Abstract machines for functional languages

Survey

October 21, 1999

1. Closures
2. The SECD machine
3. An abstract framework
   (a) The λ-calculus
   (b) The λ-calculus using De Bruijn indices
   (c) The λ-calculus with explicit substitutions
   (d) Bisimulation
4. The Krivine's machine
5. The CAM machine
6. The Krivine's machine with marks on the stack
7. The ZAM machine

Introduction

Why this survey?

- learn a functional language together with implementation issues
- illustrate what could some formalism bring
- give a framework for 'thinking' an abstract machine
- as a first step towards partial evaluation and run-time generation of specialized bytecode
- for general knowledge

Why these particular machines?

- The ZAM is the paper we survey
- The Krivine's machine is used there as a model
- The CAM because it is the historical implementation of CAML
- The SECD machine because it is the most basic machine

Run-time code generation of closures [LER98]

A naive approach:

- generate a block of code at run-time
- return a pointer to that code

```
let scale n = fun x -> x * n
```

```
let f = scale 2             let f = scale 10

res <- arg                  res <- arg
mul res, 2                  mul res, 10
return                      return
```

Problems:

- this is expensive
- it creates similar functions

Solution:

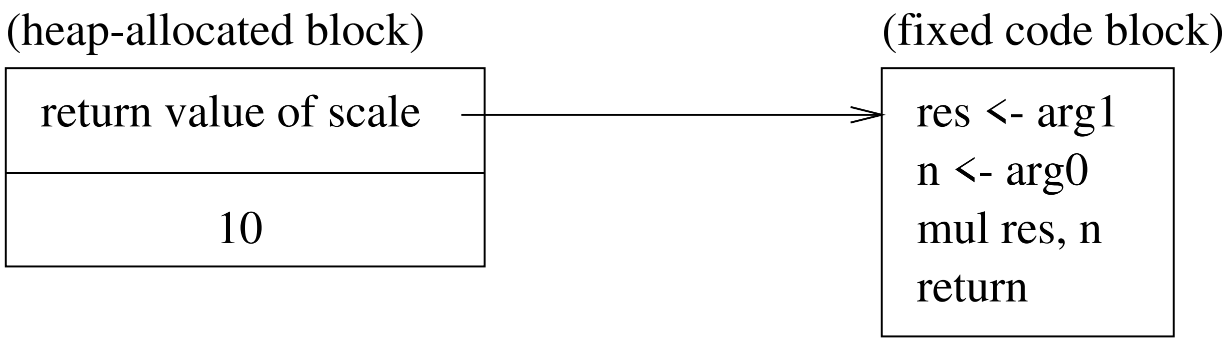
All generated blocks have the same shape, therefore we can share some common code and put the changing parts in some external structure. This can be seen as a tentative definition of a *closure*.

Representing functional values with closures

In practice a closure is a heap-allocated data structure containing:

- a pointer to a fixed piece of code computing the result
- an *environment* providing values for the free variables of the function

Practical view:



An interpreter of λ-expressions

The operational semantics of such an interpreter can be written as follows:

$$
\begin{aligned}
Eval[\![x]\!]\rho &= \rho x \\
Eval[\![\lambda x.E]\!]\rho &= \{x, E, \rho\} \\
Eval[\![F\ A]\!]\rho &= (Eval[\![F]\!]\rho)\ (Eval[\![A]\!]\rho)
\end{aligned}
$$

where

$$(\{x, E, \rho\})(v) = Eval[\![E]\!]\rho[x \leftarrow v]$$

and

- $Eval[\![\ ]\!]$ is the semantics function
- ρ's are environments memorising values
- $\{x, E, \rho\}$'s are closures of a function whose:
  - body is E
  - environment is ρ
  - formal parameter is x

Such an interpreter can be built as a state-transition machine.

The SECD machine [LAN64]

A $state$ is an object which consists of:

1. (S) a $stack$, a list of intermediate results
2. (E) an $environment$, a list of tame/values pairs;
3. (C) a $control$, a list of expressions and application tags (@)
4. (D) a $dump$, a complete state.

A transition is:

1. If $C = \emptyset$ and $D = (S', E', C', D')$, then
   $(S, E, C, D) \leftarrow (hS :: S', E', C', D')$
2. If $C \neq \emptyset$, then hC is inspected:
   (a) if $hC = x$, then $S \leftarrow Ex :: S$ and $C \leftarrow tC$
   (b) if $hC = \lambda x.F$, then $S \leftarrow \{x, F, E\} :: S$ and $C \leftarrow tC$
   (c) if $hC = @$, S changes as follows:
      i. if $hS = \{x, X', E'\}$, then
         $S \leftarrow []$
         $E \leftarrow E'[x \leftarrow thS]$
         $C \leftarrow X'$
         $D \leftarrow (t(tS), E, tC, D)$
      ii. o.w., $S \leftarrow ((hS)(htS) : t(tS))$
   (d) if $hC = F\ A$, then $C \leftarrow A :: F :: @ :: tC$

An abstract framework [HAR96]

Even though the λ-calculus is widely accepted as an archetypal language of all functional programming languages, there is no consensus about an archetypal target language.

