The SECD machine

Pedro Vasconcelos

February 16, 2024

What is the SECD machine?

- An interpreter for the functional language ISWIM (Landin, 1964)
- A virtual machine for compiling LISP/Scheme (Henderson, 1980)
- Used in some real implementations (Lispkit)
- For *call-by-value* languages
- Can be modified for *lazy evaluation* (but there are more eficient alternatives)

Abstract or Virtual machine?

- The original SECD interprets abstract syntax terms (abstract machine)
- We will present a machine that interprets pseudo-instructions (virtual machine)
- We will omit:
  1. data structures (lists, tuples, etc.);
  2. choice of concrete memory layout;
  3. translation of pseudo-instructions to real machine code;
  4. runtime system needed for execution: memory allocation, *garbage collection*, ...

Bibliography

1. Chapter 6 of *Functional Programming: Application and Implementation*, Henderson, 1980, Prentice-Hall International.
2. Chapter 7 of *The Architecture of Symbolic Computers*, Kogge, 1991, McGraw-Hill International.

SECD: Stack, Environment, Control & Dump

The configuration of the machine is a quintet:

$$\langle s, e, c, d, m \rangle$$

- s stack of temporary values;
- e environment for free variables;
- c instruction sequence (control);
- d stack of function calls (dump);
- m memory store for closures.

Name resolution

During compilation we will associate variable names to environment indices.

Interpreter

$$\begin{array}{rl} \text{term:} & x + y \\ \text{environment:} & [x \mapsto 23,\ y \mapsto 42] \end{array}$$

Compiler

$$\left.\begin{array}{rl} \text{term:} & x + y \\ \text{symbol table:} & [x \mapsto 0, y \mapsto 1] \end{array}\right\} \text{compilation}$$

$$\left.\begin{array}{rl} \text{generated code:} & [\mathrm{LD}\ 0,\ \mathrm{LD}\ 1,\ \mathrm{ADD}] \\ \text{environment:} & [23, 42] \end{array}\right\} \text{execution}$$

Note: Henderson's SECD uses nested environments indexed with pairs (i, j) instead of single integers.

De Bruijn notation

Identify variables by the depth of the λ binder:

$$\begin{array}{ccccc} \lambda x. & (\lambda y. & y & x) & x \\ \lambda & (\lambda & 0 & 1) & 0 \end{array}$$

- Each variable is associated with an index i
- Environments become simple lists of values:

$$[v_0, v_1, \ldots, v_i, \ldots]$$

Closures I

Recall that function values can represented by closures, i.e. pairs of terms and environments.

$$(\underbrace{\lambda y.\, x + y}_{\lambda\text{-term}},\ \underbrace{[x \mapsto 2]}_{\text{environment}}\ )$$

In the SECD machine, λ-terms are translated into compiled code:

$$\text{Closure} = (\text{Code}, \text{Env})$$

Closures II

We represent the store as a partial function from addresses to closures:

$$\text{Store} = \text{Addr} \to \text{Closure}$$

The next function gives the next free address:

$$\text{next} :: \text{Store} \to \text{Addr}$$

Temporary stack

The SECD is a stack machine: operands and temporary results are passed in a stack.

The stack is simply a list of values:

$$\text{Stack} = [\text{Value}]$$

$$\begin{array}{rl} [\,] & \text{empty stack} \\ v : vs & v \text{ top of stack, } \; vs \text{ rest of stack} \end{array}$$

Values are either *integers* or *addresses of closures*:

$$\begin{array}{rcl} v \in \text{Value} & = & n \in \text{Int} \\ & | & a \in \text{Addr} \end{array}$$

Dump

The dump is list of triples (s, e, c):

$$\text{Dump} = [(\text{Stack}, \text{Env}, \text{Code})]$$

Records the machine registers during function calls.

Pseudo instructions

LD n	load variable
LDC n	load constant
LDF c	load function
LDRF c	load recursive function
AP	apply
RTN	return
SEL c c'	select zero/non-zero
JOIN	join main control
ADD	add
SUB	subtract
MUL	multiply
HALT	halt execution

Note: the SECD described in Henderson's books has more instructions.

