SECD, Tail Recursion, and Continuations

CS 6520, Spring 2000

SECD Machine

P. J. Landin, "The Mechanical Evaluation of Expressions," in *Computer Journal* 6(4), 1964.

Compiler

To use the machine, we must first compile ISWIM source expressions to a sequence of "bytecodes":

$$
\begin{aligned}
[\![b]\!] &= b \\
[\![x]\!] &= x \\
[\![(M_1\ M_2)]\!] &= [\![M_1]\!]\ [\![M_2]\!]\ \mathsf{ap} \\
[\![(o\ M_1\ M_2)]\!] &= [\![M_1]\!]\ [\![M_2]\!]\ \mathsf{prim}_o \\
[\![(\lambda x.M)]\!] &= \langle x, [\![M]\!] \rangle
\end{aligned}
$$

Machine

Each evaluation step applies to a tuple of

S	a stack for values, L
E	an envrionment binding variables, x, to values, L
C	a control string
D	a "dump" representing a saved machine state

Evaluation rules:

$$
\begin{array}{lcll}
\langle S, E, b\ C, D\rangle & \longmapsto & \langle b\ S, E, C, D\rangle & 1 \\
 & & \quad \text{where } L = E(x) & \\
\langle S, E, x\ C, D\rangle & \longmapsto & \langle L\ S, E, C, D\rangle & 2 \\
 & & \quad \text{where } L = E(x) & \\
\langle b_2\ b_1\ S, E, \mathsf{prim}_o\ C, D\rangle & \longmapsto & \langle L\ S, E, C, D\rangle\rangle & 3 \\
 & & \quad \text{where } L = \delta(o, b_1, b_2) & \\
\langle S, E, \langle x, C'\rangle\ C, D\rangle & \longmapsto & \langle\langle x, C', E\rangle\ S, E, C, D\rangle & 4 \\
\langle L\ \langle x, C', E'\rangle\ S, E, \mathsf{ap}\ C, D\rangle & \longmapsto & \langle\emptyset, E'[x \leftarrow L], C', \langle S, E, C, D\rangle\rangle & 5 \\
\langle L\ S, E, \emptyset, \langle S', E', C', D\rangle\rangle & \longmapsto & \langle L\ S', E', C', D\rangle & 6
\end{array}
$$

SECD versus CEK

The SECD and CEK machines produce the same result for any expression that has a result, but they compute the result in a different way. As it turns out, a programmer might notice the difference when using a real machine with finite resources.

The difference is in how is accumulated and saved while subexpressions are evaluated:

- In the SECD machine, context is created by function calls, where the current stack, environment, and control are packaged into a dump. The stack provides a working area for assembling application arguments..

  This view of context accumulation is natural when approaching computation from the Pascal or C model.

- In the CEK machine, context is created when evaluating an application function or argument, regardless of whether the function or argument is a complex expression. (The argument stack is subsumed by the continuation.) Function application always shrinks the context, rather than expanding it.

  This view of context accumulation is natural when approaching computation from the lambda calculus model.

Due to this difference, the SECD and CEK machines behave differently on recursive programs. As an extreme example, consider the evaluation of Ω:

$$(\lambda x.x\ x)(\lambda x.x\ x)$$

Both machines must loop forever on this expression. However, the nature of the infinite loop is different. The CEK machine's state will never grow beyond a certain size while evaluating:

$$
\begin{array}{lll}
 & \langle\langle((\lambda x.x\ x)(\lambda x.x\ x)), \{\}\rangle, \mathsf{mt}\rangle & \\
\longmapsto & \langle\langle(\lambda x.x\ x), \{\}\rangle, \langle\mathsf{arg}, \langle(\lambda x.x\ x), \{\}\rangle, \mathsf{mt}\rangle\rangle & \\
\longmapsto & \langle\langle(\lambda x.x\ x), \{\}\rangle, \langle\mathsf{fun}, \langle(\lambda x.x\ x), \{\}\rangle, \mathsf{mt}\rangle\rangle & \\
\longmapsto & \langle\langle(x\ x), \{x = L\}\rangle, \mathsf{mt}\rangle & \text{where } L = \langle(\lambda x.x\ x), \{\}\rangle \\
\longmapsto & \langle\langle x, \{x = L\}\rangle, \langle\mathsf{arg}, \langle x, \{x = L\}\rangle, \mathsf{mt}\rangle\rangle & \\
\longmapsto & \langle\langle(\lambda x.x\ x), \{\}\rangle, \langle\mathsf{arg}, \langle x, \{x = L\}\rangle, \mathsf{mt}\rangle\rangle & \\
\longmapsto & \langle\langle x, \{x = L\}\rangle, \langle\mathsf{fun}, \langle(\lambda x.x\ x), \{\}\rangle, \mathsf{mt}\rangle\rangle & \\
\longmapsto & \langle\langle(\lambda x.x\ x), \{\}\rangle, \langle\mathsf{fun}, \langle(\lambda x.x\ x), \{\}\rangle, \mathsf{mt}\rangle\rangle & \\
\longmapsto & \langle\langle(x\ x), \{x = L\}\rangle, \mathsf{mt}\rangle & \text{which is the same as five steps back} \\
\ldots & &
\end{array}
$$