Two wide classes of compilers for functional languages are:

- those that compile to (super)combinator terms;
- those that compile to abstract machines (SECD, CAM, ZINC).

Nevertheless, a few common aspects arise here.

- the functions are to be compiled
- the values of free variables remain the same at every function call

Thus, the two low-level objects that represents a function are:

- a $closure$ which is a pair of code and of
- an $environment$ that collects the values of the free variables

Therefore, our universal target language should be a calculus of closures.

The λ-calculus

As a first tentative compilation scheme, the β-reduction can be partially compiled by an algorithm introducing environments:

$$\begin{aligned}
Abs[\![x]\!] &= \lambda e.e\ x \\
Abs[\![\lambda x.E]\!] &= \lambda e.\lambda y.Abs[\![E]\!]\ (e[y/x]) \\
Abs[\![F\ A]\!] &= \lambda e.(Abs[\![F]\!]\ e)(Abs[\![A]\!]\ e)
\end{aligned}$$

For instance:

$$Abs[\![\lambda x.\lambda z.z\ x]\!]\ []\ A$$

Compilation:

$$\begin{array}{l}
(\lambda e.\lambda y.Abs[\![\lambda z.z\ x]\!](e[y/x]))[]A \\
(\lambda e.\lambda y.(\lambda e.\lambda y.(Abs[\![z\ x]\!])(e[y/z]))(e[y/x]))[]A \\
(\lambda e.\lambda y.(\lambda e.\lambda y.(\lambda e.(Abs[\![z]\!]e)(Abs[\![x]\!]e))(e[y/z]))(e[y/x]))[]A \\
(\lambda e.\lambda y.(\lambda e.\lambda y.(\lambda e.(\lambda e.e\ z\ e)(\lambda e.e\ x\ e))(e[y/z]))(e[y/x]))[]A
\end{array}$$

Execution:

$$\begin{array}{l}
(\lambda e.\lambda y.(\lambda e.(\lambda e.e\ z\ e)(\lambda e.e\ x\ e))(e[y/z]))([][A/x]) \\
\lambda y.(\lambda e.(\lambda e.e\ z\ e)(\lambda e.e\ x\ e))([][A/x][y/z])
\end{array}$$

The λ-calculus can be used with success to describe implementations.

De Bruijn indices

Here are illustrations of Bourbaki's style notations:

$$\begin{array}{ll} \lambda x.x & \lambda 1 \\ \lambda x.x\lambda y.xy & \lambda 1\lambda 21 \\ \lambda x\lambda y\lambda z.x & \lambda\lambda\lambda 3 \\ \lambda y.(\lambda xy.x)y & \lambda(\lambda\lambda 2)1 \\ \lambda x.\lambda y.xy & \lambda\lambda 21 \end{array}$$

This gave rise to the De Bruijn notation where variable occurrences are replaced with positive integers called De Bruijn indices.

This solves problems of name clash.

For instance, $(\lambda xy.x)y$ by no way reduces to $\lambda y.y$. Thanks to De Bruijn indices, one never has to change explicitely $\lambda xy.x$ into say $\lambda xz.x$ to avoid the name clash.

The λ-calculus using De Bruijn indices

The λ-expressions with De Bruijn indices can be reduced thanks to the following rules:

$$
\begin{array}{l}
(\lambda.M)N \longrightarrow \sigma_0(M, N) \\
\sigma_n(F\ A, N) \longrightarrow \sigma_n(F, N)\sigma_n(A, N) \\
\sigma_n(\lambda.F, N) \longrightarrow \lambda.(\sigma_{n+1}(F, N)) \\
\sigma_n(m, N) \rightarrow m - 1, \quad m > n \\
\sigma_n(m, N) \rightarrow \tau_n^0(N), \quad m = n \\
\sigma_n(m, N)m, \quad o.w.
\end{array}
$$

For instance, let us compile $\lambda x.\lambda z.z\ x$ applied to A:

$$
\begin{array}{l}
(\lambda.\lambda.0\ 1)A \\
\rightarrow \sigma_0(\lambda.0\ 1, A) \\
\rightarrow \lambda.(\sigma_1(0\ 1, A)) \\
\rightarrow \lambda.\sigma_1(0, A)\sigma_1(1, A) \\
\rightarrow \lambda.\lambda 0\sigma_1(1, A) \\
\rightarrow \lambda.0\tau_1^0(A) \\
\rightarrow \lambda.0A
\end{array}
$$

With this system, there is no environment, but a substitution by argument. The $\lambda\sigma$-calculus uses De Bruijn indices with explicit substitutions.

