Motivating the Let in Let-Based Polymorphism

Last time we added polymorphism to our language through Rip’s let construct. But we all know that let can be re-written as procedure application. So why did we use let to infer polymorphic types? Let’s see why. Consider the following example:

\[
\begin{align*}
\text{(proc } \{ y \}) \\
\text{ (let } \{ f \ (\text{proc } \{ x \} \ x) \}) \\
\text{ (if } \{ f \ \text{true} \} \\
\text{ (f 5) } \\
\text{ 6))})
\end{align*}
\]

Using the techniques from the previous lectures, we derive the following types:

- In \{proc \{y\} \ldots\}, \( y : \alpha \)
- In \{f \{\text{proc } \{x\} x\}\}, \( f : \text{CLOSE}(\beta \rightarrow \beta, \{\alpha\}) \)
- In \{f \text{true}\}, \( f : \beta_1 \rightarrow \beta_1 \)
- In \{f 5\}, \( f : \beta_2 \rightarrow \beta_2 \)

Here, our type checker correctly concludes that this program has type \text{num}. Now consider the following slight variation to the code:

\[
\begin{align*}
\text{(proc } \{ y \}) \\
\text{ (let } \{ f \ (\text{proc } \{ x \} y) \}) \\
\text{ (if } \{ f \ \text{true} \} \\
\text{ (f 5) } \\
\text{ 6))})
\end{align*}
\]

- In \{proc \{y\} \ldots\}, \( y : \alpha \)
- In \{f \{\text{proc } \{x\} x\}\}, \( f : \text{CLOSE}(\beta \rightarrow \alpha, \{\alpha\}) \)
In \{f \; \text{true}\}, f : \beta_1 \rightarrow \alpha

In \{f \; 5\}, f : \beta_2 \rightarrow \alpha

From the last two lines, we see that it must be the case that \(\alpha = \text{bool}\) and \(\alpha = \text{num}\). There is no satisfying solution, and so we reject this program.

Now consider the following “hybrid” of the previous two programs:

\[
\{\text{proc} \ (y) \\
\quad \{\text{proc} \ (f) \\
\quad \quad \{\text{if} \ f \ \text{true} \\
\quad \quad \quad \{f \; 5\} \\
\quad \quad \quad 6}} \\
\quad \{\text{if} \ <\text{MAGIC}> \ (\text{proc} \ (x) \ x) \ (\text{proc} \ (x) \ y)\}\} \}
\]

This is a troublesome program. If \(<\text{MAGIC}>\) evaluates to \text{true} then this program is equivalent to the program in the first example of these notes; as we showed, it is an acceptable program. But if \(<\text{MAGIC}>\) evaluates to \text{false}, the program is equivalent to the second example of these notes, and we should reject the program.

The problem is that the type inference engine cannot assign a type to the fragment

\[
(\text{proc} \ (f) \\
\quad \{\text{if} \ f \ \text{true} \\
\quad \quad \{f \; 5\} \\
\quad \quad 6})
\]

As we have seen in the previous examples, anything we conclude about the type of this procedure may prove to be incorrect.

We solve this problem by restricting polymorphism: A polymorphic procedure \text{proc} can only be created with the code \{\text{let} \ (\text{var} \ \text{proc}) \ \text{body}\} where \text{proc} is a \text{proc}E, not a general RP. Variables not bound in a \text{let} will be monomorphic.

**Typing Mutation**

Let’s try to write a typing rule for mutation:

\[
\frac{}{\Gamma \vdash v : \tau} \quad \frac{}{\Gamma \vdash e : \tau} \quad \frac{}{\Gamma \vdash \{:= \ v \ e\} : \tau}
\]

But consider the following Rip program:

\[
\{\text{let} \ (f \ \text{proc} \ (x) \ x)\} \\
\quad \{\text{begin} \\
\quad \quad \{:= \ f \ \text{proc} \ (y) \ (+ \ y \ 5)\}) \\
\quad \quad \{f \ \text{true}\}\}
\]
In \{\texttt{let \{f \{proc \ldots\} \ldots\}, f:\texttt{CLOSE}(\alpha \rightarrow \alpha, \{\})}\}

In \{\texttt{:= f \{proc \ldots\}}, f:\texttt{num} \rightarrow \texttt{num}\}

In \{\texttt{f true}, f:\texttt{bool} \rightarrow \texttt{bool}\}

This program passes our type checker which assigns it the type \texttt{bool}. Unfortunately, when we run the program, it produces a type error when it tries to add \texttt{true} to 5.

It turns out that this problem doesn’t have a simple, principled solution. ML solves this problem by restricting the values bound in a polymorphic let to have special requirements. Basically, they cannot be mutated, nor can they be continuations. (The fill details are slightly more complex, but have proven to not really affect programmers in practice.)

**Type safety**

What do we mean when we say an interpreter is type safe? Here’s the definition:

*Type safety* is the property that no primitive operation is ever applied to values of the wrong (dynamic) type.

By “primitive operations,” we mean not only the application of primitives such as addition, but operations such as function application also.

Note that type safety is a property of the evaluator. In particular, a language can be type safe even if it has no static type checker; this is the case with Scheme.

We can make a table of languages based on whether they are type checked and whether they are type safe:

<table>
<thead>
<tr>
<th>safe run-time</th>
<th>static type checking</th>
<th>no static type checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsafe run-time</td>
<td>ML, Java</td>
<td>Scheme</td>
</tr>
<tr>
<td></td>
<td>C, C++</td>
<td>assembly</td>
</tr>
</tbody>
</table>

In general, as users, we want to pay the most attention to the first row. This row says that all primitive errors will be caught sometime. In ML, most of them are caught early (i.e., by the type checker), but a few, such as making sure a list is non-empty before dereferencing its first element, are deferred to run-time. Java catches fewer up-front—all casts are deferred to run-time. Scheme does all its checking at run-time.

The top-left quadrant is the nicest. Unfortunately, it’s very difficult to design a language that fits there “and stays there”. Java technically goes in that quadrant, but the sometimes extensive use of casts means it’s really somewhere half-way between ML and Scheme. So is ML the “ultimate language”? Not really. It achieves its status as king of its quadrant by placing onerous checks on some classes of programs, namely
those that extend over time, especially dynamically.

Those languages which have static type checking but are not type safe (e.g., C and C++) are truly insidious, since the type checker leads you to believe that your program will not perform any unsafe operations, when in fact you have no such guarantee.