Reconstructing types

As we saw when trying (and failing) to type \( \lambda x : ? . xx \), we can use the type proof system not just to check typing, but to work out what the variable types must be, even if they’re not given to us in the syntax. We now formalize this.

The idea is to provisionally annotate the term with *type variables*, run through the type proof construction, and see what it tells us about the variables. For example, if we want to find the types for \( \lambda x . + x x \), we will give \( x \) a variable type \( \alpha \), so \( \lambda x : \alpha . + x x \), and then apply the simply typed proof rules – whereupon we will find that \( \alpha \) must be \texttt{nat} in order to make the proof work. On the other hand, if we try to do a proof on \( \lambda x : \alpha . xx \), we find that \( \alpha = \alpha \rightarrow \beta \) for some \( \beta \), which cannot happen.

To set this up formally, we need some book-keeping notation.

**Definition 92** There is a set \( \alpha, \beta, \ldots \) of *type variables*. Every type variable is a type. A type that contains no type variables is called a *closed type*.

A *type substitution* is a function mapping type variables to types (which may themselves include type variables). If \( T \) is a type substitution, and \( t \) is a term, then \( T t \) is the result of applying \( T \) to \( t \); similarly, we apply substitutions to types, assumptions, etc. We specify substitutions explicitly by the notation \( [T(\alpha)/\alpha, T(\beta)/\beta, \ldots] \).

For example, \( \alpha \rightarrow \texttt{bool} \) is a type, and applying the substitution \( [\texttt{nat}/\alpha] \) gives the closed type \( \texttt{nat} \rightarrow \texttt{bool} \).

The following property of substitutions is obvious, although technically it requires proof:

**Proposition 93** If \( \Gamma \vdash t : \tau \) and \( T \) is a type substitution, then \( T \Gamma \vdash T t : T \tau \).

To formalize the information we learn about variables while doing the proof, we extend the proof system to record equations between variables.

**Definition 94** The symbols \( \sigma, \tau \) are *meta-variables* representing types (possibly including type variables).

A *type equation* is an equation of the form \( \sigma = \tau \). A *solution* to the equation is a type substitution \( T \) such that \( T \sigma \) is identical to \( T \tau \). A solution to a set of equations is a substitution that solves all equations simultaneously.

The typing proof rules are modified as follows; each rule application may generate a type equation.

- The Constant Axiom (Schema) becomes
  \[
  \Gamma \vdash c : \tau
  \]
  and generates the equation \( \tau = \tau' \) where \( \tau' \) is the type assigned to \( c \).
- The Assumption Rule becomes
  \[
  \frac{x : \sigma' \in \Gamma}{\Gamma \vdash x : \sigma}
  \]
  and generates the equation \( \sigma' = \sigma \).
- The Abstraction Rule becomes
  \[
  \frac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x . t : \sigma}
  \]
  and generates the equation \( \alpha \rightarrow \beta = \sigma \), where \( \alpha, \beta \) are new type variables (i.e. not used elsewhere in the proof).
• The Application Rule becomes
\[
\Gamma \vdash t : \alpha \rightarrow \tau \quad \Gamma \vdash s : \alpha \\
\Gamma \vdash ts : \tau
\]
where \( \alpha \) is a new type variable.

Then we say \( t : \tau \) iff there is proof-tree for some \( \vdash t : \tau' \) and there is a solution \( T \) for the set of equations generated by the proof-tree, such that \( \tau = T \tau' \).  \( \Box \)

So in practice, if we want to infer the type of some term \( t \), we pick a new type variable \( \alpha \), and start constructing a tree for \( \vdash t : \alpha \). If we complete it, we solve the equations, and apply the resulting substitution to \( \alpha \).

The question remains of solving the equations. Because they have a very simple form, with \( \rightarrow \) as the only operator, it is easy to solve them by hand in simple cases. There is a general algorithm for solving such equations (and many others), called unification. Details may be found in the textbook, or on the Web – unification is nothing more than the mechanization of the manual technique. Using unification, we have

**Proposition 95** Given a set of type equations, we can in linear time either find a solution or show that no solution exists.