Explicit substitutions [ABA91]

The 'éminence grise' of λ-calculus:

$$(\lambda x.a)b \rightarrow_\beta a[b/x]$$

We introduce $\_[\_]$ as the substitution operator:

$$(\lambda x.a)b \rightarrow_{Beta} a[(b/x)\cdot id]$$

A tentative explicit substitution:

$$(\lambda x.a)b \rightarrow_\beta a[b/x] = (\lambda a)b \rightarrow_\beta a\{b/\underline{1}, \underline{1}/\underline{2}, \underline{2}/\underline{3}, \ldots\}$$

We shift($\Uparrow$) when coming across a λ-abstraction:

$$(\lambda c)\{b/\underline{1}, \underline{1}/\underline{2}, \underline{2}/\underline{3}, \ldots\} = \lambda c\{\underline{1}/\underline{1}, b/\underline{2}, \underline{2}/\underline{3}, \underline{3}/\underline{4}, \ldots\}$$

And lift($\uparrow$) indices inside b:

$$(\lambda c)\{b/\underline{1}, \underline{1}/\underline{2}, \ldots\} = \lambda c\{\underline{1}/\underline{1}, b\{\underline{2}/\underline{1}, \underline{3}/\underline{2}, \ldots\}/\underline{2}, \underline{2}/\underline{3}, \ldots\}$$

$\lambda\sigma$-calculus

The $\lambda\rho$-calculus gives a first-class status to closures. The $\lambda\sigma$-calculus is similar but more convenient.

$$\begin{array}{llll} \textit{Terms} & M & ::= & \mathtt{n} \mid (MM) \mid \lambda M \mid M[s] \\ \textit{Substitutions} & s & ::= & id \mid \uparrow \mid \Uparrow \mid M \cdot s \mid s \circ s \end{array}$$

For example, in the formalism of the $\lambda\sigma$-calculus, the three first rules of the λ-calculus with De Bruijn indices can be written:

$$\begin{array}{ll} (\lambda.M)N & \rightarrow M[N \cdot id] \\ (MN)[s] & \rightarrow (M[s])(N[s]) \\ (\lambda.M)[s] & \rightarrow \lambda(M[\Uparrow s]) \end{array}$$

Let us compile $(\lambda x.\lambda z.z\ x)A$ in the $\lambda\sigma$-calculus:

$$\begin{array}{l} (\lambda.\lambda.\overline{0}\ \overline{1})A \\ \rightarrow (\lambda.\overline{0}\ \overline{1})[A \cdot id] \\ \rightarrow \lambda.((\overline{0}\ \overline{1})[\Uparrow (A \cdot id)]) \\ \rightarrow \lambda.(\overline{0}[\Uparrow (A \cdot id)])(\overline{1}[\Uparrow (A \cdot id)]) \\ \rightarrow \lambda.\overline{0}(\overline{1}[\Uparrow (A \cdot id)]) \\ \vdots \\ \rightarrow \lambda.\overline{0}\ A \end{array}$$

The λ-calculus with explicit substitutions

We recall the definition of Λ_σ:

$$
\begin{array}{lccl}
\text{TERMS:} & M & ::= & \mathtt{n} \mid (MM) \mid \lambda M \mid M[s] \\
\text{SUBSTITUTIONS:} & s & ::= & id \mid \uparrow \mid M \cdot s \mid s \circ s
\end{array}
$$

The propagation of substitutions inside terms and substitutions is defined by the following rewriting system σ_w:

$$
\begin{array}{lrcl}
\text{App} & (M_1\ M_2)[s] & \rightarrow & (M_1[s]M_2[s]) \\
\text{FVar} & \mathtt{1}[M \cdot s] & \rightarrow & M \\
\text{RVar} & \mathtt{n+1}[M \cdot s] & \rightarrow & \mathtt{n}[s] \\
\text{Clos} & (M[s])[t] & \rightarrow & M[s \circ t] \\
\text{AssEnv} & (s \circ t) \circ u & \rightarrow & s \circ (t \circ u) \\
\text{MapEnv} & (M \cdot s) \circ t & \rightarrow & M[t] \cdot (s \circ t) \\
\text{ShiftCons} & \uparrow \circ (M \cdot s) & \rightarrow & s \\
\text{IdL} & id \circ s & \rightarrow & s
\end{array}
$$

In this setting (where $\lambda\sigma$-closures are weak values), $\lambda\sigma$-closures are destructed only by applying them to arguments:

