A Tutorial on Polymorphic Type Derivations

David Christiansen

DRAFT of December 27, 2012

This is a tutorial on how to produce typing derivation trees for a simplified version of ML's type system. Rather than being incredibly precise about every detail, this tutorial will focus on building up an understanding of why the rules work as they do. This tutorial is based on the presentation of polymorphic typing in *Programming Language Concepts* [Sestoft, 2012]. For your convenience, the grammar and type system of the small version of ML used in this tutorial are reproduced in Appendix A.

DRAFT This version of the tutorial has been used for one semester, but may still contain minor errors or incomplete explanations. Please send any comments about the style in which it is written or any corrections to `drc@itu.dk`.

Notational Conventions When looking at a formal definition of a type system, it can be difficult at first to distinguish the different categories of symbols from each other. Rather than sweep this issue under the rug, colors and other typographical conventions are used to precisely indicate which category a symbol is a part of. The following table describes the applied conventions:

Category	**Typeface**	**Examples**
ML keywords	Red, underlined	if, let
ML expressions	Red, sans-serif, upright	$17 + 25$, if $23 < 42$ then 13 else 99
Variables representing expressions in rules	Italic roman red	e_1, e_2, e_3
ML Types	Blue, sans-serif, upright	int, bool, $\alpha \to$ int
Variables representing ML types in rules	Italic roman blue	t_1
Type schemes and quantified type variables	Green	$\forall\alpha\ .\ \alpha \to \alpha$, $\forall\alpha\ .\ \alpha \to$ bool, $\forall\alpha, \beta\ .\ \alpha \to \beta \to \alpha$

The distinction between *variables representing types or expressions* and *actual types or expressions*, which is represented by writing the variables in italic type, is important. A final proof will not contain any of the variables, as they are used to demonstrate the structure of the proof.

What are we doing here, anyway? It is important to have the right perspective when looking at typing rules and how to combine them. Typing rules serve two purposes: they serve as a notation for helping humans keep track of complicated ideas, and they aid communication of these complicated ideas

between humans. Even though they are quite formal and their use is highly precise, they are not primarily intended for a computer. When we build up a tree structure of typing rules, we construct something called a *typing derivation*. In ML, a typing derivation is a proof that some expression *can have* some type, given a particular context, not that the expression can *only* have that type or that the expression will have that type no matter where we encounter it. More formally, a derivation for $\rho \vdash e : t$ is a proof that in some context ρ (which assigns types to variables in e), we can show that e can have (at least) the type t, and possibly others. This proof is something that we, as creative, thinking beings, need to come up with. We cannot necessarily construct it by mechanically following rules. In fact, the discovery of an efficient algorithm for type-checking in some type system is a discovery that is at least as important as that of the type system, because it enables it to be actually used, and these algorithms can easily be non-obvious. For an example of such an algorithm, see Section 6.4 of Sestoft [2012]. So, to recap, we are constructing a proof, for other humans, that a particular expression can have a particular type in ML.

1 Examples without let

We begin by using some simple examples to demonstrate how typing rules fit together, without using more advanced features of the type system.

If we want to show that $\underline{\text{if}}\ 2 < 3\ \underline{\text{then}}\ 42\ \underline{\text{else}}\ -5$ can have type int in the empty typing context (called "$\cdot$", which is just a dot), we begin by creating a tree with this fact at the bottom.

$$\frac{?}{\cdot \vdash \underline{\text{if}}\ 2 < 3\ \underline{\text{then}}\ 42\ \underline{\text{else}}\ -5 : \mathsf{int}}$$

Inspecting our typing rules, we see that only one of them has an if-expression at the bottom: rule P7. Applying this rule gives us three new typings to prove:

- $\cdot \vdash 2 < 3 : \mathsf{bool}$
- $\cdot \vdash 42 : \mathsf{int}$
- $\cdot \vdash -5 : \mathsf{int}$

Note that the variable t in P7 has been filled out with int. We do this because int is the only type can possibly be used, because the same t occurs on the right-hand-side of the colon in the bottom of P7. Our derivation now looks like:

$$\text{P7}\ \frac{\cdot \vdash 2 < 3 : \mathsf{bool} \quad \cdot \vdash 42 : \mathsf{int} \quad \cdot \vdash -5 : \mathsf{int}}{\cdot \vdash \underline{\text{if}}\ 2 < 3\ \underline{\text{then}}\ 42\ \underline{\text{else}}\ -5 : \mathsf{int}}$$

