
CS 421 – Spring 2007

Lecture Notes Set 13:

User Defined Recursive Types

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 07 - typederiv (slides 48-end)

Made available: February 19, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Type Variables

Last time, we learned about typing rules and how to prove type derivations. Today we will cover a few more topics on type derivations.

2.1 if_then_else Rule (slide 48)

Last time, we were able to look at many of the axioms that one uses in type derivations. If there was an expression with the arithmetic operators, you knew that each side of the operator had to have an integer as the type. If you were looking at logical *and* you knew you were looking for something of type boolean on both sides. However, what if you don't know exactly what is going to be returned.

This uses the notion of type variables. The type can be almost anything. So to allow for that, we use type variables, a variable, representing some type. It stands in for any type that we can use in these positions. This type variable is a meta-variable.

But what do you see, if there is the if_then_else statement. Here, we will look at the rule:

$$\frac{\Gamma \mid - e_1 : \text{bool} \quad \Gamma \mid - e_2 : \tau \quad \Gamma \mid - e_3 : \tau}{\Gamma \mid - (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

In this expression, the if_then_else statement allows for something other types to be returned, as long as the then case and the else case return the same type. We don't know what it is exactly here, but we do know that it has to be the same in both cases. So everywhere τ is written in the rule, that type will all be the same. They could all be integers, booleans, strings, what matters is that they are the same.

2.2 Function Application Rule (slides 49-50)

$$\frac{\Gamma \mid - e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \mid - e_2 : \tau_1}{\Gamma \mid - (e_1 e_2) : \tau_2}$$

First of all, let's look at the notation. Under the line there is e_1 and e_2 . e_1 is an expression and e_2 has to be some type that is appropriate to be an argument to e_1 . So e_2 is the input for the expression e_1 .

Now let's say that you are writing a function. There are a few restraints while doing this. First of all, there has to be a function, which is signified by the arrow. Next, what you need to make sure is that the result of the function is the same as the output of the application. That is the type of what is under the line is the same as the what comes after the

¹© 2007, Share and Enjoy

arrow. In this case, they are both τ_2 , which means that so far, we're ok. The other constraint is that e_2 has to have the same type as the input of e_1 . In this case, this is true because e_2 is of type τ_1 and e_1 takes in something of type τ_1

Now there are many possibilities for what the actual code might be for these. e_1 could be a huge function that has had numerous arguments applied to it already and is just waiting for the last one of e_2 . It could be that e_1 is wait for an input function and e_2 is that function, make τ_1 be a function type. It all depends on what our types are and what our expressions are. All we know here is that we have some expression that when you evaluate will be a function and we have some other argument that when we evaluate it, it has the type that we are insisting it is. And so what ever that argument is of that right type to be passed into the function.

Notice, also, that we have a type variable, τ_1 , in our assumptions (the part above the line) about the main expression (under the line) that does not appear with our main expression. How do we know there is a τ_1 . It make this more clear, let's take a simple example. Let's say there we have an expression, f applied to x , ($f x$), and that expression results in an integer. What is the type of f ? We know that it is something to int . But for now, we will simply be ok with not knowing. Later, it may be the case that we can look up the type of f in our environment, but for now, we will be ok with not knowing.

Let's look at some other examples:

$$\frac{\Gamma \mid - \text{print_int} : \text{int} \rightarrow \text{unit} \quad \Gamma \mid - 5 : \text{int}}{\Gamma \mid - (\text{print_int } 5) : \text{unit}}$$

$e_1 = \text{print_int}$
 $e_2 = 5$
 $\tau_1 = \text{int}$
 $\tau_2 = \text{unit}$

In this first example, we want to prove that $\text{print_int } 5$ has type unit . To do this we need to show that 5 is an integer and that print_int is a function that takes an integer and returns something of type unit . There is more to this proof tree, but this is the basic idea.