$$
\text{Beta} \quad ((\lambda M_1)[s]M_2) \ \rightarrow \ M_1[M_2 \cdot s]
$$

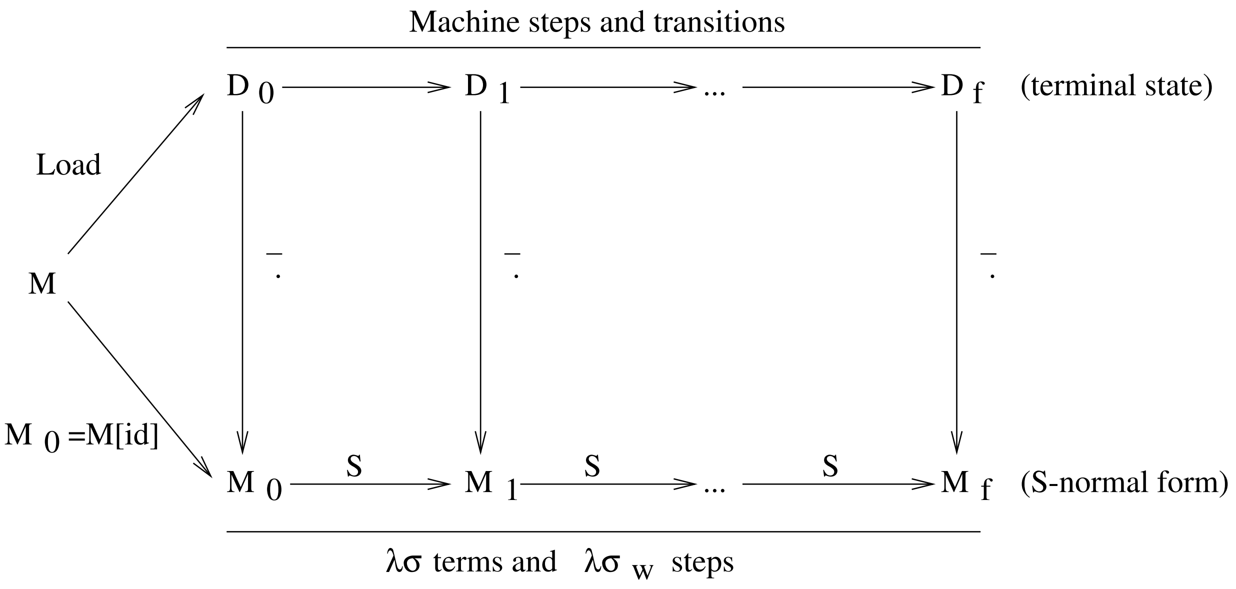
The system $\lambda\sigma_w$ is defined by the rules of σ_w plus the rule Beta.

Bisimulation

We say that a machine implements a strategy S when the following four conditions are satisfied:

1. *The inital condition*
2. *The machine follows the Strategy S*
3. *Terminal states translate to normal forms*
4. *Machine and strategy progreess at the same pace*

The machine can operate internal operations.



The Krivine's machine: 'hardware'

It is equipped with:

- a code pointer

$$\text{CODE} \quad ::= \quad () \mid \text{INSTRUCTION;CODE}$$

- a register holding the current environment

$$\begin{aligned} \text{CLOSURE} \quad &::= \quad (\text{CODE}/\text{ENVIRONMENT}) \\ \text{ENVIRONMENT} \quad &::= \quad () \mid \text{CLOSURE} \cdot \text{ENVIRONMENT} \end{aligned}$$

- a stack holding frames (growing from right to left)

$$\text{STATE} \quad ::= \quad \text{FRAME} \mid \text{FRAME} :: \text{STATE}$$

The Krivine's machine: compilation and decompilation

Instructions:

$$
\begin{array}{lcl}
\mathit{Instruction} & ::= & \mathsf{Grab} \mid \mathsf{Push}(\textsc{code}) \mid \mathsf{Access}(n) \\
\mathit{Frame} & ::= & \textsc{closure}
\end{array}
$$

Compilation scheme:

$$
\begin{array}{rcl}
[\![\mathtt{n}]\!] & = & \mathsf{Access}(n) \\
[\![\lambda N]\!] & = & \mathsf{Grab};\, [\![N]\!] \\
[\![(N_1\ N_2)]\!] & = & \mathsf{Push}([\![N_2]\!]);\, [\![N_1]\!]
\end{array}
$$

Decompilation scheme:

$$
\begin{array}{rcl}
\overline{\mathsf{Access}(n)} & = & \mathtt{n} \\
\overline{\mathsf{Grab};\, C} & = & \lambda \overline{C} \\
\overline{\mathsf{Push}(C');\, C} & = & (\overline{C}\,\overline{C'}) \\
\overline{(C/e)} & = & \overline{C}[\overline{e}] \\
\overline{()} & = & id \\
\overline{f \cdot e} & = & \overline{f} \cdot \overline{e} \\
\overline{f_n :: f_{n-1} :: \cdots :: f_1} & = & (\cdots(\overline{f_n}\ \overline{f_{n-1}}) \cdots \overline{f_1})
\end{array}
$$

The Krivine's machine: execution

Transitions:

$$
\begin{array}{rcl}
(\mathsf{Access}(1)/(C_0/e_0)\cdot e) :: D & \to^{\mathsf{lvar}} & (C_0/e_0) :: D \\
(\mathsf{Access}(n+1)/(C_0/e_0)\cdot e) :: D & \to^{\mathsf{rvar}} & (\mathsf{Access}(n)/e) :: D \\
(\mathsf{Push}(C'); C/e) :: D & \to^{\mathsf{push}} & (C/e) :: (C'/e) :: D \\
(\mathsf{Grab}; C/e) :: (C'/e') :: D & \to^{\mathsf{grab}} & (C/(C'/e')\cdot e) :: D
\end{array}
$$