The two branches of the if-expression are easy to show – as they are integer literals, we merely need to apply rule P1 to them. Rule P5 is the only one that has a less-than expression on the bottom, so we know that we will use it for the condition of our if-expression. We now have the partial derivation:

$$\text{P7}\ \frac{\text{P5}\ \dfrac{?}{\cdot \vdash 2 < 3 : \mathsf{bool}} \quad \text{P1}\ \dfrac{}{\cdot \vdash 42 : \mathsf{int}} \quad \text{P1}\ \dfrac{}{\cdot \vdash -5 : \mathsf{int}}}{\cdot \vdash \underline{\text{if}}\ 2 < 3\ \underline{\text{then}}\ 42\ \underline{\text{else}}\ -5 : \mathsf{int}}$$

We now need only to show that the arguments to $<$ can both be integers in the empty environment. We do this by using rule P1 twice, completing the derivation:

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash 2 : \mathsf{int}}\,\text{P1} \quad \dfrac{}{\cdot \vdash 3 : \mathsf{int}}\,\text{P1}}{\cdot \vdash 2 < 3 : \mathsf{bool}}\,\text{P5} \quad \dfrac{}{\cdot \vdash 42 : \mathsf{int}}\,\text{P1} \quad \dfrac{}{\cdot \vdash -5 : \mathsf{int}}\,\text{P1}}{\cdot \vdash \mathsf{if}\ 2 < 3\ \mathsf{then}\ 42\ \mathsf{else}\ -5 : \mathsf{int}}\,\text{P7}$$

On the other hand, we cannot argue that $\mathsf{true} + 7$ has any type. Let's walk through the process to see where it goes wrong. We begin with a tree that has our desired term on the bottom. For now, we'll represent the result type with a variable t that we'll attempt to fill in later. If we can't fill it in, then our attempt to find a type for the expression failed. We'll use some arbitrary context ρ to begin with.

$$\dfrac{?}{\rho \vdash \mathsf{true} + 7 : t}$$

Just as before, we examine our typing rules to find one whose bottom matches the expression that we're finding a derivation for. In other words, it needs to have a $+$ in it. There is only one: P4. Filling in its premises yields:

$$\dfrac{\rho \vdash \mathsf{true} : \mathsf{int} \quad \rho \vdash 7 : \mathsf{int}}{\rho \vdash \mathsf{true} + 7 : \mathsf{int}}\,\text{P4}$$

We have replaced our variable t with the concrete type int, because that is the result specified by P4. Unfortunately, we've reached an impasse - there are no rules that allow us to give true the type int.

2 Types vs. type schemes

In our typing rules, we distinguish between *types* and *type schemes*. The difference is that type schemes are polymorphic – their variables can be instantiated to a number of different types. An important point is that not every type variable creates polymorphism – only those that are listed between the $\forall$ and the dot. In this tutorial, they are green. Other type variables that are not listed after the $\forall$ are simply useless, because there is no way to replace them with concrete types.

Two of the rules, P6 and P8, allow adding a $\forall$ and any number of variables to the beginning of a type. Using these rules enables us to create useful polymorphic types. This distinction between the quantified type variables and free type variables (that is, between the green type variables and the blue type variables) is used to ensure that functions and variables are first polymorphic *after* they are defined and not in their own bodies, according to the rules in Sestoft [2012], Section 6.2.

3 Anatomy of a typing rule

Before we continue to more complex examples, let's take a minute to examine our typing rules to really understand what they are saying. At the top level, a typing rule gives a list of zero or more conditions that must be satisfied in order to make some conclusion. These conditions, called *premises*, are written

above the line. The conclusion is written below the line. In other words, in the following rule Foo, P_1, P_2, and P_3 are the premises and C is the conclusion.

$$\textsc{Foo}\ \frac{P_1 \quad P_2 \quad P_3}{C}$$

Sometimes, if the premises are long, they may take more than one line:

$$\textsc{Foo}\ \frac{\begin{array}{c} P_1 \quad P_2 \\ P_3 \end{array}}{C}$$

This means exactly the same thing. A typing rule means that *all* premises must be proved in order for the conclusion to also be considered proven.

In our typing rules, we have the following things:

Typings relate an environment, and expression, and a type. The typing

$$\rho \vdash e : t$$

states that e can have the type t in an environment ρ.

Substitutions take a scheme and replace its quantified (that is, green) variables with types, yielding a new type. The substitution $[\mathsf{bool}/\alpha]\ \alpha \to \alpha \to \mathsf{int}$ yields the type $\mathsf{bool} \to \mathsf{bool} \to \mathsf{int}$.