Translation examples

$1 + (2 \times 3)$ — [LDC 1, LDC 2, LDC 3, MUL, ADD]

$\lambda x.\, x + 1$ — [LDF [LD 0, LDC 1, ADD, RTN]]

$\lambda x.$ ifzero x 1 0

```
[LDF [LD 0,
        SEL [LDC 1, JOIN]
              [LDC 0, JOIN],
        RTN]]
```

Compiling and execution

The compiler is a function

$$\text{compile} :: \text{Term} \rightarrow \text{Symtable} \rightarrow \text{Code}$$

The symbol table is a list of identiers; each identifier is associated to its index on the list.

$$\text{Symtable} = [\mathit{Ident}]$$

The execution of each instruction is defined by state transition:

$$\underbrace{\langle s, e, c, d, m \rangle}_{\text{current config}} \longrightarrow \underbrace{\langle s', e', c', d', m' \rangle}_{\text{next config}}$$

Variables, constants and aritmetic operations

$$\text{compile } n \; sym = [\text{LDC } n]$$

$$\text{compile } x \; sym = [\text{LD } i] \quad \text{where } i = \text{elemIndex } x \; sym$$

$$\begin{aligned}\text{compile } (e_1 + e_2) \; sym &= \text{compile } e_1 \; sym +\!\!+ \text{compile } e_2 \; sym \\ &\quad +\!\!+ [\text{ADD}]\end{aligned}$$

$$\begin{aligned}\text{compile } (e_1 - e_2) \; sym &= \text{compile } e_1 \; sym +\!\!+ \text{compile } e_2 \; sym \\ &\quad +\!\!+ [\text{SUB}]\end{aligned}$$

$$\vdots$$

etc.

State transitions

$$\langle s,\, e,\, (\mathrm{LD}\ i) : c,\, d,\, m\rangle \longrightarrow \langle v_i : s,\, e,\, c,\, d,\, m\rangle\,,$$
$$\text{where } e = [v_0,\, v_1,\, \ldots,\, v_i,\, \ldots]$$

$$\langle s,\, e,\, (\mathrm{LDC}\ n) : c,\, d,\, m\rangle \longrightarrow \langle n : s,\, e,\, c,\, d,\, m\rangle$$

$$\langle v_2 : v_1 : s,\, e,\, \mathrm{ADD} : c,\, d,\, m\rangle \longrightarrow \langle (v_1 + v_2) : s,\, e,\, c,\, d,\, m\rangle$$

$$\langle v_2 : v_1 : s,\, e,\, \mathrm{SUB} : c,\, d,\, m\rangle \longrightarrow \langle (v_1 - v_2) : s,\, e,\, c,\, d,\, m\rangle$$

$$\langle v_2 : v_1 : s,\, e,\, \mathrm{MUL} : c,\, d,\, m\rangle \longrightarrow \langle (v_1 \times v_2) : s,\, e,\, c,\, d,\, m\rangle$$

Abstraction and application

$\lambda x.\, e$

1. Builds a new closure;
2. Pushes the address onto the top of the stack.

$(e_1\ e_2)$

1. Evaluates e_1 obtaining a closure;
2. Evaluates e_2 obtaining the argument value;
3. Records the execution context on the dump;
4. Executes the code in the closure;
5. Recovers the execution context from the dump.

Compilation

$$\text{compile } (\lambda x.\, e)\; sym = [\text{LDF } (\text{compile } e\; sym' +\!\!+ [\text{RTN}])]$$
$$\text{where } sym' = \text{extend } sym\; x$$

$$\text{compile } (e_1\ e_2)\; sym = \text{compile } e_1\; sym +\!\!+ \text{compile } e_2\; sym +\!\!+ [\text{AP}]$$

State transitions

$$\langle s,\ e,\ (\mathrm{LDF}\ c')\ :\ c,\ d,\ m\rangle \longrightarrow \langle a : s,\ e,\ c,\ d,\ m[a \mapsto (c', e)]\rangle$$
$$\text{where } a = \text{next } m$$

$$\langle v : a : s,\ e,\ \mathrm{AP} : c,\ d,\ m\rangle \longrightarrow \langle [\,],\ v : e',\ c',\ (s, e, c) : d,\ m\rangle$$
$$\text{if } m(a) = (c', e')$$

$$\langle v : s,\ e,\ \mathrm{RTN} : c,\ (s', e', c') : d,\ m\rangle \longrightarrow \langle v : s',\ e',\ c',\ d,\ m\rangle$$

Conditional

ifzero e_0 e_1 e_2

1. Evaluates e_0; the result should be an integer;
2. Record the execution context on the dump;
3. If the top of the stack is é 0 evaluate e_1; othewise, evaluate e_2;
4. Recover the evaluation context from the dump.