$$\frac{\Gamma \mid - \text{map } \text{print_int} : \text{int list} \rightarrow \text{unit list} \quad \Gamma \mid - [3;7] : \text{int list}}{\Gamma \mid - (\text{map } \text{print_int } [3;7]) : \text{unit list}}$$

$e_1 = \text{map } \text{print_int}$
 $e_2 = [3;7]$
 $\tau_1 = \text{int list}$
 $\tau_2 = \text{unit list}$

In this example, we want to prove that $\text{map } \text{print_int } [3;7]$ has type unit list . The first thing we do is look at the structure on the bottom and see how to start breaking the expression up into smaller parts. The first this we see is that $\text{map } \text{print_int}$ is a function with $[3;7]$ as its argument. So, we need to show that $[3;7]$ is an integer list and that $\text{map } \text{print_int}$ is a function that takes an integer list and returns something of type unit list . We we were to take this farther, we would then break map and print_int into smaller parts and work from there.

2.3 Functions Rule (slides 51-52)

We have been talking about applying functions to arguments...but we haven't yet discussed how to know of something is actually a function.

$$\frac{[x : \tau_1] \cup \Gamma \mid - e : \tau_2}{\Gamma \mid - \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

What the function rule is going to boil down to is basically $\text{fun } x$ goes to... Here, the x should be treated as a mathematical variable for any identifier. First thing that we know about functions is that...well, it's a function, it takes some input and gives some output. So under the line, that is the type we have. We haave some function that takes in something of type τ_1 and outputs something of type τ_2 . To check that this is true, we need to check to see that the body of the function, which in the example is e , evaluates to something of type τ_2 . We also allowed to assume that our input x to the function, has the type τ_1 and we add that assumption to our environment. That x will hide any other x 's in Γ at that point. So, in the enviroment below the line, we don't know what type x has. But above the line, we have just stated that x has type τ_1 .

Let's look at some examples for this rule:

$$\frac{[y : \text{int}] \cup \Gamma \mid - y + 3 : \text{int}}{\Gamma \mid - \text{fun } y \rightarrow y + 3 : \text{int} \rightarrow \text{int}}$$

In this example, since our new environment has that y is of type int , all we have to show is that $y + 3$ has the type int .

$$\frac{[f : \text{int} \rightarrow \text{bool}] \cup \Gamma \mid - f\ 2 :: [\text{true}] : \text{bool list}}{\Gamma \mid - (\text{fun } f \rightarrow f\ 2 :: [\text{true}]) : (\text{int} \rightarrow \text{bool}) \text{ bool list.}}$$

This example is saying that we have a function that takes in a function f and applies 2 to the function f and then cons-es on that to a list. That in the end will return a boolean list. Function f is has to be a function that takes integer to booleans. So we overall, have a function that takes a function (that takes integers to booleans) and returns an boolean list.

2.4 Let and Let Rec Rules (slide 53)

The let rule:

$$\frac{\Gamma \mid - e_1 : \tau_1 \quad [x : \tau_1] \cup \Gamma \mid - e_2 : \tau_2}{(\text{let } x = e_1 \text{ in } e_2 : \tau_2)}$$

The let rec rule:

$$\frac{[x : \tau_1] \cup \Gamma \mid - e_1 : \tau_1 \quad [x : \tau_1] \cup \Gamma \mid - e_2 : \tau_2}{(\text{let rec } x = e_1 \text{ in } e_2 : \tau_2)}$$

What is the difference here? The difference is that in `let rec`, the function is allowed to be recursive. That is, the functions name is allowed to exist in its environment. But more generally, the identifiers name is allowed to be in the body of its definition.

So, let us use the example, `let x = x + 3`. For this to make sense, what do we know about the x ? We know that x had to previously exist. We could even say, `let x = x::[]`. Meaning the new x is a list, but the old x was some element being cons-ed onto a list. But if we said `let rec x = x::[]`. Now there is a problem. The problem is that x can't be an element and a list (remember that `cons` takes an element in front and a list after it), it has to be one or the other.

