# Let-Based Polymorphism: Il buono, il brutto, il cattivo

sk and dbtucker

November 3, 2000

## Let-based polymorphism

We've seen an algorithm for type checking programs that use implicit polymorphism. What is the type judgment for **let**?

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma[v : \tau'] \vdash b : \tau}{\Gamma \vdash (\textbf{let}\ (v\ e)\ b) : \tau}$$

We don't need to introduce a type variable for $v$ because we can immediately determine that the type of $e$ is $\tau'$.

However, this rule doesn't allow us to use $v$ polymorphically. Consider the following expression:

```
(let (id (λ (x) x))
  (if (id true)
    (id 5)
    (id 6)))
```

Since $id$ is a function, it has some arrow type:

$$\text{type}(id) = \alpha \to \beta$$

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

The body is just $x$, so:

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

Now look at the applications of $id$. Because we apply it to $\mathsf{true}$, we know:

$$\alpha = \mathsf{bool}$$

We also apply $id$ to $\mathsf{5}$ (and $\mathsf{6}$), so:

$$\alpha = \mathsf{num}$$

Now you can see the problem: by transitivity, we have $\mathsf{bool} = \mathsf{num}$. Therefore, this expression won't type check. The problem is that once we determine that

$(\lambda\ (x)\ x)$ has type $\alpha \to \alpha$, we are stuck with the type variable $\alpha$ for every use of *id* in the body.

Remember the solution we suggested in a previous lecture—to just substitute $e$ for $v$ in the body $b$, and then type check the resulting body. More formally, the type judgment is:

$$\frac{\Gamma \vdash b[v \leftarrow e] : \tau}{\Gamma \vdash (\textbf{let}\ (v\ e)\ b) : \tau}$$

In our above example, the **let**-expression would be expanded to:

```
(if ((λ (x) x) true)
  ((λ (x) x) 5)
  ((λ (x) x) 6))
```

This expression *will* type check, because every time we use $(\lambda\ (x)\ x)$, we get a new set of type variables. That is, the first $(\lambda\ (x)\ x)$ will have type $\alpha \to \alpha$, the second $\beta \to \beta$, and the third $\gamma \to \gamma$. Thus, the constraints from function application would be:

$$\alpha = \mathsf{bool}$$

$$\beta = \mathsf{num}$$

$$\gamma = \mathsf{num}$$

and there would not be any conflicts.

The problem with this approach is that it's slow—it can make the program exponentially larger. Consider an expression with nested **let**s:

```
(let (x (let (y (let (z 3)
                     (z + z)))
              (y + y)))
     (x + x))
```

After substitution, the expression will be:

$$(((3 + 3) + (3 + 3)) + ((3 + 3) + (3 + 3)))$$

For an expression of $n$ such nested **let**s, the expansion will contain $2^n$ numeric terms.

For the expression $(\textbf{let}\ (v\ e)\ b)$, we want to introduce fresh type variables for $v$ every time we use it in $b$. After we determine that $e$ has some type $\tau'$, we will bind $v$ to a special type $\mathsf{CLOSE}(\tau')$ when type checking the body:

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma[v : \mathsf{CLOSE}(\tau')] \vdash b : \tau}{\Gamma \vdash (\textbf{let}\ (v\ e)\ b) : \tau}$$

The idea is when we use this special type, we will rename all the type variables with fresh names. The type judgment is:

$$\frac{\Gamma \vdash e : \mathsf{CLOSE}(\tau')}{\Gamma \vdash e : \tau} \qquad \text{where } \tau = \tau' \text{ with fresh type variables}$$

Now we can return to our example:

```
(let (id (λ (x) x))
  (if (id true)
    (id 5)
    (id 6)))
```

First, the type checker will determine that $id$ has type $\alpha \to \alpha$. Then, $\Gamma$ will be extended with $id : \mathsf{CLOSE}(\alpha \to \alpha)$ when type checking the body. Now $id$ will have a different type every time it is used in the body: $\alpha_1 \to \alpha_1$ when applied to true, $\alpha_2 \to \alpha_2$ when applied to 5, and $\alpha_3 \to \alpha_3$ when applied to 6. Thus, no conflicts occur in the constraint equations.

There is one subtlety we are missing, though. When we rename all the type variables in a CLOSE type, we may be renaming variables that were not created in the **let**-expression. Consider the following expression:

```
(λ (y)
  (let (f (λ (x) y))
    (if (f true)
        (+ (f false) 5)
        6)))
```

This function should not type check, since the return value of $f$ is used as both a bool in $f$ true, and as a num in $f$ false. If we apply the entire expression to true, then there will be a type error at $+$; similarly, if we apply the expression to 42, there will be a type error at **if**. However, this function *does* type check in our current type system. Let's see why.

