

---

# CS 421 – Spring 2007

## Lecture Notes Set 14:

### Type Inference

Elsa L. Gunter<sup>1</sup>

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 08 - typeinfer

Made available: February 21, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 Overview

### 2.1 The Problem (slide 2)

Previously, we discussed that given some expression  $e$ , and an environment  $\Gamma$ , could we prove the judgement that  $e$  had some type  $\tau$ . That was type derivation. But here, the question is, given an expression, does there exist a type such that we can start to build a derivation.

For variables, we could look up the type in  $\Gamma$ . For constants, we assumed we knew what the type of a constant was. For `if_then_else` expressions, we knew the `if` part had to give a `bool` and then the `then_else` part had to give the same type as the whole expression. In each of those cases, what went above the line was completely filled in with stuff that was received from below the line.

But the function application rule...you don't get to check your types with what's below the line... You have something that looks like:

$$\frac{\Gamma \mid - e_1 : \sigma \rightarrow \tau \quad \Gamma \mid - e_2 : \sigma}{\Gamma \mid - (e_1 e_2) : \tau}$$

Here, we have some function  $e_1$  with the argument  $e_2$  being applied to it. We know that overall, we will get back something of type  $\tau$ . But, above the line, when we split up the function application to do find the specific types, we don't know what exactly the function  $e_1$  takes. All we know is that input has to be the same as the type of  $e_2$ . But we don't know the type of  $e_2$  either. We can't get it from  $\Gamma$ , nor can we find the type below the line... Above, we put the placeholder,  $\sigma$ ...but that is just a guess, we had to infer what the type was.

### 2.2 Outline (slide 3)

We will start the process of type inference with the introduction of meta-variables. These variables are basically names for our types, they will help us when we don't know what the actual type of an expression is. Next we will be constantly looking at the outermost expression, trying to figure out, what it is and what type it has and how we can decompose it into its various components. For example, is it a function, an application, and `if_then_else`, etc. As we do this, we may need to add in new type variables. As we generate these variables, we will also be generating various constraints for the components and the expression as a whole. For example, in the function application rule, the input type of  $e_1$  had to be the same as the type of  $e_2$ , so things like that being to constrain, what happens next. So, you

---

<sup>1</sup>© 2007, Share and Enjoy

keep doing this decomposition of terms until you are left with a system of equations, or constraints. You then use substitution from there to get your answer, here, this is called unification.

We need to break this into two parts. First we need to create the type derivation skeleton. And the next part is the solving the system of equations that we got from that skeleton.

### 3 An example (slides 4 - 16)

Suppose we want to find the type for the following:

$$\text{fun } x \rightarrow \text{fun } f \rightarrow f \ x$$

We want to try and infer a type for this. We will start with the empty environment. Also, we will give that expression an unknown type, which we will call  $\alpha$ . That looks like:

$$[] \mid - (\text{fun } x \rightarrow \text{fun } f \rightarrow f \ x) : \alpha$$

Well, first and foremost, we have a function, so anything we do, will end with the rule for functions. So we introduce  $x$  into the environment and we will give it some type. We don't know what type that is, so we will infer that it is of type  $\beta$ . Then we take the rest of the expression and give that some type, which will be called  $\gamma$ . After that, we will require that  $\alpha$  is equal to  $\beta$  goes to  $\gamma$ . That is,  $\alpha$  is a function type that takes some type  $\beta$  to some type  $\gamma$ .

$$\frac{[x : \beta] \mid - (\text{fun } f \rightarrow f \ x) : \gamma}{[] \mid - (\text{fun } x \rightarrow \text{fun } f \rightarrow f \ x) : \alpha}$$

$$\alpha \equiv (\beta \rightarrow \gamma)$$

Now, on the outermost level, we have another function. So we are going to do the same thing again, by applying the function rule. We introduce  $f$  to the environment now, and give it some type, in this case  $\delta$ . Then we take the rest of this expression and give it some type, in this case,  $\epsilon$ . Then, we will require that  $\gamma$  is equal to  $\delta$  goes to  $\epsilon$ . That is,  $\gamma$  is a function type that takes some type  $\delta$  to some type  $\epsilon$ . Because of this new constraint, we now picked up an additional constraint on  $\alpha$  as well, since it contains  $\gamma$ .

