RECURSION IN FUNCTIONAL LANGUAGES

The simple, pure functional language we described is incomplete without some form of iteration. We will add the two components necessary to make the language as expressive as any real functional language. With the addition of arithmetic the language would come very close to Scheme, the cleanest, purest of the functional languages in common use.

Functional languages achieve iteration through recursion, and the addition of a conditional form will enable the language to test for base cases which stop the recursion. As with the semantics of the while loop, which can yield an infinite computation we will need to handle the possibility of an infinite recursion. Not surprisingly, the fixed point technique used for the while loop will be used for recursion.

The additional forms of the expression are:

\[
E ::= \ldots | \text{LETREC } I = E_1 \text{ IN } E_2 | \text{IFNULL } E_1 \text{ THEN } E_2 \text{ ELSE } E_3
\]

The conditional only needs to test its first argument against the empty list NIL since there is no arithmetic. The bug difference is that whereas the normal LET does not allow recursive references, the new form LETREC does. For example, in

\[
\text{LET } F = \text{LAMBDA (X) F (HEAD X)}
\]

the reference to F in the body of the LAMBDA expression does not refer to the F being defined, which would be a recursive reference, but instead must refer to another version of F bound in some outer LET. On the other hand,

\[
\text{LETREC } F = \text{LAMBDA (X) F (HEAD X)}
\]

changes the picture so that F is now bound to a recursively defined function. As is normal with static scoping, if a local reference is available it takes precedence over any non-local version.

SEMANTICS OF RECURSION

First, the semantics of the conditional are very reminiscent of the same form in the imperative world:

\[
E\left[\text{IFNULL } E_1 \text{ THEN } E_2 \text{ ELSE } E_3\right] =
\]

\[
\lambda e.\text{let } x = (E\left[E_1\right]e)\text{ in}
\]

\[
\text{cases } x \text{ of}
\]

\[
\text{isFunction(f)} \rightarrow \text{inError()}
\]

\[
\square \text{isList(t)} \rightarrow (\square (\text{null t}) \rightarrow (E\left[E_2\right]e) \square (E\left[E_3\right]e))
\]

\[
\square \text{isAtom(a)} \rightarrow \text{inError()}
\]

\[
\square \text{isError()} \rightarrow \text{inError()}
\]

\end

Recursion is handled with a recursively defined environment:

\[
E\left[\text{LETREC } I = E_1 \text{ IN } E_2\right] = \lambda e.E\left[E_2\right]e'
\]

where \(e' = \text{updateenv}\left[I\right](E\left[E_1\right]e')e\)

Why this works is because each recursive call adds another binding of I to the environment. The first environment is the one handed to LETREC, i.e. \(e = e_0\). The next one is \(\text{updateenv}\left[I\right](E\left[E\right]e_0)e_0 = e_1\), the next is \(\text{updateenv}\left[I\right](E\left[E\right]e_1)e_1\), and so on. The solution to the recursive equation is the fixed point of the functional \(G = \lambda e'.\text{updateenv}\left[I\right](E\left[E\right]e')e\). As with the while loop solution the zeroth partial solution is the “no information” function. The partial solutions are built up by applying the functional to produce a chain:
\[ G^0 = \lambda i. \bot \]

\[ G^1 = \text{updateenv} \left[ I \right]\left( E \left[ E_1 \right]\left( G^0 \right) \right) e \]

\[ = \text{updateenv} \left[ I \right]\left( E \left[ E_1 \right]\left( \lambda i. \bot \right) \right) e \]

\[ G^2 = \text{updateenv} \left[ I \right]\left( E \left[ E_1 \right]\left( G^1 \right) \right) e \]

\[ = \text{updateenv} \left[ I \right]\left( E \left[ E_1 \right]\left( \text{updateenv} \left[ I \right]\left( \lambda i. \bot \right) \right) e \right) \]

etc.