Environment lookups check that some name is mapped to some type scheme in an environment. We typically use this when we expect something to have been added to the environment further down the tree, such as when a function or variable has been defined and we need to use its type. The assertion that x in our environment ρ has type scheme $\forall\alpha\ .\ \alpha \to \alpha$ is written $\rho\,(\mathsf{x}) = \forall\alpha\ .\ \alpha \to \alpha$. This only occurs in the rules together with substitution, which is why rule P3 can be called the *specialization* rule – it specializes a polymorphic type scheme from the environment to a useful concrete type.

Environment extensions take an environment and add some mappings to it, possibly replace ones that already exist. $\rho\,[\mathsf{x} \mapsto \mathsf{int}]$ denotes ρ extended with a mapping from x to int.

Side conditions are pieces of English text that capture a property that would be to complex or take up too much space to define purely mathematically. In the rules given here, only P6 and P8 have side conditions that state that the type variables being quantified are not free. The precise definition of this is somewhat subtle and beyond the scope of this tutorial – it is sufficient for you to know that you should pick unique names for your type variables, and then you will never need to worry about this condition.

4 Monomorphic let-bindings

Returning to our examples, we will now see how *monomorphic* functions and let-bound variables are typed. In other words, the examples in this section have only concrete types.

4.1 Variables

We begin with a simple let-binding. We will show that $\underline{\mathsf{let}}\ \mathsf{n} = 17\ \underline{\mathsf{in}}\ \mathsf{n} + 1\ \underline{\mathsf{end}}$ can have type int in any environment. We represent the "in any environment" part by simply placing some abstract environment variable ρ to the left of the turnstile. Because we never use the initial contents of ρ, they can be anything.

We want to show, therefore, $\rho \vdash \underline{\mathsf{let}}\ \mathsf{n} = 17\ \underline{\mathsf{in}}\ \mathsf{n} + 1\ \underline{\mathsf{end}} : \mathsf{int}$, using our typing rules. We begin as before, by creating a new tree with our goal at the bottom:

$$\frac{?}{\rho \vdash \underline{\mathsf{let}}\ \mathsf{n} = 17\ \underline{\mathsf{in}}\ \mathsf{n} + 1\ \underline{\mathsf{end}} : \mathsf{int}}$$

We have only one rule that applies: P6. We cannot use P8 because it only applies when there are *two* names following the $\underline{\mathsf{let}}$. The first premise of P6 states that we need to type-check the expression being bound to n, and then extend our environment with the type we discover for n and typecheck the body of the $\underline{\mathsf{let}}$ with this extended environment. When we extend our environment, we also have the possibility to create a polymorphic type, but we will not use that possibility here. See Section 5 for examples of this. Our current state is now:

$$\text{P6}\ \frac{\rho \vdash 17 : \mathsf{int} \qquad \rho\,[\mathsf{n} \mapsto \mathsf{int}] \vdash \mathsf{n} + 1 : \mathsf{int}}{\rho \vdash \underline{\mathsf{let}}\ \mathsf{n} = 17\ \underline{\mathsf{in}}\ \mathsf{n} + 1\ \underline{\mathsf{end}} : \mathsf{int}}$$

The first premise, that 17 can be an int, is easy to show using rule P1. In the second case, the only rule that has a $+$ in the conclusion is P4:

$$\text{P6}\ \frac{\text{P1}\ \dfrac{}{\rho \vdash 17 : \mathsf{int}} \qquad \text{P4}\ \dfrac{\rho\,[\mathsf{n} \mapsto \mathsf{int}] \vdash \mathsf{n} : \mathsf{int} \qquad \rho\,[\mathsf{n} \mapsto \mathsf{int}] \vdash 1 : \mathsf{int}}{\rho\,[\mathsf{n} \mapsto \mathsf{int}] \vdash \mathsf{n} + 1 : \mathsf{int}}}{\rho \vdash \underline{\mathsf{let}}\ \mathsf{n} = 17\ \underline{\mathsf{in}}\ \mathsf{n} + 1\ \underline{\mathsf{end}} : \mathsf{int}}$$

