3.36pt

A máquina SECD

February 25, 2019

O que é a máquina SECD?

- Um interpretador da linguagem funcional ISWIM (Landin, 1964)
- Máquina virtual para compilação LISP/Scheme (Henderson, 1980)
- Utilizada em implementações reais (LispMe no Palm Pilot)
- Desenhada para linguagens *call-by-value*
- Pode ser modificada para *lazy evaluation* (embora existam alternativas mais eficientes)

Máquina abstracta ou virtual?

- A SECD original interpreta directamente termos de sintaxe abstracta (máquina abstracta)
- Vamos apresentar uma máquina que interpreta pseudo-instruções (máquina virtual)
- Omitimos:
  1. estruturas de dados (listas, tuplos, etc.);
  2. escolha de representações concretas em memória;
  3. tradução das pseudo-instruções para código-máquina real;
  4. ambiente necessário para execução: alocação de memória, *garbage collection*, I/O. . .

Bibliografia

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

SECD: Stack, Environment, Control & Dump

A configuração da máquina é um quinteto

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

- s pilha de valores temporários (*stack*);
- e pilha de valores das variáveis livres (*environment*);
- c sequência de instruções (*control*);
- d pilha de continuações (*dump*);
- m memória (*closures*).

Resolução de nomes

Durante compilação vamos associar nomes de variáveis a *índices* no ambiente.

Interpretador

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

Compilador

$$\left.\begin{array}{rl} \text{termo:} & x + y \\ \text{tabela de símbolos:} & [x \mapsto 0, y \mapsto 1] \end{array}\right\} \text{compilação}$$

$$\left.\begin{array}{rl} \text{código gerado:} & [\text{LD } 0,\ \text{LD } 1,\ \text{ADD}] \\ \text{ambiente:} & [23, 42] \end{array}\right\} \text{execução}$$

Notação de De Bruijn

Identifica as variáveis pela profundidade do ligador λ:

$$\begin{array}{cccccc} \lambda x. & (\lambda y. & y & x) & x \\ \lambda & (\lambda & 1 & 2) & 1 \end{array}$$

Os ambiente passam a ser apenas listas de valores:

$$[v_1, v_2, \ldots, v_i, \ldots, v_n]$$

Cada variável é associada a um índice i.

Closures

Valores funcionais são representados por *closures*, e.g.

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

Na máquina SECD os λ-termos são traduzido para código compilado:

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

Closures

Representamos a memória como uma função parcial que associa endereços a *closures*:

$$\mathrm{Store} = \mathrm{Addr} \rightarrow \mathrm{Closure}$$

A função next dá o próximo endereço livre:

$$\mathrm{next} :: \mathrm{Store} \rightarrow \mathrm{Addr}$$

Pilha de temporários

Os operandos e resultado de instruções são passados na pilha de temporários.

A pilha é uma lista de valores:

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

$$\begin{array}{rl} [\,] & \text{pilha vazia} \\ v : vs & v \text{ topo da pilha, } vs \text{ resto da pilha} \end{array}$$

Os valores são inteiros ou endereços de *closures*:

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

Pilha de continuações

A pilha de continuações é uma lista de trios (s, e, c):

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

Guarda temporariamente os registos da máquina durante a chamada de funções.

Conjunto de pseudo-instruções

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

Nota: a SECD descrita no livro de Henderson tem mais instruções.

Exemplos de compilação

$1 + (2 \times 3)$	[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]]
```

Compilação e execução de instruções

O compilador é uma função

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

A tabela de símbolos é uma lista; cada identificador é associado ao seu índice na lista.

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

Cada instrução é definida por uma transição de estado:

$$\underbrace{\langle s, e, c, d, m\rangle}_{\text{configuração actual}} \longrightarrow \underbrace{\langle s', e', c', d', m'\rangle}_{\text{configuração seguinte}}$$

Variáveis, constantes e operações aritméticas

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

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

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

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

$$\vdots$$

etc.

Execução

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

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

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

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

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

Abstração e aplicação

$\lambda x.\, e$

1. Constrói uma nova *closure*;
2. Deixa o endereço do resultado na pilha.

$(e_1\ e_2)$

1. Avalia e_1 (obtém uma *closure*);
2. Avalia e_2 (obtém o valor do argumento);
3. Guarda o contexto de execução na *dump*;
4. Executa o código da *closure*;
5. Recupera o contexto de execução.

Compilação

$$
\begin{aligned}
\text{compile } (\lambda x.\, e)\ sym &= [\text{LDF } (\text{compile } e\ sym' +\!\!+ [\text{RTN}])] \\
&\qquad \text{onde } sym' = \text{extend } sym\ x
\end{aligned}
$$

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

Execução

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

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

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

Condicional

$$\textbf{ifzero}\ e_0\ e_1\ e_2$$

1. Avalia e_0 (resultado deve ser um inteiro);
2. Guarda o contexto de execução na *dump*
3. Se topo da pilha é 0 avalia e_1; caso contrário, avalia e_2;
4. Recupera o contexto de execução guardado.

Compilação

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

Execução

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

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

$$\text{se } v \in \mathrm{Int} \wedge v \neq 0$$

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

Definições locais

Solução simples: traduzir como uma aplicação.

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

Alternativa com optimização: ver os livros do Henderson e Kogge.

Compilação do operador ponto-fixo

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

Execução

Constrói uma *closure* cíclica:

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

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

Nota: a SECD apresentado no livro de Henderson usa duas instruções (DUM/RAP) para construir *closures* cíclicas.

Mais informações

Muita informação e implementações na *web*. . .

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

Exercícios (1)

Compilar para a máquina SECD e executar no interpretador:

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

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

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

Exercícios (2)

Modificar o interpretador da SECD para reportar o tamanho máximos da pilha.