Compilation

$$
\begin{aligned}
\text{compile } (\textbf{if } e_0\ e_1\ e_2)\ \mathit{sym} &= \text{compile } e_0\ \mathit{sym} \mathbin{+\!\!+} [\text{SEL } c_1\ c_2] \\
\text{where } c_1 &= \text{compile } e_1\ \mathit{sym} \mathbin{+\!\!+} [\text{JOIN}] \\
c_2 &= \text{compile } e_2\ \mathit{sym} \mathbin{+\!\!+} [\text{JOIN}]
\end{aligned}
$$

State transitions

$$\langle 0 : s,\ e,\ (\mathrm{SEL}\ c_1\ c_2) : c,\ d,\ m\rangle \longrightarrow \langle s,\ e,\ c_1,\ ([], [], c) : d,\ m\rangle$$

$$\langle v : s,\ e,\ (\mathrm{SEL}\ c_1\ c_2) : c,\ d,\ m\rangle \longrightarrow \langle s,\ e,\ c_2,\ ([], [], c) : d,\ m\rangle$$
$$\text{if } v \in \mathrm{Int} \land v \neq 0$$

$$\langle s,\ e,\ \mathrm{JOIN} : c,\ (\_, \_, c') : d,\ m\rangle \longrightarrow \langle s,\ e,\ c',\ d,\ m\rangle$$

Local definitions

Simple solution: translate into λ-abstraction plus application.

$$\text{compile } (\textbf{let } x = e_1 \textbf{ in } e_2)\ sym = \text{compile } ((\lambda x.\, e_2)\ e_1)\ sym$$

Better alternative: exercise II (see Henderson and Kogge).

Compiling fixed point operator

$$
\begin{aligned}
\text{compile } (\textbf{fix } \lambda f.\, \lambda x.\, e)\ \mathit{sym} \;&=\; [\text{LDRF } (\text{compile } e\ \mathit{sym}' +\!\!+ [\text{RTN}])] \\
&\quad \text{where } \mathit{sym}' = \text{extend } (\text{extend } \mathit{sym}\ f)\ x
\end{aligned}
$$

State transition

Builds a cyclic closure:

$$\langle s,\, e,\, (\text{LDRF } c') : c,\, d,\, m \rangle \longrightarrow \langle a : s,\, e,\, c,\, d,\, m' \rangle$$

$$\begin{aligned} \text{where} \quad & a = \mathit{next}(m) \\ & m' = m[a \mapsto (c', a : e)] \end{aligned}$$

Note: the machine presented in Henderson's book uses two instruções (DUM/RAP) to build the cyclic closure.

Exercise I

Translate into SECD machine instructions:

$$\textbf{let } x = 42 \textbf{ in } 2 * x \tag{1}$$

$$\lambda x.\, \lambda y.\, \textbf{ifzero } (x - y)\ 1 \textbf{ else } 0 \tag{2}$$

$$\begin{aligned} \textbf{let } \mathit{fib} = \textbf{fix } \lambda f.\, \lambda n.\quad & \textbf{ifzero } (n-1)\ 1 \\ & (\textbf{ifzero } (n-2)\ 1 \\ & \quad (f\ (n-1) + f\ (n-2))) \\ \textbf{in } \mathit{fib}\ 3 & \end{aligned} \tag{3}$$

Exercise II

Add a machine instruction AA ("add argument") to move an argument from the temporary stack to the environment:

$$\langle v : s,\ e,\ \mathrm{AA} : c,\ d,\ m\rangle \longrightarrow \langle s,\ v : e,\ c,\ d,\ m\rangle$$

Use this instruction to compile **let** $x = e_1$ **in** e_2 more efficiently.

Exercise III

Modify the SECD interpreter to keep track of the maximum stack and dump sizes.

More information

Lots of information and implementations on the *web*. . .

- Wikipedia:
  `https://en.wikipedia.org/wiki/SECD_machine`
- *SECD mania*: `http://skelet.ludost.net/sec`
- *A Rational Deconstruction of Landin's SECD Machine*, Olivier Danvy, BRICS research report