In contrast, the state of the SECD machine will grow forever:

$$
\begin{array}{lll}
 & \langle\emptyset, \{\}, \langle x, x\ x\ \mathsf{ap}\rangle\ \langle x, x\ x\ \mathsf{ap}\rangle\ \mathsf{ap}, \emptyset\rangle & \text{since } [\![(\lambda x.x\ x)(\lambda x.x\ x)]\!] = \langle x, x\ x\ \mathsf{ap}\rangle\ \langle x, x\ x\ \mathsf{ap}\rangle\ \mathsf{ap} \\
\longmapsto & \langle\langle x, x\ x\ \mathsf{ap}, \{\}\rangle, \{\}, \langle x, x\ x\ \mathsf{ap}\rangle\ \mathsf{ap}, \emptyset\rangle & \\
\longmapsto & \langle\langle x, x\ x\ \mathsf{ap}, \{\}\rangle\ \langle x, x\ x, \{\}\rangle, \{\}, \mathsf{ap}, \emptyset\rangle & \\
\longmapsto & \langle\emptyset, \{x = L\}, x\ x\ \mathsf{ap}, \langle\emptyset, \{\}, \emptyset, \emptyset\rangle\rangle & \text{where } L = \langle x, x\ x\ \mathsf{ap}, \{\}\rangle \\
\longmapsto & \langle L, \{x = L\}\rangle, \{x = L\}, x\ \mathsf{ap}, \langle\emptyset, \{\}, \emptyset, \emptyset\rangle\rangle & \\
\longmapsto & \langle L\ L, \{x = L\}, \mathsf{ap}, \langle\emptyset, \{\}, \emptyset, \emptyset\rangle\rangle & \\
\longmapsto & \langle\emptyset, \{x = L\}, x\ x\ \mathsf{ap}, \langle\emptyset, \{x = L\}, \emptyset, \langle\emptyset, \{\}, \emptyset, \emptyset\rangle\rangle\rangle & \text{same S, E, and C as before, but larger D} \\
\ldots & & \\
\longmapsto & \langle\emptyset, \{x = L\}, x\ x\ \mathsf{ap}, \langle\emptyset, \{x = L\}, \emptyset, \langle\emptyset, \{x = L\}, \emptyset, \langle\emptyset, \{\}, \emptyset, \emptyset\rangle\rangle\rangle\rangle & \\
\ldots & &
\end{array}
$$

In other words, the CEK machine evaluates Ω as a loop, while the SECD machine evaluates Ω as infinite recursion *a la* Pascal or C.

The notion of saving context on argument evaluations, rather than on function calls, corresponds to *tail recursion* in programming languages. For example, the following expression runs forever in Scheme, without running out of stack space:

```
(begin
  (define (f x) (f x))
  (f 'ignored))
```

More generally, the semantics of Scheme must be understood in terms of a CEK-like model, rather than an SECD-like model. More precisely, the CEK model inspired the definition of Scheme.

Of course, the relevance of tail recursion extends beyond infinite loops. Since real machines generally have a finite amount of memory, the difference between loops and stack-based recursion is important for large (but bounded) recursions as well.

For example, every programmer knows that C function which processes a long list should be implemented as a loop rather than with recursion, to avoid blowing the stack. In languages that provide tail recursion, a recursive definition can be just as efficient as a loop. Indeed, languages such as Scheme and ML do not even provide a looping construct, since it is unecessary.

Continuations

The CEK machine has inspired language designers in another way: explicit continuations.