The interesting case here is showing that n can indeed have type int. To do this, we look up n in our extended environment using rule P3. We can safely ignore the substitutions performed in P3's conclusion, because we don't have any $\forall$-bound type variables to replace. Our final tree is then:

$$\text{P6}\ \frac{\text{P1}\ \dfrac{}{\rho \vdash 17 : \mathsf{int}} \qquad \text{P4}\ \dfrac{\text{P3}\ \dfrac{\rho\,[\mathsf{n} \mapsto \mathsf{int}]\,(\mathsf{n}) = \mathsf{int}}{\rho\,[\mathsf{n} \mapsto \mathsf{int}] \vdash \mathsf{n} : \mathsf{int}} \qquad \text{P1}\ \dfrac{}{\rho\,[\mathsf{n} \mapsto \mathsf{int}] \vdash 1 : \mathsf{int}}}{\rho\,[\mathsf{n} \mapsto \mathsf{int}] \vdash \mathsf{n} + 1 : \mathsf{int}}}{\rho \vdash \underline{\mathsf{let}}\ \mathsf{n} = 17\ \underline{\mathsf{in}}\ \mathsf{n} + 1\ \underline{\mathsf{end}} : \mathsf{int}}$$

4.2 The Fibonacci function

The Fibonacci function can be defined as follows in our little dialect of ML:

$$\underline{\mathsf{let}}\ \mathsf{fib}\ \mathsf{n} = \underline{\mathsf{if}}\ \mathsf{n} < 1\ \underline{\mathsf{then}}\ 1\ \underline{\mathsf{else}}\ \mathsf{fib}\ (\mathsf{n} + -1) + \mathsf{fib}\ (\mathsf{n} + -2)\ \underline{\mathsf{in}}\ \mathsf{fib}\ \underline{\mathsf{end}}$$

By looking at the function, we can see that it should have type $\mathsf{int} \to \mathsf{int}$ in any context. Thus, we will prove that

$$\rho \vdash \underline{\mathsf{let}}\ \mathsf{fib}\ \mathsf{n} = \underline{\mathsf{if}}\ \mathsf{n} < 1\ \underline{\mathsf{then}}\ 1\ \underline{\mathsf{else}}\ \mathsf{fib}\ (\mathsf{n} + -1) + \mathsf{fib}\ (\mathsf{n} + -2)\ \underline{\mathsf{in}}\ \mathsf{fib}\ \underline{\mathsf{end}} : \mathsf{int} \to \mathsf{int}$$

We are beginning to approach the maximum width that can be displayed on a printed page, so the derivation will be constructed in manageable pieces and presented whole at the end. Because we are defining a function, we must use

P8. This rule has two premises that we need to prove:

$$\rho\,[\mathsf{n} \mapsto t_x, \mathsf{fib} \mapsto t_x \to t_r] \;\vdash\; \underline{\mathsf{if}}\ \mathsf{n} < 1\ \underline{\mathsf{then}}\ 1\ \underline{\mathsf{else}}\ \mathsf{fib}\ (\mathsf{n} + -1) + \mathsf{fib}\ (\mathsf{n} + -2) \;:\; t_r$$

and

$$\rho\,[\mathsf{fib} \mapsto t_x \to t_r] \;\vdash\; \mathsf{fib} \;:\; \mathsf{int} \to \mathsf{int}$$

Proving them requires us to do two things: we must find appropriate types to fill in t_x and t_r, and we must construct derivation trees that demonstrate that these values are indeed valid types. By looking at the above statements, it is straightforward to see that both t_x and t_r need to be int, because n is compared to an integer and the result of the function is used in an addition. Thus, the second premise will be straightforward, requiring only a use of P3 to look up fib's type in the context.

To construct the derivation for the first premise, we use P7, which now requires us to show the following:

- $\rho\,[\mathsf{n} \mapsto \mathsf{int}, \mathsf{fib} \mapsto \mathsf{int} \to \mathsf{int}] \;\vdash\; \mathsf{n} < 1 \;:\; \mathsf{bool}$
- $\rho\,[\mathsf{n} \mapsto \mathsf{int}, \mathsf{fib} \mapsto \mathsf{int} \to \mathsf{int}] \;\vdash\; 1 \;:\; \mathsf{int}$
- $\rho\,[\mathsf{n} \mapsto \mathsf{int}, \mathsf{fib} \mapsto \mathsf{int} \to \mathsf{int}] \;\vdash\; \mathsf{fib}\ (\mathsf{n} + -1) + \mathsf{fib}\ (\mathsf{n} + -2) \;:\; \mathsf{int}$