The general case is:

\[ G^{i+1} = \text{updateenv} \left[ I \right]\left( E \left[ E_1 \right]\left( G^i \right) \right) e \]

Each successive \( G \) produces a better approximation to the actual environment which is the least upper bound of the chain, which is the fixed point of the functional. The valuation function for LETREC will thus be written:

\[ E \left[ \text{LETREC I = E}_1 \ \text{IN E}_2 \right] = \lambda e. E \left[ E_2 \right]\left( \text{fix} \left( \lambda e'. \text{updateenv} \left[ I \right]\left( E \left[ E_1 \right] e' \right) \right) \right) \]

**EXAMPLE DERIVATION**

We will derive the semantics of the program

\[
\text{LETREC F = LAMBDA (X) IFNULL X THEN NIL ELSE a}_0 \ \text{CONS (F (TAIL X)) IN}
\text{F (a}_1 \ \text{CONS (a}_2 \ \text{CONS NIL))}
\]

This recursive function traverses the list using TAIL, and returns a list of the same length but with the atom \( a_0 \) in the place of whatever was in the original list. If we use the substitution method of derivation we get:

\[
\text{LETREC F = LAMBDA (X) IFNULL X THEN NIL ELSE a}_0 \ \text{CONS (F (TAIL X)) IN}
\text{F (a}_1 \ \text{CONS (a}_2 \ \text{CONS NIL))}
\]

\[= (\text{LAMBDA (X) IFNULL X THEN NIL ELSE a}_0 \ \text{CONS (F (TAIL X))}) \ (a}_1 \ \text{CONS a}_2 \ \text{CONS NIL})\]

\[= a}_0 \ \text{CONS (F (TAIL (a}_1 \ \text{CONS (a}_2 \ \text{CONS NIL)))))\]

\[= a}_0 \ \text{CONS (F (a}_2 \ \text{CONS NIL))}\]

\[= a}_0 \ \text{CONS ((LAMBDA (X) IFNULL X THEN NIL ELSE a}_0 \ \text{CONS (F (TAIL X))}) \ (a}_2 \ \text{CONS NIL))\]

\[= a}_0 \ \text{CONS (a}_0 \ \text{CONS (F (TAIL (a}_2 \ \text{CONS NIL)))))\]

\[= a}_0 \ \text{CONS (a}_0 \ \text{CONS (F NIL))}\]

\[= a}_0 \ \text{CONS (a}_0 \ \text{CONS ((LAMBDA (X) IFNULL X THEN NIL ELSE a}_0 \ \text{CONS (F (TAIL X)) NIL))}\]

\[= a}_0 \ \text{CONS (a}_0 \ \text{CONS NIL)}\]

If we use the semantics of LETREC, we get

Let \( E_0 = \text{LAMBDA (X) E}_1 \)

\( E_1 = \text{IFNULL X THEN NIL ELSE a}_0 \ \text{CONS (F (TAIL X))} \)

\( E_2 = F(a}_1 \ \text{CONS (a}_2 \ \text{CONS NIL}) \)
\[ E \text{LETREC } F = E_{0} \text{ IN } E_{2} \parallel e_{0} \]
\[ = E[ E_{2} ] e_{1} \text{, where } e_{1} = \text{fix}(G) , G = (\lambda e'. \text{updateenv}[F] (E[ E_{0} ] e') e_{0} ) \]
\[ = E[ F \ (a_{1} \ \text{CONS} \ (a_{2} \ \text{CONS NIL})) ] e_{1} \]
\[ = \text{let } x = E[ F ] (\text{fix}(G)) \text{ in} \]
\[ \quad \text{cases } x \text{ of} \]
\[ \quad \quad \text{isFunction}(f) \to f (E[ (a_{1} \ \text{CONS} \ (a_{2} \ \text{CONS NIL})) ] e_{1}) \]
\[ \quad \quad \text{... end} \]