We introduce type variables $\beta$ for $y$ and $\alpha$ for $x$. So, in the body of the **let**, $f$ will have type $\mathsf{CLOSE}(\alpha \to \beta)$. At its first application ($f$ true), $f$ will get type $\alpha_1 \to \beta_1$, and we will derive the constraints:

| | |
|---|---|
| $\alpha_1 = \mathsf{bool}$ | since true is a bool |
| $\beta_1 = \mathsf{bool}$ | since the first subexpression of **if** is a bool |

At its second application ($f$ false), $f$ will get type $\alpha_2 \to \beta_2$, and we have these constraints:

| | |
|---|---|
| $\alpha_2 = \mathsf{bool}$ | since true is a bool |
| $\beta_2 = \mathsf{num}$ | since $+$ takes nums as arguments |

Since $\beta_1$ and $\beta_2$ are distinct type variables, there are no conflicts in the constraint equations; thus, the function type checks. The problem is that we renamed $\beta$ for each use of $f$, even though $\beta$ was not created in the **let**-expression.

We only want to rename type variables that were created at **let**. To keep track of which type variables already existed, we need to store the type environment $\Gamma$ in the CLOSE type. So, we change our type judgment for **let** to be the following:

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma[v : \mathsf{CLOSE}(\tau', \Gamma)] \vdash b : \tau}{\Gamma \vdash (\textbf{let}\ (v\ e)\ b) : \tau}$$

When we use the CLOSE type, we only rename those type variables which did not already exist in the type environment:

$$\frac{\Gamma \vdash e : \mathsf{CLOSE}(\tau', \Gamma')}{\Gamma \vdash e : \tau}$$

where $\tau = \tau'$ with fresh type variables for those not in $\Gamma'$

Applying these rules in the above example, we see that the first use of $f$ will have type $\alpha_1 \to \beta$, and the second use will have type $\alpha_2 \to \beta$. Now we derive the constraints:

$$\beta = \mathsf{bool}$$

$$\beta = \mathsf{num}$$

which gives us the conflict.

This kind of polymorphism, called *let-based polymorphism*, is stronger than the kind we've seen so far. Before, our notion of polymorphism included type variables, but we could not apply a function to values of different types in the *same* context. Let-based polymorphism gives us this power; as we saw above, we could apply *id* to both a bool and a num in the same context. In ML, the top-level is basically a series of nested **let**s, which gives you polymorphism in a seemingly interactive setting.

Since we can use **let** to bind a polymorphic function, can we also do so using $\lambda$? Consider the following expression:

```
(λ (f)
  (if (f true)
      (f 5)
      6))
```

As in the example with **let**, we use the function $f$ polymorphically. However, $f$ is not bound to a value immediately, as it was in the **let**-expression; $f$ is only bound when this expression is applied, which might occur quite a bit later. So, how can we type check this function?

Our original approach to let-based polymorphism was to substitute for the bound variable into the body of the **let**-expression, and then type check the resulting expression. As an optimization, we instead created a CLOSE type, and renamed variables in the CLOSE type upon use.

This idea won't work with $\lambda$. Since we don't know what $f$ is bound to, we can't substitute for $f$ in the body of the $\lambda$-expression. Thus we rely on our current type rules, which dictate that the expression contains a type conflict, and reject the above expression.

## Boxes

Let's take a look at using **let** with boxes. Recall that operations on boxes are polymorphic functions:

$$
\begin{aligned}
&\mathit{box} : \alpha \to \mathsf{ref}(\alpha)\\
&\mathit{unbox} : \mathsf{ref}(\alpha) \to \alpha\\
&\mathit{set\text{-}box!} : \mathsf{ref}(\alpha) \times \alpha \to \mathsf{ref}(\alpha)
\end{aligned}
$$

Now consider the following expression:

```
(let (f (box (λ (x) x)))
  (set-box! (f (λ (y) (+ y 5)))
  ((unbox f) true)))
```

When evaluated, this expression will have a type error because + will be called with its first argument being true. However, this expression *does* type check under our current type system. To see why, remember that $f$ will have type $\mathsf{CLOSE}(\mathsf{ref}(\alpha \to \alpha), \{\})$ in the body of the **let**, so $\alpha$ will be renamed at each use.

The moral is that the naïve type rule for references, when combined with **let**-polymorphism, breaks down. The simplest fix is to not allow boxes on the right-hand side of **let**.