We proceed with the derivations for these typings. In the interests of saving space, repeated copies of the same environment extensions have been replaced by "$\cdots$" when they were not directly relevant to the derivation and could be seen elsewhere. The first derivation checks that both arguments to the $<$ are integers, looking up n in the environment:

$$\mathrm{P5}\,\dfrac{\mathrm{P3}\,\dfrac{\rho\,[\mathsf{n} \mapsto \mathsf{int}, \mathsf{fib} \mapsto \mathsf{int} \to \mathsf{int}]\,(\mathsf{n}) = \mathsf{int}}{\rho\,[\mathsf{n} \mapsto \mathsf{int}, \mathsf{fib} \mapsto \mathsf{int} \to \mathsf{int}] \;\vdash\; \mathsf{n} \;:\; \mathsf{int}} \qquad \mathrm{P1}\,\dfrac{}{\rho\,[\cdots] \;\vdash\; 1 \;:\; \mathsf{int}}}{\rho\,[\mathsf{n} \mapsto \mathsf{int}, \mathsf{fib} \mapsto \mathsf{int} \to \mathsf{int}] \;\vdash\; \mathsf{n} < 1 \;:\; \mathsf{bool}}$$

The base case for the recursion is straightforward to check. We use P1, because it is an integer constant.

$$\mathrm{P1}\,\dfrac{}{\rho\,[\mathsf{n} \mapsto \mathsf{int}, \mathsf{fib} \mapsto \mathsf{int} \to \mathsf{int}] \;\vdash\; 1 \;:\; \mathsf{int}}$$

To check our recursive call, we need to check that both arguments to $+$ can have type int in order to use P4. Here, we see only the first one, as the second can be constructed by replacing all instances of -1 with -2.

$$\mathrm{P9}\,\dfrac{\mathrm{P3}\,\dfrac{\rho\,[\mathsf{n} \mapsto \mathsf{int}, \mathsf{fib} \mapsto \mathsf{int} \to \mathsf{int}]\,(\mathsf{fib}) = \mathsf{int} \to \mathsf{int}}{\rho\,[\mathsf{n} \mapsto \mathsf{int}, \mathsf{fib} \mapsto \mathsf{int} \to \mathsf{int}] \;\vdash\; \mathsf{fib} \;:\; \mathsf{int} \to \mathsf{int}} \qquad \mathrm{P4}\,\dfrac{\vdots \qquad\qquad \vdots}{\rho\,[\mathsf{n} \mapsto \mathsf{int}, \cdots] \;\vdash\; \mathsf{n} + -1 \;:\; \mathsf{int}}}{\rho\,[\mathsf{n} \mapsto \mathsf{int}, \mathsf{fib} \mapsto \mathsf{int} \to \mathsf{int}] \;\vdash\; \mathsf{fib}\ (\mathsf{n} + -1) \;:\; \mathsf{int}}$$

The full derivation can be seen in Figure 1.

5 Polymorphic let-bindings

In the preceding examples, we have defined variables and functions through let-bindings. In these examples, we've added new names to contexts and we've

