# Today's Lecture Notes for cs173

sk and dbtucker

October 23, 2000

## Magical polymorphism

Yesterday we saw one way to implement parametric polymorphism. The big problem was that you had to write a lot of code—every time you wanted to use a polymorphic function, you had to specify any parameterized types before invoking the function.

Let's say we're no longer going to annotate function arguments with their types, and instead we'll have the type checker somehow figure out those types. Previously, our type checker consumed an expression and returned true or false depending on whether there were any type errors. Now, the type checker consumes an expression and returns a type it has inferred for that expression.

Since we don't annotate arguments with their types, there's no point in specifying the types a function is parameterized over (using $\Lambda$). So, the type checker has to determine whether a function is polymorphic, and then ensure that it is applied legally based on its polymorphic type.

To infer these function types, we need do to two steps. First, we'll label each subexpression with a type, and then develop equations relating these types. Second, we'll solve the set of equations to determine the type of the function. For now, we'll assume we have some sort of "magic" that will perform the second step for us.

We'll look at a number of examples where we don't specify the type of a function's parameter, and derive the correct type of the function.

### Example 1

$(\lambda\ (x : _)\ x)$

This is the identity function. For any value of type $\tau$, it simply returns that value of type $\tau$. So, we expect this function to have type $\tau \rightarrow \tau$. Now let's derive this using our wits, with a little help from magic. We'll write down types for subexpressions, then relate those types with equations.

The first subexpression to label is the entire function. Let's say it has some type $\alpha$, where $\alpha$ is an unconstrained variable—it could be *bool*, *num*, *bool* $\rightarrow$ *num*, or any other of your favorite types.

Since the entire expression is a function, we can assume it has some sort of arrow type. So we write down this equation:

$$\alpha = \beta \rightarrow \gamma$$

where $\beta$ and $\gamma$ are new unconstrained types. The argument to the function, $x$, must be of type $\beta$:

$$\text{type}(x) = \beta$$

Say the body of the function has some type $\delta$. The body is just $x$, so:

$$\text{type}(x) = \delta$$

Finally, we note that the return type of the function is the same as the type of the body:

$$\gamma = \delta$$

Here's the complete list of equations:

$$\alpha = \beta \rightarrow \gamma$$

$$\text{type}(x) = \beta$$

$$\text{type}(x) = \delta$$

$$\gamma = \delta$$

Now we invoke magic to solve these equations, and we see that $\alpha = \beta \rightarrow \beta$.

## Example 2

$(\lambda\ (x : _)\ (+\ x\ 10))$

Since the + function operates on *num*s, $x$ must be a *num*. The + function also returns a *num*, so our function above has type $num \rightarrow num$. Now let's derive this type using the same method as before. First, we say the function has type $\alpha$:

$$\alpha = \beta \rightarrow \gamma$$

$$\text{type}(x) = \beta$$

Let's say the body has type $\delta$, so:

$$\delta = \text{rng}(\text{type}(+))$$

where rng means the range (return type) of a function ($\text{rng}(\omega \rightarrow \tau) = \tau$). We know the type of +, so:

$$\delta = \text{rng}((num \times num) \rightarrow num) = num$$

The type of + also requires that its two arguments have type *num*:

$$\text{type}(x) = num$$

$$\text{type}(10) = num$$

Since $\text{type}(x) = \beta$ and $\text{type}(x) = num$:

$$\beta = num$$

The return type of the function ($\gamma$) is the same as the type of the body:

$$\gamma = \delta = num$$

Now we know the type of the function:

$$\alpha = num \rightarrow num$$

## Example 3

$(\lambda\ (x : _)\ (x\ 3))$

The argument $x$ is applied to 3, so it must be a function which takes a *num* and returns a value of some type $\tau$. The overall function has the same return type as $x$, so its type must be $(num \rightarrow \tau) \rightarrow \tau$. Let's try to infer this type. Again, we start by saying the function has some type $\alpha$:

$$\alpha = \beta \rightarrow \gamma$$

$$\text{type}(x) = \beta$$

The body of the function must have type $\gamma$:

$$\gamma = \text{type}((x\ 3))$$

Since $x$ is a function, it must have some arrow type:

$$\text{type}(x) = \delta \rightarrow \varphi$$

We know that $\delta$ must equal the type of the argument:

$$\delta = \text{type}(3) = num$$

The type of the $(x\ 3)$ is the return type of $x$:

$$\text{type}((x\ 3)) = \varphi$$

By transitivity, we now have:

$$\gamma = \varphi$$

So the type of the overall function is:

$$\alpha = (num \rightarrow \gamma) \rightarrow \gamma$$

### Example 4

$(\lambda\ (x : _)\ (*\ x\ (x\ 10)))$

This function should not type check, since $x$ is used both as *num* and as a function. How do we prove that there is no type we can infer? We know the type of $*$:

$$\text{type}(*) = num \times num \rightarrow num$$

Since $*$ takes two *num*s as arguments, and $x$ is one of the arguments, we know:

$$\text{type}(x) = num$$

In the subexpression $(x\ 10)$, $x$ is a function, so it must have an arrow type:

$$\text{type}(x) = \delta \rightarrow \varphi$$

So by transitivity, we have:

$$num = \delta \rightarrow \varphi$$

This is an unresolvable conflict—there is no way to make type *num* equal to an arrow type. Therefore the overall function has a type error.

### Example 5

Recall the *map* function:

```
(define map
  (λ (f L)
    (cond
      [(empty? L) empty]
      [(cons? L) (cons (f (first L))
                       (map f (rest L)))])))
```

**Exercise:** Verify that *map* has the following type:

$$(\alpha \rightarrow \beta) \times \text{list}(\alpha) \rightarrow \text{list}(\beta)$$

## Let expressions

What is the type rule for **let**?

$$\frac{\Gamma \vdash val : \delta \quad \Gamma[var \leftarrow \delta] \vdash body : \tau}{\Gamma \vdash (\textbf{let}\ (var\ val)\ body) : \tau}$$

Remember that we said **let** is just shorthand for the following expression:

$$(\textbf{let}\ (var\ val)\ body) \quad \equiv \quad ((\lambda\ (var)\ body)\ val)$$

In our typed language, though, we were required to annotate the function argument *var* with its type. Why don't we need to do so in a **let** expression? The reason is that we know the type of *val* immediately, so *var* must also have that type.

Can we type check the following expression using magical polymorphism?

```
(let (f (λ (x) x))
  (f 3))
```

Yes, we can. How about the following expression, though?

```
(let (f (λ (x) x))
  (if (f true)
    (f 3)
    (f 4)))
```

No, we can't type check this expression, because we use the function $f$ in two places—one where it has type $bool \rightarrow bool$, and another where it has type $num \rightarrow num$. We'll end up with a set of equations which includes:

$$\text{type}(f) = \delta \rightarrow \varphi$$

$$\delta = num$$

$$\delta = bool$$

So we end up with $num = bool$, which is a contradiction.

The above expression *should* type check though. In fact, if we just substituted $f$ into the body of the let-expression before type checking, the resulting expression would type check perfectly fine.