Now looking up \( F \) in the environment is
\[ \text{accessenv}[F] (\text{fix}(G)) \]
\[ = \left( \text{fix}(G) \right)[F] \]
\[ = \left( G(\text{fix}(G)) \right)[F] , \text{ from the fixed point property of } G \]
\[ = \left( \text{updateenv}[F] (E[ E_{0} ] (\text{fix}(G))) e_{0} \right)[F] , \text{ substituting in } G \]
\[ = E[ E_{0} ] (\text{fix}(G)) , \text{ since we are accessing } F \]
\[ = E[ \text{LAMBDA} (X) E_{1} ] (\text{fix}(G)) \]
\[ = E[ \text{LAMBDA} (X) E_{1} ] e_{1} \]
\[ = \text{inFunction}(\lambda d. E[ E_{1} ] (\text{updateenv}[X] d e_{1})) \]

Going back to the original application of \( F \), having shown \( x \) to be a function, we can apply it:
\[ (\lambda d. E[ E_{1} ] (\text{updateenv}[X] d e_{1})) (E[ a_{1} \ \text{CONS} \ (a_{2} \ \text{CONS NIL}) ] e_{1}) \]
\[ = E[ E_{1} ] (\text{updateenv}[X] (E[ a_{1} \ \text{CONS} \ (a_{2} \ \text{CONS NIL}) ] e_{1}) e_{1}) \]

The evaluation of the constant list produces \( \text{inList} (\text{inAtom}(a_{1}) \ \text{cons} \ (\text{inAtom}(a_{2}) \ \text{cons nil})) \), and the new environment binds this to \( X \). Call this \( e_{2} \). Then
\[ E[ E_{1} ] e_{2} \]
\[ = E[ \text{IFnulL X THEN NIL ELSE a}_{0} \ \text{CONS} \ (F \ \text{TAIL X}) ] e_{2} \]
\[ = \text{let } x = (E[X] e_{2}) \text{ in} \]
\[ \quad \text{cases } x \text{ of } \ldots \text{end} \]
\[ = \text{null} (\text{inList} (\text{inAtom}(a_{1}) \ \text{cons} \ (\text{inAtom}(a_{2}) \ \text{cons nil}))) \to \]
\[ \quad (E[ \text{NIL} ] e_{2}) \cup (E[ a_{0} \ \text{CONS} \ F \ \text{TAIL X}] e_{2}) \]
\[ = E[ a_{0} \ \text{CONS} \ F \ \text{TAIL X}] e_{2} \]
\[ = \text{inList} (\text{inAtom}(a_{0}) \ \text{cons } E[ F \ \text{TAIL X}] e_{2}) \]

The application gives
\[
= E[F(TAIL \ X)]e_2 \\
= \text{let } x = E[F](fix(G)) \text{ in} \\
\text{cases } x \text{ of} \\
\text{isFunction}(f) \rightarrow f(E[TAIL \ X]e_2) \\
\ldots \text{ end}
\]

The evaluation of TAIL \ X gives \text{inList}(\text{inAtom}(a_2) \ cons \ nil). The expansion of \text{fix}(G) \text{ is as before, so applying the function } f \text{ gives}

\[
E[E_1]\left(\text{updateenv}[X]\left(\text{inList}(\text{inAtom}(a_2) \ cons \ nil)\right)e_1\right) \\
= E[E_1]e_3
\]

Again the evaluation of \text{E}_1 \text{ is not true, so we have another application of } F \text{ to the TAIL, which is now } \text{nil}.

\[
= E[F(TAIL \ X)]e_3 \\
= \text{let } x = E[F](fix(G)) \text{ in} \\
\text{cases } x \text{ of} \\
\text{isFunction}(f) \rightarrow f(\text{nil}) \\
\ldots \text{ end}
\]

\[
= \text{null}(\text{nil}) \rightarrow \\
\quad\left(E[NIL]e_2\right) \sqcup \left(E[a_0 \ CONS F(TAIL \ X)]e_2\right) \\
= E[NIL]e_3 \\
= \text{nil}
\]

The final result is, as expected, \text{inList}(\text{inAtom}(a_0) \ cons (\text{inAtom}(a_0) \ cons \ nil))