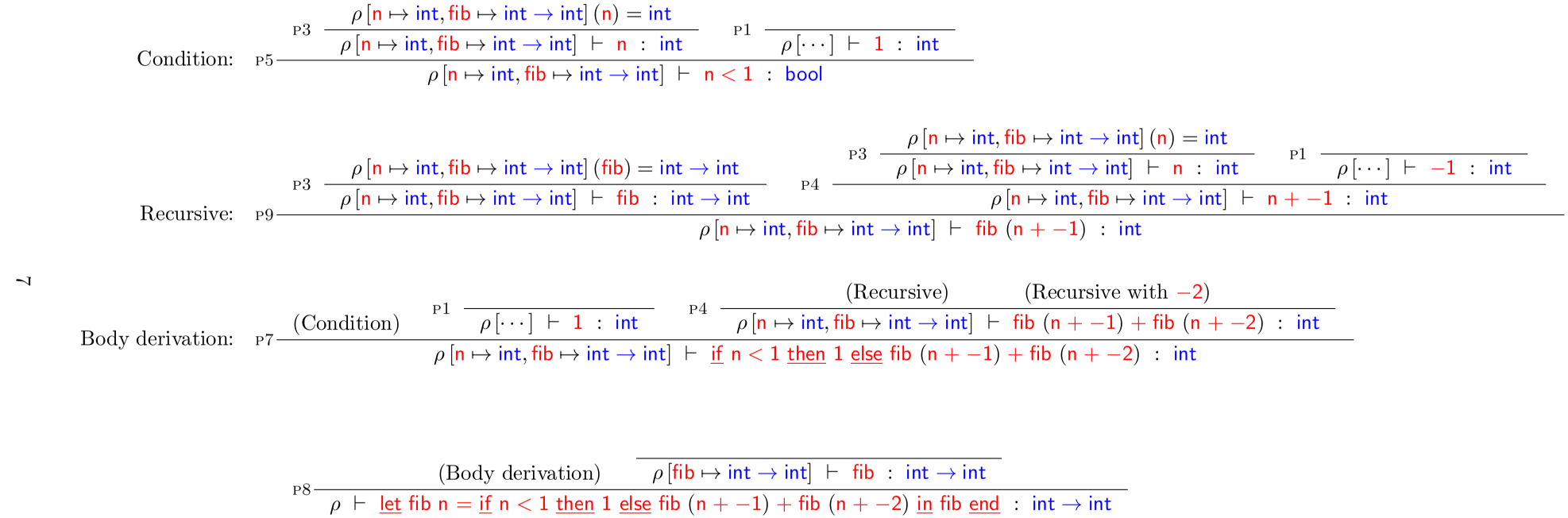

Figure 1: The full typing derivation for the Fibonacci function. Sub-derivations are named and written separately.

looked them up. However, we have not yet defined any polymorphic values or functions, so the substitution step in the lookup has been trivial. In this section, we'll explore how polymorphic functions are defined and how their type derivations are constructed. In doing so, we'll actually use the substitution feature.

5.1 The identity function

The identity function, written let id x = x in id end, can have any number of different types. Let's begin by showing that it can have the type int → int in any environment. We begin by using an unspecified environment ρ, our function, and our desired type:

$$\frac{?}{\rho \vdash \texttt{let id x = x in id end} : \texttt{int} \to \texttt{int}}$$

Applying P8, we get two new premises. The body of our function, in this case x, must be typable as some result type t_r, given that id can be a function from x's type to that result type. Additionally, if we then attach a type scheme quantifier over some type variables, we then need to show that the body has the correct type for the whole let (in this case int → int). Our tree is as follows:

$$\text{P8}\ \frac{\dfrac{?}{\rho\,[\texttt{id} \mapsto t_x \to t_r, \texttt{x} \mapsto t_x] \vdash \texttt{x} : t_r} \qquad \dfrac{?}{\rho\,[\texttt{id} \mapsto \forall \alpha_1, \ldots, \alpha_n \,.\, t_x \to t_r] \vdash \texttt{id} : \texttt{int} \to \texttt{int}}}{\rho \vdash \texttt{let id x = x in id end} : \texttt{int} \to \texttt{int}}$$

To complete the derivation, we must figure out how to fill out t_x, t_r, and $\alpha_1, \ldots, \alpha_n$ such that we can use P3 for both cases.

The first observation to be made is that t_x has to be the same as t_r. This is because id returns its argument unchanged. The other observation is that we have no other restrictions on them, because no operations are applied to x in id's definition. We are free to choose *any* type here. Thus, we can pick a type variable α. We now have:

$$\text{P8}\ \frac{\dfrac{?}{\rho\,[\texttt{id} \mapsto \alpha \to \alpha, \texttt{x} \mapsto \alpha] \vdash \texttt{x} : \alpha} \qquad \dfrac{?}{\rho\,[\texttt{id} \mapsto \forall \alpha_1, \ldots, \alpha_n \,.\, \alpha \to \alpha] \vdash \texttt{id} : \texttt{int} \to \texttt{int}}}{\rho \vdash \texttt{let id x = x in id end} : \texttt{int} \to \texttt{int}}$$

Secondly, we can replace the $\alpha_1, \ldots, \alpha_n$ with the single variable α, as it is the only type variable in the right-hand part of the type scheme. Because the variable α is now bound by the scheme, I change its color to green.

$$\text{P8}\ \frac{\dfrac{?}{\rho\,[\texttt{id} \mapsto \alpha \to \alpha, \texttt{x} \mapsto \alpha] \vdash \texttt{x} : \alpha} \qquad \dfrac{?}{\rho\,[\texttt{id} \mapsto \forall \alpha \,.\, \alpha \to \alpha] \vdash \texttt{id} : \texttt{int} \to \texttt{int}}}{\rho \vdash \texttt{let id x = x in id end} : \texttt{int} \to \texttt{int}}$$

We're almost done. What's left is to use P3 to look up our x and id in the environments. Here, it's important to look at the structure of P3 carefully: the $\alpha_1, \ldots, \alpha_n$ being substituted on the bottom are the αs between the $\forall$ and the dot, not just any arbitrary variables in the type. In this tutorial, the variables that can be substituted are always green.