So in the type proof trees above, for the `let` rule, we want to show that the `let x = e1 in e2` has type τ_2 . To do that, the type τ_2 is going to come from e_2 . so those types, the type of the `let x` under the line and e_2 have to be the same. Now since we are saying `let x equal something in e2`, we expect to see x in e_2 somewhere, so we have to make sure that x is in the environment for e_2 , which explains the union to the environment. That *something* that has x in it and is used by e_2 , we called e_1 , and e_1 has to have some type, which we will say is τ_1 . So when x is used in e_2 , it must also have type τ_1 there as well.

Note that these are typing rules for expressions, because they use the word *in*. Without the *in*, they are considered declarations and do not have types, they give an increment to the environment, a new binding that the environment holds.

So, with `let rec x`, it's the same as the `let` rule, except that the type of x is also in its own environment.

2.5 Polymorphism in OCaml (slide 54)

Now, the `let` rule is where true polymorphism could come in, if we were to allow polymorphism. But the τ 's that we have been using are mathematical variables, not exactly type variables.

But these typing rules are not flexible enough to allow polymorphism. For polymorphism, we need to have a true type variable, not a mathematical variable, but a type variable and you also need a the notion of a bound type variable and the notion of instansiating a type variable. What we are doing is a more limited version of type derivation than what OCaml can actually handle.

3 An Example

3.1 A little more complicated than last time... (slides 55-66)

$$\vdash (\text{let rec one} = 1::\text{one in let } x = 2 \text{ in fun } y \rightarrow (x::y::\text{one})) : \text{int} \rightarrow \text{int list}$$

What do we do first?

Well, on the outer most level, we have the let rec rule. So we make our e_1 and our e_2 . And in doing this, we are saying that *one* being of int list is sufficient to show that e_1 and e_2 have types int list and $\text{int} \rightarrow \text{int list}$ respectively.

$$\frac{[\text{one} : \text{int list}] \vdash (1::\text{one}) : \text{int list} \quad [\text{one} : \text{int list}] \vdash (\text{let } x = 2 \text{ in fun } y \rightarrow (x::y::\text{one})) : \text{int} \rightarrow \text{int list}}{\vdash (\text{let rec one} = 1::\text{one in let } x = 2 \text{ in fun } y \rightarrow (x::y::\text{one})) : \text{int} \rightarrow \text{int list}}$$

Now we have two things to prove, so let's look at the left side first. What rule can be applied here? Now, since we don't have typing rules for constructors, we are going to try consider constructors as functions.

$$[\text{one} : \text{int list}] \vdash (1::\text{one}) : \text{int list}$$

So, since we are applying the 'function' cons, this is considered a function application. So using that rule, we get:

$$\frac{[\text{one} : \text{int list}] \vdash ((::) 1) : \text{int list} \rightarrow \text{int list} \quad [\text{one} : \text{int list}] \vdash \text{one} : \text{int list}}{[\text{one} : \text{int list}] \vdash (1::\text{one}) : \text{int list}}$$

Now, for the right, we use the variable rule to show that *one* is and int list and we see we are right, since it is in the environment. Then we know that type can be placed on the left side, with the cons 'function'. So now we have to finish proving the type for the left side of the proof tree above. We get this:

$$\frac{[\text{one} : \text{int list}] \vdash ((::)) : \text{int} \rightarrow \text{int list} \rightarrow \text{int list} \quad [\text{one} : \text{int list}] \vdash 1 : \text{int}}{[\text{one} : \text{int list}] \vdash ((::) 1) : \text{int list} \rightarrow \text{int list}}$$

To get the type for the 1 (the right side) we use the constant rule. And then we are done with that. For the left side, we get the type int from the right, and for the rest of the type, we would need a type table to do a look up for the cons 'function', so that is done.

Now, we finished the e_1 side of the original proof. What we have next is a the let rule. Which can be followed from the slides.