Example:

$$
\begin{array}{l}
(\mathtt{P(X);G;G;P(A1);A2}/\emptyset) :: \emptyset \\
(\mathtt{G;G;P(A1);A2}/\emptyset) :: (\mathtt{X}/\emptyset) :: \emptyset \\
(\mathtt{G;P(A1);A2}/(\mathtt{X}/\emptyset)) :: \emptyset \\
(\mathtt{P(A1);A2}/(\mathtt{X}/\emptyset)) :: \emptyset \\
(\mathtt{A2}/(\mathtt{X}/\emptyset)) :: (\mathtt{A1}/(\mathtt{X}/\emptyset)) :: \emptyset \\
(\mathtt{A1}/\emptyset) :: (\mathtt{A1}/(\mathtt{X}/\emptyset)) :: \emptyset
\end{array}
$$

Correctness of the Krivine's machine

$(\mathsf{Access}(1)/(C_0/e_0)\cdot e) :: D_0$	$\longrightarrow^{\mathsf{lvar}}$	$(C_0/e_0) :: D_0$
$\overline{\cdot}\downarrow$		$\overline{\cdot}\downarrow$
$(\cdots(\mathtt{1}[\overline{C_0}[\overline{e_0}]\cdot\overline{e}])\cdots\overline{f_1})$	$\longrightarrow^{\lambda\sigma w}$	$(\cdots(\overline{C_0}[e_0])\cdots\overline{f_1})$
$(\mathsf{Access}(n+1)/(C_0/e_0)\cdot e) :: D_0$	$\longrightarrow^{\mathsf{lvar}}$	$(\mathsf{Access}(n)/e) :: D_0$
$\overline{\cdot}\downarrow$		$\overline{\cdot}\downarrow$
$(\cdots(\mathtt{n}+\mathtt{1}[\overline{C_0}[\overline{e_0}]\cdot\overline{e}])\cdots\overline{f_1})$	$\longrightarrow^{\lambda\sigma w}$	$(\cdots(\mathtt{n}[\overline{e}])\cdots\overline{f_1})$
$(\mathsf{Push}(C');C/e) :: D_0$	$\longrightarrow^{\mathsf{push}}$	$(C/e) :: (C'/e) :: D_0$
$\overline{\cdot}\downarrow$		$\overline{\cdot}\downarrow$
$(\cdots(\overline{C C'})[\overline{e}]\cdots\overline{f_1})$	$\longrightarrow^{\lambda\sigma w}$	$(\cdots(\overline{C}[\overline{e}]\overline{C'}[\overline{e}])\cdots\overline{f_1})$
$(\mathsf{Grab};C/e) :: (C'/e') :: D_0$	$\longrightarrow^{\mathsf{grab}}$	$(C/(C'/e')\cdot e) :: D_0$
$\overline{\cdot}\downarrow$		$\overline{\cdot}\downarrow$
$(\cdots(\lambda\overline{C}[\overline{e}]\overline{C'}[\overline{e'}])\cdots\overline{f_1})$	$\longrightarrow^{\lambda\sigma w}$	$(\cdots(\overline{C}[\overline{C'}[\overline{e'}]\cdot\overline{e}])\cdots\overline{f_1})$

The CAM machine: introduction [COU85]

```
let x = plus in x(4,(x where x=3));;
```

$$
\begin{array}{rcl}
[\![1]\!] & = & \mathsf{Snd} \\
[\![\mathtt{n}+1]\!] & = & \mathsf{Fst}[\![\mathtt{n}]\!] \\
[\![(N_1\ N_2)]\!] & = & \langle [\![N_1]\!], [\![N_2]\!] \rangle \mathsf{App} \\
[\![\lambda N]\!] & = & \Lambda([\![N]\!])
\end{array}
$$

$$
\begin{array}{rcl}
M & = & (\lambda x.x(4,(\lambda x.x)3)+ \\
 & = & (\lambda \underline{1}(4,(\lambda \underline{1})3))+ \\
 & = & \langle \lambda \underline{1}(4,(\lambda \underline{1})3), + \rangle \mathsf{App} \\
 & = & \langle \Lambda(\underline{1}(4,(\lambda \underline{1})3)), + \rangle \mathsf{App} \\
 & = & \langle \Lambda(\langle \underline{1},(4,(\lambda \underline{1})3)) \rangle \mathsf{App}, + \rangle \mathsf{App} \\
 & = & \langle \Lambda(\langle \underline{1}, \langle 4, \langle (\lambda \underline{1}), 3 \rangle \mathsf{App} \rangle \rangle \mathsf{App}), + \rangle \mathsf{App} \\
 & = & \langle \Lambda(\langle \underline{1}, \langle '4, \langle (\lambda \underline{1}), '3 \rangle \mathsf{App} \rangle \rangle \mathsf{App}), + \rangle \mathsf{App} \\
 & = & \langle \Lambda(\langle \mathsf{Snd}, \langle '4, \langle (\lambda \mathsf{Snd}), '3 \rangle \mathsf{App} \rangle \rangle \mathsf{App}), + \rangle \mathsf{App} \\
 & = & \langle \Lambda(\langle \mathsf{Snd}, \langle '4, \langle (\Lambda \mathsf{Snd}), '3 \rangle \mathsf{App} \rangle \rangle \mathsf{App}), + \rangle \mathsf{App} \\
 & = & \langle \Lambda(\langle \mathsf{Snd}, \langle '4, \langle (\Lambda \mathsf{Snd}), '3 \rangle \mathsf{App} \rangle \rangle \mathsf{App}), \mathsf{Snd}+ \rangle \mathsf{App} \\
 & = & \langle \Lambda(A), \Lambda(\mathsf{Snd}+) \rangle \mathsf{App}
\end{array}
$$

where $A = \langle \mathsf{Snd}, \langle '4, B \rangle \rangle \mathsf{App}$ and $B = \langle \Lambda(\mathsf{Snd}, '3) \rangle \mathsf{App}$