$$\text{P8}\ \frac{\text{P3}\ \dfrac{\rho\,[\texttt{id} \mapsto \alpha \to \alpha, \texttt{x} \mapsto \alpha]\,(\texttt{x}) = \alpha}{\rho\,[\texttt{id} \mapsto \alpha \to \alpha, \texttt{x} \mapsto \alpha] \vdash \texttt{x} : \alpha} \qquad \dfrac{\rho\,[\texttt{id} \mapsto \forall \alpha \,.\, \alpha \to \alpha]\,(\texttt{id}) = \forall \alpha \,.\, \alpha \to \alpha}{\rho\,[\texttt{id} \mapsto \forall \alpha \,.\, \alpha \to \alpha] \vdash \texttt{id} : \texttt{int} \to \texttt{int}}}{\rho \vdash \texttt{let id x = x in id end} : \texttt{int} \to \texttt{int}}$$

The leftmost lookup performs no substitutions. The rightmost is justified because the $[\mathsf{int}/\alpha]\alpha \to \alpha$ from the bottom of P3 is equivalent to $\mathsf{int} \to \mathsf{int}$.

5.2 Applying id to itself

In the previous example, we used polymorphism to allow id to be used with integers. The same argument could have been used for any type - we would simply replace both ints with the new type in our derivation. However, polymorphism gives us more power than this. The same function can be used with *different* types in the same expression. For an example of this, we will apply the identity function to itself, producing an identity function on bools.

Formally speaking, we will show that:

$$\rho \vdash \underline{\mathsf{let}}\ \mathsf{id}\ \mathsf{x} = \mathsf{x}\ \underline{\mathsf{in}}\ \mathsf{id}\ \mathsf{id}\ \underline{\mathsf{end}} : \mathsf{bool} \to \mathsf{bool}$$

As usual, we place our goal at the bottom of a new tree. The outermost layer is a function let, so we use P8. The first premise is solved just as in the last example, using P3.

$$\text{P8}\ \dfrac{\text{P3}\ \dfrac{\rho\,[\mathsf{id} \mapsto \alpha \to \alpha, \mathsf{x} \mapsto \alpha]\,(\mathsf{x}) = \alpha}{\rho\,[\mathsf{id} \mapsto \alpha \to \alpha, \mathsf{x} \mapsto \alpha] \vdash \mathsf{x} : \alpha} \qquad \dfrac{?}{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha] \vdash \mathsf{id}\ \mathsf{id} : \mathsf{bool} \to \mathsf{bool}}}{\rho \vdash \underline{\mathsf{let}}\ \mathsf{id}\ \mathsf{x} = \mathsf{x}\ \underline{\mathsf{in}}\ \mathsf{id}\ \mathsf{id}\ \underline{\mathsf{end}} : \mathsf{bool} \to \mathsf{bool}}$$

Let's focus on the second premise of P8 to save space:

$$\dfrac{?}{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha] \vdash \mathsf{id}\ \mathsf{id} : \mathsf{bool} \to \mathsf{bool}}$$

Because it is a function call, we use P9:

$$\text{P9}\ \dfrac{\dfrac{?}{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha] \vdash \mathsf{id} : t_x \to t_r} \qquad \dfrac{?}{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha] \vdash \mathsf{id} : t_x}}{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha] \vdash \mathsf{id}\ \mathsf{id} : \mathsf{bool} \to \mathsf{bool}}$$

We need to demonstrate two different types for id! This is because one instance if id is the one that we will later use on our Booleans, while the other is actually being used on the $\mathsf{bool} \to \mathsf{bool}$ function id. Therefore, both t_x and t_r need to be $\mathsf{bool} \to \mathsf{bool}$. Splitting the premises in the interest of space, we get:

$$\dfrac{?}{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha] \vdash \mathsf{id} : (\mathsf{bool} \to \mathsf{bool}) \to (\mathsf{bool} \to \mathsf{bool})}$$

and

$$\dfrac{?}{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha] \vdash \mathsf{id} : \mathsf{bool} \to \mathsf{bool}}$$

The second premise is straightforward. We simply apply P3, substituting bool for α:

$$\text{P3}\ \dfrac{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha]\,(\mathsf{id}) = \forall\alpha\,.\,\alpha \to \alpha}{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha] \vdash \mathsf{id} : \mathsf{bool} \to \mathsf{bool}}$$

This is acceptable because $\mathsf{bool} \to \mathsf{bool} = [\mathsf{bool}/\alpha]\alpha \to \alpha$.