$$\frac{\frac{[f : \delta; x : \beta] \mid - (f \ x) : \epsilon}{[x : \beta] \mid - (\text{fun } f \rightarrow f \ x) : \gamma}}{[] \mid - (\text{fun } x \rightarrow \text{fun } f \rightarrow f \ x) : \alpha}$$

$$\alpha \equiv (\beta \rightarrow \gamma); \gamma \equiv (\delta \rightarrow \epsilon)$$

Now, at the outermost level, we have a function application. Here, we finally can bring something up from below the line, with the  $\epsilon$ . In the third level of the tree, we have that  $(f \ x)$  outputs something of type  $\epsilon$ . That means the function  $f$  outputs something of type  $\epsilon$ . So, in the top level of the tree, that is the output type. But we don't know what the input type is exactly, so we make up a new type variable,  $\phi$  and we now have the constraint that what ever is the input type for  $f$  is, it has to be the same as the type of  $x$ .

$$\frac{\frac{[f : \delta; x : \beta] \mid - f : \phi \rightarrow \epsilon \quad [f : \delta; x : \beta] \mid - x : \phi}{[f : \delta; x : \beta] \mid - (f \ x) : \epsilon}}{[x : \beta] \mid - (\text{fun } f \rightarrow f \ x) : \gamma}}{[] \mid - (\text{fun } x \rightarrow \text{fun } f \rightarrow f \ x) : \alpha}$$

$$\alpha \equiv (\beta \rightarrow \gamma); \gamma \equiv (\delta \rightarrow \epsilon)$$

We're almost done now. We have to do the variable rule on  $f$  and the variable rule on  $x$ . So, for  $f$ , we have to check the environment. We see that  $f$  has type  $\delta$  in the environment...but, uh oh..., we gave the expression type  $\phi \rightarrow \epsilon$ ... That just means another equation. So we add to the bottom, that  $\delta$  is equivalent to  $\phi \rightarrow \epsilon$ . Then for the  $x$  we use the variable and have the same issue. In the expression  $x$  has type  $\phi$ , but in the environment,  $x$  has type  $\beta$ . That just means, we again, add another equation that  $\phi$  is equivalent to  $\beta$ . So we get:

$$\frac{\frac{\frac{[f : \delta; x : \beta] \mid - f : \phi \rightarrow \epsilon \quad [f : \delta; x : \beta] \mid - x : \phi}{[f : \delta; x : \beta] \mid - (f \ x) : \epsilon}}{[x : \beta] \mid - (\text{fun } f \rightarrow f \ x) : \gamma}}{[] \mid - (\text{fun } x \rightarrow \text{fun } f \rightarrow f \ x) : \alpha}}$$

$$\alpha \equiv (\beta \rightarrow \gamma); \gamma \equiv (\delta \rightarrow \epsilon); \delta \equiv (\phi \rightarrow \epsilon); \phi \equiv \beta$$

So, if we can fill in types for  $\alpha$ ,  $\gamma$ ,  $\delta$ , and  $\phi$ , such that the equations all hold, then we're done. Solving these equations involve unification, which we will explain in more detail in the next lecture, but for now, we will simply talk

about substitution. So, we will start at the end of the equation list. So, for any place we see  $\phi$ , we will place  $\beta$  there. So then we get:

$$\frac{\frac{\frac{[f : \delta; x : \beta] \mid - f : \beta \rightarrow \epsilon \quad [f : \delta; x : \beta] \mid - x : \beta}{[f : \delta; x : \beta] \mid - (f x) : \epsilon}}{[x : \beta] \mid - (\text{fun } f \rightarrow f x) : \gamma}}{[] \mid - (\text{fun } x \rightarrow \text{fun } f \rightarrow f x) : \alpha}}{\alpha \equiv (\beta \rightarrow \gamma); \gamma \equiv (\delta \rightarrow \epsilon); \delta \equiv (\beta \rightarrow \epsilon); \phi \equiv \beta}$$