(we assume ρ has the shape $(\cdots((), v_n) \cdots, v_0))$)

The CAM machine: compilation

The code of an application will:

- `push` the environment onto the stack
- `exec` the functional part
- `swap` the functional value and the saved environment
- `eval` the argument
- `cons` the resulting closure
- `apply` the argument to the function

```
let rec code_of = function
    ...
  |     Const(n) -> [Quote(n)]
  |     App(e1,e2) -> push::(code_of e1)@[swap]
                      @(code_of e2)@[cons]@[app]
  | ...
```

The CAM has eight instructions:

Fst	Snd	$<$	,	$>$	App	Λ	$'$
car	cdr	push	swap	cons	app	cur	quote

The CAM machine: execution

The transition of the CAM are as follows:

term	code	stack	term	code	stack
(s,t)	car.C	S	s	C	S
(s,t)	cdr.C	S	t	C	S
s	(quote$c$).C	S	c	C	S
s	(cur$C$).$C1$	S	$(C:s)$	$C1$	S
s	push.C	S	s	C	$s.S$
t	swap.C	$s.S$	s	C	$t.S$
t	cons.C	$s.S$	(s,t)	C	S
$(C:s,t)$	app.$C1$	S	(s,t)	$C@C1$	S

```
let step state = match state with
    ...
  |     {term=s; code=quote c::C; stack=S} ->
                state.term <- c; state.code <- C
  |     {term=s; code=cur C::C1; stack=S} ->
                state.term <- Closure(C,s);
                state.code <- C1
  |     {term=s; code=push::C; stack=S} ->
                state.code <- C; state.stack <- s::S
  |     {term=t; pc=swap::C; stack=s::S} ->
                state.term <- s; state.code <- code;
                state.stack <- t::S
```

The CAM machine: example

$()$	$\langle\Lambda(A),\Lambda(\mathsf{Snd}+)\rangle\mathsf{App}$	$[]$
$()$	$\Lambda(A),\Lambda(\mathsf{Snd}+)\rangle\mathsf{App}$	$[()]$
$A:()$	$,\Lambda(\mathsf{Snd}+)\rangle\mathsf{App}$	$[()]$
$()$	$\Lambda(\mathsf{Snd}+)\rangle\mathsf{App}$	$[A:()]$
$(\mathsf{Snd}+):()$	$\rangle\mathsf{App}$	$[A:()]$
$(A:(),(\mathsf{Snd}+):())$	App	$[]$
$((),(\mathsf{Snd}+):())$	$\langle\mathsf{Snd},\langle'4,B\rangle\rangle\mathsf{App}$	$[]$
$((),(\mathsf{Snd}+):())$	$\mathsf{Snd},\langle'4,B\rangle\rangle\mathsf{App}$	$[((),(\mathsf{Snd}+):())]$
$(\mathsf{Snd}+):()$	$,\langle'4,B\rangle\rangle\mathsf{App}$	$[((),(\mathsf{Snd}+):())]$
$((),(\mathsf{Snd}+):())$	$\langle'4,B\rangle\rangle\mathsf{App}$	$[(\mathsf{Snd}+):()]$
$((),(\mathsf{Snd}+):())$	$'4,B\rangle\rangle\mathsf{App}$	$[((),(\mathsf{Snd}+):());(\mathsf{Snd}+):()]$
4	$,B\rangle\rangle\mathsf{App}$	$[((),(\mathsf{Snd}+):());(\mathsf{Snd}+):()]$
$((),(\mathsf{Snd}+):())$	$B\rangle\rangle\mathsf{App}$	$[4;(\mathsf{Snd}+):()]$

The ZAM machine: principle designs [LER90]

To lower the interpretative overhead:

- C as an implementation language
- Specialized bytecode
- Stack machine
- Threaded code interpretation

Curried rather than N-ary functions

- For convenience:

```
List.map (g 4) [1; 2; 3; 4; 5];;
List.map (fun y -> f(4,y)) [1; 2; 3; 4; 5];;
```

- For partial evaluation:

```
let h x = let z = fib x in fun y -> y + z;;
List.map (h 30) [1; 2; 3; 4; 5];;
```

	lazy	strict
without spine	Lazy CAM: $2k - 1$ closures	Strict CAM: $k - 1$ closures
with spine	Krivine's machine k closures	ZAM 0 closures?

