want to find a solution for T_1, T_2, T_3 that satisfies this equation. The solution can be expressed as a substitution S that maps type variables T to types τ. Given a type τ, we write $S(\tau)$ for the result of replacing all type variables T in τ that are in dom(()S) with the corresponding type $S(T)$. (Because there are no type binders, we don't have to worry about variable capture.) Given an equation $\tau = \tau'$, our goal is to find a substitution S such that $S(\tau)$ and $S(\tau')$ are syntactically identical. This substitution is a *unifier* for the two types, and finding such an S is called *unification*.

To understand how to find a unifier, we think about the abstract syntax tree for the types being equated. Here are the ASTs for our example:



We walk down both trees in parallel until we either hit a contradiction (different type constructors) or a variable, in which case the variable must be equal to corresponding subtree. So we can see that we have a solution if $T_1 = T_3 + \mathbf{int}$ and $T_2 = T_1 * \mathbf{bool} = (T_3 + \mathbf{int}) * \mathbf{bool}$. The desired unifier is, then, $\{T_1 \mapsto T_3 + \mathbf{int}, T_2 \mapsto (T_3 + \mathbf{int}) * \mathbf{bool}\}$, which unifies both sides to $(T_3 + \mathbf{int}) \to (T_3 \to \mathbf{int}) * \mathbf{bool}$. However, since there are no constraints on T_3, other unifiers are possible, e.g. $\{T_1 \mapsto T_3 \to \mathbf{int}, T_2 \mapsto (T_3 \to \mathbf{int}) \to \mathbf{bool}, T_3 \mapsto \mathbf{unit}\}$, which unifies both sides to $(\mathbf{unit} + \mathbf{int}) \to (\mathbf{unit} \to \mathbf{int}) * \mathbf{bool}$. To allow us solve this set of equations in conjunction with other equations that may mention the same variables, and to preserve maximum polymorphism, we want the *weakest unifier*, where we say that substitution S_1 is weaker than S_2 if there exists a nontrivial substitution S_3 such that $S_2 = S_3 \circ S_1$. For example, in this case $\{T_1 \mapsto T_3 \to \mathbf{int}, T_2 \mapsto (T_3 \to \mathbf{int}) * \mathbf{bool}\}\{T_3 \mapsto \mathbf{unit}\}$ is the weakest unifier. It is weaker than the other substitution, which is seen using $S_2 = \{T_3 \mapsto \mathbf{unit}\}$.

2 Robinson's algorithm

Robinson's algorithm finds the weakest unifier for a finite set of equations $E = \{\tau_1 = \tau_1', \ldots \tau_n = \tau_n'\}$. We define this as a function $\mathit{unify}(E)$:

$$\begin{aligned}
\mathit{unify}(\emptyset) &= \emptyset \\
\mathit{unify}(\{B = B\} \cup E) &= \mathit{unify}(E) \\
\mathit{unify}(\{B_1 = B_2\} \cup E) &= \textit{error} && \text{(if } B_1 \neq B_2\text{)} \\
\mathit{unify}(\{T = T\} \cup E) &= \mathit{unify}(E) \\
\mathit{unify}(\{T = \tau\} \cup E) &= \mathit{unify}(\{\tau = T\} \cup E) = \mathit{unify}(E\{\tau/T\}) \circ \{T \mapsto \tau\} && \text{(if } T \text{ is not mentioned in } \tau\text{)} \\
\mathit{unify}(\{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'\} \cup E) &= \mathit{unify}(\{\tau_1 = \tau_1', \tau_2 = \tau_2'\} \cup E) \\
\mathit{unify}(\{\tau = \tau'\} \cup E) &= \textit{error} && \text{(if no previous rules match)}
\end{aligned}$$

This definition is well-founded though it is not immediately obvious why. Consider the uses of $\mathit{unify}(\cdot)$ that appear on the right-hand side. In each case, either the number of variables in the equations being solved becomes smaller or stays the same, and if the number of variables stays the same, the total size of the equations becomes smaller (counting the number of nodes in their abstract syntax). Therefore the definition is well-founded in an ordering on (number of variables, size of equations) in which the number of variables is primary.