Now, that we eliminated  $\phi$ , we will now, substitute in for  $\delta$  and get:

$$\frac{\frac{\frac{[f : (\beta \rightarrow \epsilon) x : \beta] \mid - f : \beta \rightarrow \epsilon \quad [f : (\beta \rightarrow \epsilon); x : \beta] \mid - x : \beta}{[f : (\beta \rightarrow \epsilon); x : \beta] \mid - (f x) : \epsilon}}{[x : \beta] \mid - (\text{fun } f \rightarrow f x) : \gamma}}{[] \mid - (\text{fun } x \rightarrow \text{fun } f \rightarrow f x) : \alpha}}{\alpha \equiv (\beta \rightarrow \gamma); \gamma \equiv ((\beta \rightarrow \epsilon) \rightarrow \epsilon); \delta \equiv (\beta \rightarrow \epsilon)}$$

Now, substitute in for  $\gamma$  and get:

$$\frac{\frac{\frac{[f : (\beta \rightarrow \epsilon) x : \beta] \mid - f : \beta \rightarrow \epsilon \quad [f : (\beta \rightarrow \epsilon); x : \beta] \mid - x : \beta}{[f : (\beta \rightarrow \epsilon); x : \beta] \mid - (f x) : \epsilon}}{[x : \beta] \mid - (\text{fun } f \rightarrow f x) : ((\beta \rightarrow \epsilon) \rightarrow \epsilon)}}{[] \mid - (\text{fun } x \rightarrow \text{fun } f \rightarrow f x) : \alpha}}{\alpha \equiv (\beta \rightarrow ((\beta \rightarrow \epsilon) \rightarrow \epsilon)); \gamma \equiv ((\beta \rightarrow \epsilon) \rightarrow \epsilon)}$$

Finally, we can substitute in for  $\alpha$  and get:

$$\frac{\frac{\frac{[f : (\beta \rightarrow \epsilon) x : \beta] \mid - f : \beta \rightarrow \epsilon \quad [f : (\beta \rightarrow \epsilon); x : \beta] \mid - x : \beta}{[f : (\beta \rightarrow \epsilon); x : \beta] \mid - (f x) : \epsilon}}{[x : \beta] \mid - (\text{fun } f \rightarrow f x) : ((\beta \rightarrow \epsilon) \rightarrow \epsilon)}}{[] \mid - (\text{fun } x \rightarrow \text{fun } f \rightarrow f x) : (\beta \rightarrow ((\beta \rightarrow \epsilon) \rightarrow \epsilon))}}{\alpha \equiv (\beta \rightarrow ((\beta \rightarrow \epsilon) \rightarrow \epsilon))}$$

Now we have a completely valid proof tree. What that means is that you can choose any types for  $\beta$  and  $\epsilon$  and you will have a valid type derivation.

## 4 The Algorithm (slides 18 - 24)

Now we will discuss the algorithm, a little more formally, that tells whether there exists a type that fits.

We will break this in to a few pieces. First we will have a function, `has_type` that returns the system of equations that we derived in the example that we used for the substitution. The input is an environment, an expression, and a type, it will return the system of equations ( $S$ ) such that the input has a proof derivation if and only if there's a solution to the system of equations. Once we have  $S$ , we will call the unification algorithm on it to solve for the variables and do the substitution.

So, how do we write `has_type`. Well, there are a few different cases. If the expression was simply a variable, then the variable will have the same type in the environment as was the input type, since the input type was associated with the expression.

$$\text{Var } v \text{ return } \tau \equiv \Gamma(v)$$

If the expression is a constant, then we return the type that constant has. That is, if the constant is the number 5, we return `int`, if the constant is `true`, we return `bool`, etc.

$$\text{Const } c \rightarrow \text{return } \{\tau \equiv \sigma\} \text{ where } \Gamma \mid - c : \sigma \text{ by the rule for constants}$$

If the expression is a function, we have to introduce two new variables, we'll call them  $\alpha$  and  $\beta$  and say that the input of the function goes into the environment and gets type  $\alpha$ . The expression gets type  $\beta$  we recursively call `has_type` on that new set of information and then we return the set of equations we have made so far with the additional constraint that our original input type of  $\tau$  has type  $\alpha \rightarrow \beta$ .

```

fun x → exp →
  let  $\alpha, \beta$  be new variables
  let  $S = \text{has\_type}([x:\alpha] \cup \Gamma, e, \beta)$ 
  Return  $\{\tau \equiv \alpha \rightarrow \beta\} \cup S$ 