The Krivine's machine with marks on the stack

Eager compilation of applications: $[\![(M\ \ N)_s]\!] = \text{Reduce}([\![N]\!]); [\![M]\!]$

$$
\begin{aligned}
\overline{\text{Reduce}(c_2); c_1} &= (\overline{c_1 c_2}) \\
\overline{(c/e) :: (c_0/e_0) :: s} &= (\cdots(\overline{c}[\overline{e}]\overline{c_0}[\overline{e_0}])\cdots\overline{f_1}) \\
\overline{(c/e) :: \langle c_0/e_0\rangle :: s} &= (\cdots(\overline{c_0}[\overline{e_0}]\overline{c}[\overline{e}])\cdots\overline{f_1})
\end{aligned}
$$

$$
\begin{array}{rll}
(\text{Access}(1)/(C_0/e_0)\cdot e) :: D & \rightarrow^{\text{lvar}} & (C_0/e_0) :: D \\
(\text{Access}(n+1)/(C_0/e_0)\cdot e) :: D & \rightarrow^{\text{rvar}} & (\text{Access}(n)/e) :: D \\
(\text{Push}(C'); C/e) :: D & \rightarrow^{\text{push}} & (C/e) :: (C'/e) :: D \\
(\text{Grab}; C/e) :: (C'/e') :: D & \rightarrow^{\text{grab}} & (C/(C'/e')\cdot e) :: D \\
(\text{Grab}; C/e) :: \langle C'/e'\rangle :: D & \rightarrow^{\text{grab2}} & (C'/e') :: (\text{Grab}; C/e) :: D \\
(\text{Reduce}(C'); C/e) :: D & \rightarrow^{\text{red}} & (C'/e) :: \langle C/e\rangle :: D
\end{array}
$$

Krivine's machine: example

Without marked closures: $(\lambda((\lambda(\underline{1}\ \underline{2}))(\lambda\underline{1})))(\lambda\underline{1})$

$\texttt{(P(G;A1);G;P(G;A1);G;P(A2);A1/}\emptyset\texttt{)} :: \emptyset$
$\texttt{(G;P(G;A1);G;P(A2);A1/}\emptyset\texttt{)} :: \texttt{(G;A1/}\emptyset\texttt{)} :: \emptyset$
$\texttt{(P(G;A1);G;P(A2);A1/(G;A1/}\emptyset\texttt{))} :: \emptyset$
$\texttt{(G;P(A2);A1/(G;A1/}\emptyset\texttt{))} :: \texttt{(G;A1/(G;A1/}\emptyset\texttt{))} :: \emptyset$
$\texttt{(P(A2);A1/(G;A1/(G;A1/}\emptyset\texttt{))} \cdot \texttt{(G;A1/}\emptyset\texttt{))} :: \emptyset$
$\texttt{(A1/(G;A1/(G;A1/}\emptyset\texttt{))} \cdot \texttt{(G;A1/}\emptyset\texttt{))} :: \texttt{(A2/(G;A1/(G;A1/}\emptyset\texttt{))} \cdot \texttt{(G;A1/}\emptyset\texttt{))} :: \emptyset$
$\texttt{(G;A1/(G;A1/}\emptyset\texttt{))} :: \texttt{(A2/(G;A1/(G;A1/}\emptyset\texttt{))} \cdot \texttt{(G;A1/}\emptyset\texttt{))} :: \emptyset$
$\texttt{(A1/(A2/(G;A1/(G;A1/}\emptyset\texttt{))} \cdot \texttt{(G;A1/}\emptyset\texttt{))} \cdot \texttt{(G;A1/}\emptyset\texttt{))} :: \emptyset$
$\texttt{(A2/(G;A1/(G;A1/}\emptyset\texttt{))} \cdot \texttt{(G;A1/}\emptyset\texttt{))} :: \emptyset$
$\texttt{(G;A1/}\emptyset\texttt{)} :: \emptyset$

Krivine's machine with marks on the stack: example

With marked closures: $(\lambda((\lambda(\underline{1}\ \underline{2}))(\lambda\underline{1})))(\lambda\underline{1})$

$(\mathtt{R(G;A1);G;R(G;A1);G;R(A2);A1/}\emptyset) :: \emptyset$
$(\mathtt{G;A1/}\emptyset) :: \langle\mathtt{G;R(G;A1);G;R(A2);A1/}\emptyset\rangle :: \emptyset$
$(\mathtt{G;R(G;A1);G;R(A2);A1/}\emptyset) :: (\mathtt{G;A1/}\emptyset) :: \emptyset$
$(\mathtt{R(G;A1);G;R(A2);A1/}(\mathtt{G;A1/}\emptyset)) :: \emptyset$
$(\mathtt{G;A1/}(\mathtt{G;A1/}\emptyset)) :: \langle\mathtt{G;R(A2);A1/}(\mathtt{G;A1/}\emptyset)\rangle :: \emptyset$
$(\mathtt{G;R(A2);A1/}(\mathtt{G;A1/}\emptyset)) :: (\mathtt{G;A1/}(\mathtt{G;A1/}\emptyset)) :: \emptyset$
$(\mathtt{R(A2);A1/}(\mathtt{G;A1/}(\mathtt{G;A1/}\emptyset)) \cdot (\mathtt{G;A1/}\emptyset)) :: \emptyset$
$(\mathtt{A2/}(\mathtt{G;A1/}(\mathtt{G;A1/}\emptyset)) \cdot (\mathtt{G;A1/}\emptyset)) :: \langle\mathtt{A1/}(\mathtt{G;A1/}(\mathtt{G;A1/}\emptyset)) \cdot (\mathtt{G;A1/}\emptyset)\rangle :: \emptyset$
$(\mathtt{G;A1/}\emptyset) :: \langle\mathtt{A1/}(\mathtt{G;A1/}(\mathtt{G;A1/}\emptyset)) \cdot (\mathtt{G;A1/}\emptyset)\rangle :: \emptyset$
$(\mathtt{A1/}(\mathtt{G;A1/}(\mathtt{G;A1/}\emptyset)) \cdot (\mathtt{G;A1/}\emptyset)) :: (\mathtt{G;A1/}\emptyset) :: \emptyset$
$(\mathtt{G;A1/}(\mathtt{G;A1/}\emptyset)) :: (\mathtt{G;A1/}\emptyset) :: \emptyset$
$(\mathtt{A1/}(\mathtt{G;A1/}\emptyset) \cdot (\mathtt{G;A1/}\emptyset)) :: \emptyset$
$(\mathtt{G;A1/}\emptyset) :: \emptyset$