Moreover, we can find the most general solution, i.e. a solution \( T \) such that any other solution \( T' \) is equal to \( T'' \) applied to \( T \) for some \( T'' \).

For example, if we try to type the identity function \( \lambda x.x \), our tree will be:

\[
\frac{x : \beta \in \{x : \beta\} \quad \beta = \gamma}{\vdash \lambda x.x : \alpha}
\]

and solving the equations leaves us with \( \alpha = \beta \rightarrow \beta \). Thus the inference tells us that \( \lambda x.x \) can have type \( \beta \rightarrow \beta \) for any substitution of a type for \( \beta \) – for example, \( \text{nat} \rightarrow \text{nat} \), or \( \text{bool} \rightarrow \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool}) \). Note that we are **not** saying that a single term \( \lambda x.x \) can have all these types at the same time, only that when it is given a simple type, that type will be a substitution of \( \beta \rightarrow \beta \).

On the lecture slides, we did the example of finding a type for \( \lambda f.\lambda x.\lambda y.\text{suc}(f(\text{+} x y)) \), and in order to get a tree that fitted on a slide, we used a slightly abbreviated presentation, where we first assigned type variables to all the lambda variables, and compressed various steps in the proof. We shall now do the proof again in a different style: we will use the full mechanism described above, and we’ll also present it in a ‘goal-directed’ style, in other words upside-down from the proof-tree presentation. This shows how the inference algorithm in a compiler actually proceeds, and is also easier to lay out. To turn the following example into a proof tree, simply turn it upside down, and draw a line above each rule conclusion!

(see following page)
Example 96 The type inference for \( \lambda f. \lambda x. \lambda y. \text{suc}(f (x + y)) \) is as follows, where Const is the Constant Axiom, App is the Application Rule, Abs is the Abstraction Rule, and Ass is the Assumption Rule.

Goal: find \( \alpha \) such that \( \vdash \lambda f. \lambda x. \lambda y. \text{suc}(f (x + y)) : \alpha \): apply Abs, putting \( \alpha = \beta \to \gamma \), giving subgoal:

\[ \vdash f : \beta 
\vdash \lambda x. \lambda y. \text{suc}(f (x + y)) : \gamma : \text{apply Abs, putting } \gamma = \delta \to \epsilon, \text{ giving subgoal} \]

\[ \vdash f : \beta, x : \delta \vdash \lambda y. \text{suc}(f (x + y)) : \epsilon : \text{apply Abs, putting } \epsilon = \zeta \to \eta, \text{ giving subgoal} \]

\[ \vdash f : \beta, x : \delta, y : \zeta \vdash \text{suc}(f (x + y)) : \eta : \text{apply App, giving two subgoals} \]

\[ \vdash f : \beta, x : \delta, y : \zeta \vdash \theta : \text{apply Const, with } \theta \to \eta = \text{nat} \to \text{nat} \]

\[ \vdash f : \beta, x : \delta, y : \zeta \vdash f (x + y) : \theta : \text{apply App, giving two subgoals} \]

\[ \vdash f : \beta, x : \delta, y : \zeta \vdash f : \iota \to \theta : \text{apply Ass, with } \beta = \iota \to \theta \]

\[ \vdash f : \beta, x : \delta, y : \zeta \vdash (x + y) : \iota : \text{apply App, giving two subgoals} \]

\[ \vdash f : \beta, x : \delta, y : \zeta \vdash + : \omega \to (\kappa \to \iota) : \text{apply Const, with } \omega \to (\kappa \to \iota) = \text{nat} \to (\text{nat} \to \text{nat}) \]

\[ \vdash f : \beta, x : \delta, y : \zeta \vdash x : \omega : \text{apply Ass, with } \omega = \delta \]

\[ \vdash f : \beta, x : \delta, y : \zeta \vdash y : \kappa : \text{apply Ass, with } \kappa = \zeta \]

Now solve the underlined equations by substitution and matching, giving \( \omega = \kappa = \iota = \zeta = \delta = \theta = \eta = \text{nat} \) and the rest accordingly.

\( \triangleright \)