Although we had in mind a control stack for the K register of the CEK machine, we have implemented it using structures. In principle, there's no reason that those structures cannot be delivered to a program as values.

What would a program do with such a value? The only use for a continuation is in the last slot of the machine. So the only useful operation a program might perform on a continution is to use it — replacing the current continuation!

To investigate the use of continuations as values, we extend ISWIM as follows:

$$\begin{array}{rcl}
V & = & \ldots \mid K \\
M & = & \ldots \mid (\texttt{letcc}\ x\ M) \mid (\texttt{cc}\ M\ M) \\
K & = & \ldots \mid \langle \mathsf{ccval}, \langle M, E\rangle, K\rangle \mid \langle \mathsf{cc}, K, K\rangle
\end{array}$$

Now, continuations are allowed as values, and we have two new syntactic forms. The first one grabs the current continuation and binds it to a variable while evaluting an expression. The second one replaces the current continuation with the value of its first subexpression, and supplies the value of its second subexpression as the result. We also have two new kinds of continuation, since the `cc` form has two subexpressions.

To define the semantics of our extended language, we add the following rules to the CEK machine:

$$\begin{array}{lcll}
\langle\langle(\texttt{letcc}\ x\ M), E\rangle, K\rangle & \longmapsto & \langle\langle M, E[x \leftarrow \langle K, \{\}\rangle]\rangle, K\rangle & 8 \\
\langle\langle(\texttt{cc}\ M\ N), E\rangle, K\rangle & \longmapsto & \langle\langle M, E\rangle, \langle \mathsf{ccval}, \langle N, E\rangle, K\rangle\rangle & 9 \\
\langle\langle K', E\rangle, \langle \mathsf{ccval}, \langle N, E'\rangle, K\rangle\rangle & \longmapsto & \langle\langle N, E'\rangle, \langle \mathsf{cc}, K', K\rangle\rangle & 10 \\
\langle\langle V, E\rangle, \langle \mathsf{cc}, K', K\rangle\rangle & \longmapsto & \langle\langle V, E\rangle, K'\rangle\rangle & 11 \\
\quad \text{if } V \notin Vars & & &
\end{array}$$

Using Continuations

Our extensions to ISWIM allow a program to grab the current context and restore it later — regardless of whatever might be happening at that later point. The context at the later point is discarded, so if we were in the process of evaluating some argument to a primitive, then we forget about the primitive application, etc., and return to whatever we were doing before.

Here's an example:

$$\begin{array}{l}
(+\ 1 \\
\quad (\texttt{letcc}\ \mathrm{x} \\
\qquad (+\ (\lambda\ \mathrm{y}\ .\ \mathrm{y}) \\
\qquad\quad (\texttt{cc}\ \mathrm{x}\ 12))))
\end{array}$$

The example contains an addition where the function is the first argument. And evaluating the expression does start evaluating the addition. However, before the addition can discover that the first argument is not a number, the ($\texttt{cc}x12$) expression "jumps" out of the addition. The result is going to be 13.

Let's walk through it slowly. To evaluate the outside addition, we first evaluate 1 (which has the value 1), and then consider the `letcc` expression. At that point, the context (i.e., the work left to do when we finish the `letcc` expression) is

$$\begin{array}{l}
(+\ 1 \\
\quad \boxed{})
\end{array}$$

The box above corresponds to the "hole" where the result of the `letcc` expression goes.

The `letcc` expression does something special: it binds the context above (with a hole) to the variable x. Then it starts evaluating the addition in the body of the `letcc` expression.

When we get to the `cc` expression, the context is

$$\begin{array}{l}
(+\ 1 \\
\quad (\texttt{letcc}\ \mathrm{x} \\
\qquad (+\ (\lambda\ \mathrm{y}\ .\ \mathrm{y}) \\
\qquad\quad \boxed{})))
\end{array}$$

However, this context turns out not to matter, because the `cc` expression throws it away and replaces it with the older context bound to x:

$$\begin{array}{l}(+\ 1\\ \quad \boxed{})\end{array}$$

The `cc` expression also provides 12 as the result to go into the restored context, so the expression to evaluate becomes

$$\begin{array}{l}(+\ 1\\ \quad 12)\end{array}$$

The final result is 13.

The example above shows how continuations can be used to define an error-handling mechanism. Continuations can also be used for implementing cooperative threads ("coroutines"), or for saving and restoring speculative computations.