```

In the case that the expression is a function application, then we just need one new variable,  $\alpha$ . We are going to have two recursive calls to `has_type`. One in which we call it on the function, of the function application and the other call will be with the input of the function. Then, we will take the union of the equation sets, they return.

```

App ( $e_1, e_2$ ) →
  let  $\alpha$  be a new variable
  let  $S_1 = \text{has\_type}(\Gamma, e_1, \alpha \rightarrow \tau)$ 
  let  $S_2 = \text{has\_type}(\Gamma, e_2, \alpha)$ 
  Return  $S_1 \cup S_2$ 

```

For the `if_then_else` rule, we know that  $e_1$  and  $e_2$  have to have the same type. More specifically, we know that they have to be of type  $\tau$  because when we called `has_type` on this expression, we said also inputted that the expression had type  $\tau$ . So we called `has_type` on the two inner expressions. We also have to make sure that  $b$  has type `bool`. If all calls work out, then the `if_then_else` is ok.

```

if b then  $e_1$  else  $e_2$  →
  let  $S_1 = \text{has\_type}(\Gamma, e_1, \tau)$ 
  let  $S_2 = \text{has\_type}(\Gamma, e_2, \tau)$ 
  let  $S_3 = \text{has\_type}(\Gamma, b, \text{bool})$ 
  Return  $S_1 \cup S_2 \cup S_3$ 

```

Let's look at the `let` rule now. This is like the merging of a function and an application. The  $x$  in the `let x` needs some type, that type will be  $\alpha$  in this case. Our overall expression has type  $\tau$ . So we add into the environment that  $x$  has type  $\alpha$  and then we call `has_type` on that new environment,  $e_2$ , since  $e_2$  is the main expression with type  $\tau$ . Since it is the main expression that has that type. But we also need to make sure that  $e_1$  has the the same type that we assigned to  $x$ . So, we call `has_type` on  $e_1$  with the type  $\alpha$ . Then return the union of those two sets of equations.

```

Let  $x = e_1$  in  $e_2$ ) →
  let  $\alpha$  be a new variable
  let  $S_1 = \text{has\_type}(\Gamma, e_1, \alpha)$ 
  let  $S_2 = \text{has\_type}([x: \alpha] \cup \Gamma, e_2, \tau)$ 
  Return  $S_1 \cup S_2$ 

```

Now look at the `let rec` rule. Again, the  $x$  in the `let rec x` needs some type, that type will be  $\alpha$ . Our overall expression has type  $\tau$ . So we add into the environment that  $x$  has type  $\alpha$  and then we call `has_type` on that new environment,  $e_2$ , since  $e_2$  is the main expression with type  $\tau$ . Since it is the main expression that has that type. But we also need to make sure that  $e_1$  has the the same type that we assigned to  $x$ . So, we call `has_type` on  $e_1$  with the type  $\alpha$  but since it is recursive,  $x$  having type  $\alpha$  has to appear in the environment as well. So adding that in there is the only change that we will make from the `let` case. Then return the union of those two sets of equations.

```

Let rec  $x = e_1$  in  $e_2$ ) →
  let  $\alpha$  be a new variable
  let  $S_1 = \text{has\_type}([x: \alpha] \cup \Gamma, e_1, \alpha)$ 
  let  $S_2 = \text{has\_type}([x: \alpha] \cup \Gamma, e_2, \tau)$ 
  Return  $S_1 \cup S_2$ 

```

There are also the cases for arithmetic operations, but those can technically be treated as function applications as well. Basically, what we have done so far is formalized all the axioms, the rules, that we have seen when doing type derivations.

Now, to start off, we need to start with a type somewhere. So, we will introduce the function `type_of`. It will only take the environment and the expression and in the function we will call `has_type` with the input of `type_of` and also with our initial type guess of some  $\alpha$ . We will get a systems of equations, and attempt to unify it. If we can do that, then we know our guess of  $\alpha$  is satisfiable. This looks like:

```
type_of( $\Gamma$ , e) =  
  Let S has_type( $\Gamma$ , e,  $\alpha$ )  
  Return Unif(S)( $\alpha$ )
```

Note, that any time you introduce a new variable, make sure that specific variable has never been used so far. Even though we continuously used  $\alpha$  for each case, we meant some different  $\alpha$  each time.

Next time, we will go through the unify algorithm.