What about the running time of this algorithm? It turns out to be exponential. For example, consider a program of the following form:

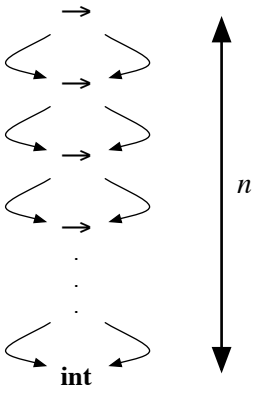
let b = true in
let f$_0$ = λx. x+1 in
let f$_1$ = λx. if b then f$_0$ else λy.x y in
let f$_2$ = λx. if b then f$_1$ else λy.x y in
$\vdots$
let f$_n$ = λx. if b then f$_{n-1}$ else λy.x y in
0

The types of the functions grow exponentially in n. If τ_n is the type of f_n, then $\tau_0 = \mathbf{int} \to \mathbf{int}$ and $\tau_{n+1} = \tau_n \to \tau_n$. So the substitution operations will take exponential time. On this example, running time will be exponential even though the type of the program is simply $\mathbf{int}$. In fact, the running time of the algorithm is in the worst case doubly exponential in the size of the program.

In practice, type inference takes linear time for ML, which is because programmers don't write code that involves extremely complex types. However, a program similar to this example will cause the SML type inference algorithm to take a long time. Is this unavoidable?

3 Type inference in polynomial time

With the algorithm as presented, types can grow very large. The real problem is how types are represented, however. Suppose we represent types using directed acyclic graphs (DAGs) rather than trees. Then τ_n can be represented as a DAG with only linear size:



To produce types of this sort through type inference, we restructure the type inference algorithm so that no substitution is involved. With every type that appears in the equations E, including type variables and subexpressions of types, we associate a distinct identity. The problem of type inference can then be expressed as determining which of these various identified types need to be equal to each other to satisfy all the type equations. Since type equality is transitive, this means that we need to sort the various types into disjoint sets in a way that satisfies the equations. The problem of computing disjoint sets can be solved efficiently using the union-find algorithm. With each type identity we have a reference cell that can be filled in to point to another type that the given type is equal to. For type variables, these reference cells will all initially be empty. For a type defined using a type constructor, the cell points to a node representing the constructor.

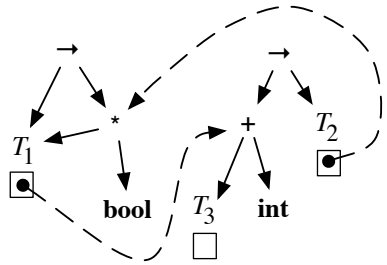
In general, a type variable cell can lead to a chain of pointers terminating in a cell that is either empty or points to a type constructor. This final cell is the representative for the whole disjoint set. If two types have the same representative, they are equal. Whenever we need to compare two types, we can quickly test whether they have the same representative before looking at their structure. To avoid repeatedly chasing chains of pointers, the union-find algorithm uses *path compression* : whenever a chain of pointers is chased, all the cells along the chain are updated to point directly to the representative.

The inference algorithm proceeds as before, except that it does not compute an explicit substitution, and an equation of the form $T = \tau$ is solved by making the reference cell for T point to the cell for τ.

Consider our earlier example:

$$T_1 \to T_1 * \mathbf{bool} = T_3 + \mathbf{int} \to T_2$$

Solving this equation results in the following data structure:



The solution proceeds by first splitting the equation into two equations $T_1 = T_3 + \mathbf{int}$ and $T_1 * \mathbf{bool} = T_2$. The first of these is solved by making T_1 point to the $+$ node. The second is solved by making T_2 point to the * node. We never need to set the cell for the T_3 node. Notice that we only have to do constant work to solve T_1, because updating its pointer once effectively substitutes it throughout all the equations.

How long does this algorithm take? It needs to compare any given pair of type nodes at most once, and takes $\alpha(n)$ time to do so assuming path compression. Therefore the total time is $O(n^2\alpha(n))$, which is polynomial. A tighter bound due to McAllester is $O(n\alpha(n)+nd)$, where d is the depth of the type schema. In practice this means linear time, though the worst case is quadratic—still a nice improvement over the simple implementation based on substitution. The absence of substitution tends to make this implementation faster in practice as well.