Proving first premise is almost the same – the only difference is our choice of substitution for α. Here, we perform the substitution $[\mathsf{bool} \to \mathsf{bool}/\alpha]\alpha \to \alpha$.

$$\text{P3}\ \dfrac{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha]\,(\mathsf{id}) = \forall\alpha\,.\,\alpha \to \alpha}{\rho\,[\mathsf{id} \mapsto \forall\alpha\,.\,\alpha \to \alpha] \vdash \mathsf{id} : (\mathsf{bool} \to \mathsf{bool}) \to (\mathsf{bool} \to \mathsf{bool})}$$

By instantiating our type schemes differently, we can get a variety of types for the same term.

A Formal rules

Types

t	::=		*Types*
	|	int	*Integers*
	|	bool	*Booleans*
	|	$t \to t$	*Functions*
	|	$\alpha \mid \beta \mid \gamma \mid \delta \mid \ldots$	*Type variables*

Type schemes

σ	::=		*Typing schemes*
	|	t	*Simple type with no parameters*
	|	$\forall \alpha_1, \ldots, \alpha_n \,.\, t$	*t with $\alpha_1, \ldots, \alpha_n$ as parameters*

Expressions

e	::=		*Expressions*
	|	$x \mid y \mid z \mid f$	*Variables*
	|	i	*Integer literals*
	|	$e + e$	*Addition*
	|	$e < e$	*Less than*
	|	true	*True*
	|	false	*False*
	|	$e\ e$	*Function application*
	|	if e then e else e	*Conditional expression*
	|	let $x = e$ in e end	*Variable binding*
	|	let $x\ x = e$ in e end	*Variable binding*

Contexts

ρ	::=		*Typing contexts*
	|	$\cdot$	*The empty context*
	|	$\rho\,[x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n]$	*ρ extended with new name/scheme pairs*

The symbol $\mapsto$ is pronounced "maps to".

Typing Rules

These typing rules are taken from Sestoft [2012, p. 98], and have been slightly reformatted to fit the page and match the color scheme of the tutorial.

$$\text{P1}\ \frac{}{\rho \vdash i : \text{int}}$$

$$\text{P2}\ \frac{}{\rho \vdash b : \mathsf{bool}}$$

$$\text{P3}\ \frac{\rho(\mathsf{f}) = \forall \alpha_1 \,.\, t}{\rho \vdash \mathsf{f} : [t_1/\alpha_1, \ldots, t_n/\alpha_n]t}$$

$$\text{P4}\ \frac{\rho \vdash e_1 : \mathsf{int} \quad \rho \vdash e_2 : \mathsf{int}}{\rho \vdash e_1 + e_2 : \mathsf{int}}$$

$$\text{P5}\ \frac{\rho \vdash e_1 : \mathsf{int} \quad \rho \vdash e_2 : \mathsf{int}}{\rho \vdash e_1 < e_2 : \mathsf{bool}}$$

$$\text{P6}\ \frac{\rho \vdash e_r : t_r \quad \rho[x \mapsto \forall \alpha_1, \ldots, \alpha_n \,.\, t_r] \vdash e_b : t \quad \alpha_1, \ldots, \alpha_n \text{ not free in } \rho}{\rho \vdash \underline{\mathsf{let}}\ x = e_r\ \underline{\mathsf{in}}\ e_b\ \underline{\mathsf{end}} : t}$$

$$\text{P7}\ \frac{\rho \vdash e_1 : \mathsf{bool} \quad \rho \vdash e_2 : t \quad \rho \vdash e_3 : t}{\rho \vdash \underline{\mathsf{if}}\ e_1\ \underline{\mathsf{then}}\ e_2\ \underline{\mathsf{else}}\ e_3 : t}$$

$$\text{P8}\ \frac{\rho[x \mapsto t_x, f \mapsto t_x \to t_r] \vdash e_r : t_r \quad \rho[f \mapsto \forall \alpha_1, \ldots, \alpha_n \,.\, t_x \to t_r] \vdash e_b : t \quad \alpha_1, \ldots, \alpha_n \text{ not free in } \rho}{\rho \vdash \underline{\mathsf{let}}\ f\ x = e_r\ \underline{\mathsf{in}}\ e_b\ \underline{\mathsf{end}} : t}$$

$$\text{P9}\ \frac{\rho \vdash e_1 : t_x \to t_r \quad \rho \vdash e_2 : t_x}{\rho \vdash e_1\ e_2 : t_r}$$

Acknowledgments

I would like to thank Hannes Mehnert and Peter Sestoft for comments on drafts of this tutorial as well as David Georg Korczynski for catching an error.

References

Peter Sestoft. *Programming Language Concepts*. Springer, 2012.