Language Implementation: Generalities

Template Activation record

Consider the imperative language program

procedure f(w);
begin
  int x,y;
  x := 6 + z;
  z := 7
end
We need

1. the *entry point* to the code;

2. Means of finding the values of non local variables.
Activation records

All the data structure necessary for the proper execution of a procedure.

Note that whereas there is just one Template activation record per procedure there can be many activation records, one for each invocation.
The activation record contains:

- work space

- local environment

- global environment

- return

- caller a pointer to the activation record of the caller
Virtual Machines

When compiling code in a procedural language we produce a sequence of instructions for the target machine which, when evaluated gives rise to an evaluation record structure.

One can, for example, place the information required to form the activation record in registers. This amounts to calling the procedure.
Implementing a functional language

For a conventional machine architectures we can regard a program of machine code instructions as a sequence or list of actions to be performed.

Another way to think of this is as a right linear binary tree. So each left node is an instruction. Note that the execution does not change the tree.
Combinator Compilers

- The Functional program program is represented by a combinator

- There is NO global state, the program IS the state.

- Evaluation of the program proceeds only by altering the structure of the program.

- Programs are again binary trees but they are not necessarily right linear.

- Nodes are (mainly) application operators.

- A node represents the application of its left subprogram to its right subprogram.

- This has implications for parallelism.
Compiling Combinatory Programs
Getting the combinatory form

Consider the program:

\[
\text{def } f \ n = \text{if } (n=0) \ 1 \ (n \ast (f(n-1)))
\]

Abstracting wrt to \( n \) gives

\[
\text{def } f = [n]\text{if } (n=0) \ 1 \ (n \ast (f(n-1)))
\]
Combinator Compilation 2.

Which leads to the following combinatory code:

\[
f = S(S(k \textbf{if})(S (S (K \textbf{eq}) (K 0)) I)) \\
    (K 1))(S (S (K \textbf{times}) I) (S (K f) \\
    (S (S (K \textbf{minus}) I) (K 1))))
\]

This can be optimized to

\[
f = S(C(B \textbf{if}(eq 0)) 1) \\
    (S \textbf{times} (B f(C \textbf{minus} 1)))
\]

Note that these expressions involve combinators \textbf{if}, \textbf{eq}, \textbf{times} and \textbf{minus}.
Syntax

fundef ::= clause | clause fundef
clause ::= def v pl = e
pl ::= empty | p | pl p
p ::= v | c | (p) | p:p

v variables
pl pattern lists
c constants
e expressions
\texttt{compile} [clause fundef] =
T (\texttt{compile} [clause])(\texttt{compile} [fundef])

\begin{align*}
Txyz & = yz & \text{when } xz = F \\
Txyz & = T(xz)(yz) & \text{when } (xz) \text{ applicative} \\
Txyz & = xz & \text{otherwise}
\end{align*}

F is a failure combinator. \(Fx = F\) for all \(x\)
Compiling the individual clauses

\textbf{compile } [ \texttt{def} v \ p l = e ] = [ p l ] e

where $[ p l ] e$ signifies abstraction. So we need:

empty $e = e$

$p l \ p = [ p l ] ([ p ] e)$
Patterns

For variables we already have:

\begin{align*}
[x]x & = I \\
[x]y & = K \ y \ (y \neq x) \\
[x]E_1E_2 & = S[x]E_1[x]E_2 \\

[(p)]e & = [p] \ e \\
[p_1 : p_2]e & = U[p_1]([p_2]e)
\end{align*}

$U$ (the Uncurry combinator) is defined by

$U \ x \ y : z = x \ y \ z$

$Ux \ y = F \ \text{otherwise}$
Constants

\[ [c]e = M \ c \ e \]

\( M \) is the matching combinator defined as:

\[ M \ c \ e \ c = e \]
\[ M \ c \ e \ x = F \]
cons

If we represent cons as:

\[
\begin{array}{c}
\vdots \\
\downarrow \\
\Downarrow \\
\vdots \\
\end{array}
\quad \quad \text{and not}

\begin{array}{c}
\vdots \\
\downarrow \\
\Downarrow \\
\vdots \\
\end{array}
\]

\[e_1 \quad e_2 \quad e_2 \quad e_1\]

Then we must deal with \([v]e_0 : e_1\)

\[[v]e_0 : e_1 = S:[v]e_0[v]e_1\]

where \(S:x\ y\ z = (x\ z):(y\ z)\)
Representations and Rewrites

\[
\begin{array}{c}
\text{ap} \\
\text{operator} \\
\text{operand}
\end{array} \quad \begin{array}{c}
\text{;} \\
\text{head} \\
\text{tail}
\end{array}
\]
Cons.

[E0 E1 E2]
Application

S E0 E1 E2
Reduction

\[ S \ E_0 \ E_1 \ E_2 = E_0 \ E_2 \ (E_1 \ E_2) \]

Dotted line denotes structure sharing.