Writing an infinite loop

Yesterday we showed how to write an infinite loop in \texttt{AFun!Exp}. To recall, our first attempt was:

\begin{verbatim}
(let (f (fun (n) (f (n))))
 (f (7)))
\end{verbatim}

This expression fails because it isn’t closed. We can try to fix it by binding an \texttt{f} above:

\begin{verbatim}
(let (f 0)
 (let (f (fun (n) (f (n))))
  (f (7))))
\end{verbatim}

But the \texttt{f} inside the function is still bound to \texttt{0}, so we get an error upon applying \texttt{0} to \texttt{7}. Finally, we decided to mutate the \texttt{f}:

\begin{verbatim}
(let (f 0)
 [(set f (fun (n) (f (n))))
  (7)])
\end{verbatim}

Now the \texttt{f} in the function refers to a box, but the contents of that box are \texttt{set} to the function. So the recursion works.

Let’s look at the example a bit more carefully:

1. The initial environment is:

\[ E_0 \equiv (\texttt{fresh-sub}) \]

2. The \texttt{let} expression creates a new environment that binds \texttt{f} to \texttt{0} in its body:

\[ E_1 \equiv (\texttt{new-sub }'f\ (\texttt{numV 0})\ E_0) \]

3. When the \texttt{fun} is evaluated, a closure is created which captures the environment \( E_1 \):

\[ (\texttt{funV }'n\ (...)\ E_1) \]
4. The set expression then changes \( E_1 \), so we get the following equivalence:

\[
E_1 \equiv (\texttt{new-sub} \ 'f \ (\texttt{funV} \ 'n \ (\ldots) \ E_1) \ E_0)
\]

Looking at this last equality, we see that we need an environment that refers to itself. If we let:

\[
P(E) = (\texttt{new-sub} \ 'f \ (\texttt{funV} \ 'n \ (\ldots) \ E) \ E_0)
\]

the equation for \( E_1 \) can be rewritten as:

\[
E_1 = P(E_1)
\]

\( E_1 \) is called the fixed point of \( P \).

**Fixed point digression**

It’s not obvious that a function has a unique fixed point. Consider these three functions:

1. \( f(x) = x \) has infinitely many fixed points.
2. \( f(x) = 0 \) has exactly 1 fixed point.
3. \( f(x) = x + 1 \) has no fixed points.

However, \( P \) does have a fixed point.

**Adding recursive bindings to our language**

We often use recursive functions when writing programs, so we’ll add a construct \texttt{rec} to our language\(^1\) to explicitly support recursion:

\[
L ::= \ldots \\
| (\texttt{rec} \langle \texttt{id} \rangle \langle L \rangle \langle L \rangle)
\]

For example, we can write an infinite loop as follows:

\[
(\texttt{rec} \ f \ (\texttt{fun} \ (x) \ (f \ (x)))) \\
(f \ (17)))
\]

The variable \( f \) is bound to the function, and the environment in the function’s closure includes this binding.

We add a \texttt{recE} variant to the datatype for the abstract syntax tree:

\[
(\texttt{define-datatype} \ A\texttt{FunRecExp} \ A\texttt{FunRecExp}?) \\
\texttt{[varE} \ (v \ \texttt{symbol?})]\]

\(^1\)Similar to Scheme’s \texttt{letrec}.
Now we need to write the recE case in the interpreter:

\[\text{recE } \text{(var fun body) . . .} \]

Remember our discussion above—we want to interpret body in the environment given by the fixed point of \(P\). Let’s suppose we have a function \(\text{fix-env}\) which computes the desired fixed point:

\[\text{recE } \text{(var fun body)} \text{ (let ([P . . .])} \text{ (interp body (fix-env P))})\]

Next we fill in \(P\), which is the environment transformer we defined above:

\[\text{recE } \text{(var fun body)} \text{ (let ([P (lambda (env) \text{(new-sub var)}})} \text{ (funV (get-funE-param fun)}} \text{ (get-funE-body fun)}} \text{ env)}) \text{ d))}) \text{ (interp body (fix-env P))})\]

We still have to write the function \(\text{fix-env}\) which computes the fixed point of the environment transformer \(P\). If we use the function representation for environments, we can use Scheme’s \text{letrec}\ to define \(\text{fix-env}\):

\[
\text{(define (fix-env P)} \text{ (letrec ([rec-env (P (lambda (id)}} \text{ (get-sub id rec-env))}] \text{ (get-sub id rec-env))}] \text{ rec-env}))
\]

How does \(\text{fix-env}\) work? We define a new environment \(\text{rec-env}\) which contains the binding for the recursive function. The environment in the recursive function’s closure is \(\text{lambda (id) (get-sub id rec-env)}\); in other words, it just looks up the identifier in \(\text{rec-env}\). Thus, the recursive function can refer to itself.

Another approach to writing \(\text{fix-env}\) is to use mutation. We can create a dummy value as the closure’s environment, and then apply the transformer \(P\) to get the environment in which to evaluate body (call it \(\text{rec-env}\)). Then we mutate the dummy value to be \(\text{rec-env}\), so the function closure’s environment contains the binding for the function. The code is:
(define (fix-env F)
  (let* ([set-env (box (fresh-sub))]
          [rec-env (F set-env)])
    (set-box! set-env rec-env)
    rec-env))

(The let* expression is just shorthand for nested let expressions.) We also have to modify two other parts of the interpreter because we use boxes to implement mutation: environments are boxed when closures are created in funE, and environments are unboxed when calling interp in apply-fun.