Let-based Polymorphism
Lecture Notes for cs173, Fall 2001

sk, rob, and dbtucker

November 7, 2001

1 Let-based polymorphism

We’ve seen an algorithm for type checking programs that use implicit polymorphism.
Let’s explore how powerful this polymorphism is. Consider the seemingly inane ques-
tion: What is the type judgment for $\text{let}$? Here is a reasonable judgment for it:

$$
\Gamma \vdash v : \tau \quad \Gamma[v : \tau'] \vdash b : \tau \\
\Gamma \vdash \{\text{let} \ {x} \ {v} \ {b} \} : \tau
$$

Now consider the following expression:

{let {id {proc {x} x}}
  {if {id true}
    {id 5}
  {id 6}}}

Since $\text{id}$ is a function, it has some arrow type:

$$[[\text{id}]] = \alpha \rightarrow \beta$$

$$[[x]] = \alpha$$

The body is just $x$, so:

$$\beta = [[x]] = \alpha$$

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

$$\alpha = \text{bool}$$

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

$$\alpha = \text{num}$$

Now you can see the problem: Since bool does not equal num, this expression won’t
type check. The problem is that once we determine that $\{\text{proc} \ {x} \ {x}\}$ has type
$\alpha \rightarrow \alpha$, we are stuck with the type variable $\alpha$ for every use of $\text{id}$ in the body. That is,
we can’t use \texttt{id} polymorphically—it can’t be both \texttt{bool} \rightarrow \texttt{bool} and \texttt{num} \rightarrow \texttt{num}.

The problem here is we’re trying to use the same type variable multiple times. Here’s a suggestion—substitute \texttt{val} for \texttt{var} in the body \texttt{b}, and then type check the resulting body. More formally, the type judgment is:

\[
\begin{align*}
\Gamma \vdash b[\texttt{var} \rightarrow \texttt{val}] : \tau \\
\Gamma \vdash \texttt{let} \{ \texttt{var} \texttt{val} \} \ b : \tau
\end{align*}
\]

where \(b[\texttt{var} \rightarrow \texttt{val}]\) means we syntactically substitute all instances of \texttt{var} in \texttt{b} with the expression \texttt{val}.

In our above example, the \texttt{let}-expression would be expanded to:

\[
\{\texttt{if \{\texttt{proc} \{x\} \ x\} \texttt{true}}\} \\
\{\texttt{(proc \{x\} \ x\) \ 5}\} \\
\{\texttt{(proc \{x\} \ x\) \ 6}\}
\]

This expression \textit{will} type check, because every time we use \{\texttt{proc} \{x\} \ x\}, we get a new set of type variables. That is, the first \{\texttt{proc} \{x\} \ x\} will have type \(\alpha \rightarrow \alpha\), the second \(\beta \rightarrow \beta\), and the third \(\gamma \rightarrow \gamma\). That is, the constraints from function application are

\[
\begin{align*}
\alpha &= \texttt{bool} \\
\beta &= \texttt{num} \\
\gamma &= \texttt{num}
\end{align*}
\]

and there are no conflicts.

The problem with this approach is that it’s slow—it can make the program exponentially larger. Consider an expression with nested \texttt{let}s:

\[
\{\texttt{let} \ x \ \{\texttt{let} \ y \ \{\texttt{let} \ z \ 3\} \\
\quad \{+ \ z \ z\}\} \\
\quad \{+ \ y \ y\}\} \\
\quad \{+ \ x \ x\}\}
\]

After substitution, the expression will be:

\[
(+ \ (+ \ (+ \ 3 \ 3) \ (+ \ 3 \ 3)) \ (+ \ (+ \ 3 \ 3) \ (+ \ 3 \ 3)))
\]

For an expression of \(n\) such nested \texttt{let}s, the expansion will contain \(2^n\) numeric terms.

For the expression \{\texttt{proc} \{\texttt{var} \ \texttt{val}\} \ \texttt{b}\}, we want to introduce fresh type variables for \texttt{var} every time it appears in \texttt{b}. After we determine that \texttt{val} has some type \(\tau'\), we will bind \texttt{var} to a special type \texttt{CLOSE}(\(\tau'\)) when type checking the body:

\[
\begin{align*}
\Gamma \vdash \texttt{val} : \tau' \\
\Gamma[\texttt{var} : \texttt{CLOSE}(\tau')] \vdash b : \tau \\
\Gamma \vdash \texttt{let} \{\texttt{var} \texttt{val}\} \ b : \tau
\end{align*}
\]
The idea is that when we use this special type, we will rename all the type variables with fresh names. The type judgment for `CLOSE` is:

\[ \Gamma \vdash e : \text{CLOSE}(\tau') \]  
\[ \Gamma \vdash e : \tau \]  
where \( \tau \) is \( \tau' \) with all type variables consistently renamed

Now we can return to our example:

```lang-
{let {id {proc (x) x}}
  {if {id true}
    {id 5}
    {id 6}}}
```

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

There is one subtlety we are missing. 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:

```lang-
{proc {y}
  {let {f {proc (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 `{f true}`. 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 `CLOSE(\alpha \rightarrow \beta)`. At its first application `{f true}`, `f` will get type \( \alpha_1 \rightarrow \beta_1 \), and we will derive the constraints:

\[ \alpha_1 = \text{bool} \] since `true` is a `bool`
\[ \beta_1 = \text{bool} \] since the first subexpression of `if` is a `bool`

At its second application `{f false}`, `f` will get type \( \alpha_2 \rightarrow \beta_2 \), and we have these constraints:

\[ \alpha_2 = \text{bool} \] since `false` is a `bool`
\[ \beta_2 = \text{num} \] since `+` takes `num`s 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 \texttt{let}-expression.

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

\[
\begin{align*}
\Gamma \vdash \texttt{val} : \tau' & \quad \Gamma[\texttt{var}: \texttt{CLOSE}(\tau', \Gamma)] \vdash \texttt{b} : \tau \\
\Gamma \vdash \{ \texttt{let} \ \{ \texttt{var} \ \texttt{val} \} \ \texttt{b} \} : \tau
\end{align*}
\]

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

\[
\begin{align*}
\Gamma \vdash \texttt{e} : \texttt{CLOSE}(\tau', \Gamma) \\
\Gamma \vdash \texttt{e} : \tau
\end{align*}
\]

where \( \tau \) is \( \tau' \) with all type variables not in \( \Gamma \) consistently renamed.

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

\[
\begin{align*}
\beta &= \texttt{bool} \\
\beta &= \texttt{num}
\end{align*}
\]

which gives us the conflict we want.

This kind of polymorphism, called \textit{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 \textit{same} context. Let-based polymorphism gives us this power; as we saw above, we could apply \texttt{id} to both a \texttt{bool} and a \texttt{num} in the same context. In ML, the top-level is basically a series of nested \texttt{lets}, which gives you this kind of polymorphism.