The ZAM machine

It is equipped with:

- a code pointer

$$\textsc{code} \quad ::= \quad () \mid \textsc{instruction;code}$$

- an accumulater holding intermediate results
- a register holding the current environment

$$\begin{array}{rcl} \textsc{closure} & ::= & (\textsc{code}/\textsc{environment}) \\ \textsc{environment} & ::= & () \mid \textsc{closure} \cdot \textsc{environment} \\ & & \mid \textsc{constant} \cdot \textsc{environment} \end{array}$$

- an argument stack holding arguments to function calls
- a return stack holding unallocated closures

Applications

$$\mathcal{T}[\![(M\ \ N_1\ \ \cdots\ \ N_k)]\!] = \mathcal{C}[\![N_k]\!]; \mathsf{Push}; \cdots; \mathcal{C}[\![N_1]\!]; \mathsf{Push}; \mathcal{C}[\![M]\!]; \mathsf{Appterm}$$
$$\mathcal{C}[\![(M\ \ N_1\ \ \cdots\ \ N_k)]\!] = \mathsf{Pushmark}; \mathcal{C}[\![N_k]\!]; \mathsf{Push}; \cdots; \mathcal{C}[\![N_1]\!]; \mathsf{Push}; \mathcal{C}[\![M]\!]; \mathsf{Apply}$$

Code	Accu	Env	ArgSta	RetStack
Appterm; c_0	$a = (c_1, e_1)$	e_0	$v.s$	r
c_1	a	$v.e_1$	s	r
Apply; c_0	$a = (c_1, e_1)$	e_0	$v.s$	r
c_1	a	$v.e_1$	s	$(c_0, e_0).r$
Push; c_0	a	e	s	r
c_0	a	e	$a.s$	r
Pushmark; c_0	a	e	s	r
c_0	a	e	$\epsilon.s$	r

Abstractions

$$\mathcal{T}[\![\lambda E]\!] = \mathsf{Grab}; \mathcal{T}[\![E]\!]$$
$$\mathcal{C}[\![\lambda E]\!] = \mathsf{Cur}(\mathcal{T}[\![E]\!]; \mathsf{Return})$$

Code; c_0	Accu	Env	ArgSta	RetSta
Cur(c_1); c_0 c_0	a (c_1, e)	e e	s s	r r
Grab; c_0 c_1	a (c_0, e_0)	e_0 e_1	$\epsilon.s$ s	$(c_1, e_1).r$ r
Grab; c c	a a	e $v.e$	$v.s$ s	r r
Return; c_0 c_1	a a	e_0 e_1	$\epsilon.s$ s	$(c_1, e_1).r$ r
Return; c_0 c_1	$a = (c_1, e_1)$ a	e_0 $v.e_1$	$v.s$ s	r r

Conclusion

Is that all?

- graph reducer have been voluntarily ignored for lack of time
- there are many mathematical developments
- the Ocaml compiler is a more recent but derived implementation of the ZAM
- practical view of compilation
- practical performance issues as discussed for the JVM

References

[LER90] Xavier Leroy, *The ZINC Experiment: An Economical Implementation of the ML Language*, Rapport technique 117, INRIA, 1990.

[COU85] G. Cousineau, P.-L. Curien, M. Mauny, *The Categorical Abstract Machine*, In Functional Programming Languages and Computer Architecture, LNCS 201, 1985.

[LER98] Xavier Leroy, *An Introduction to Compiling Functional Languages*, tutorial given at the Workshop Types in Compilation, 1998.

[LAN64] P.J. Landin, *The Mechanical Evaluation of Expressions*, Computer Journal Vol. 6, 1964.

[HAR96] Thérèse Hardin, Luc Maranget, Bruno Pagano, *Functional Back-Ends within the Lambda-Sigma Calculus*, Rapport de recherche 3034, INRIA, 1996.

[ABA91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, *Explicit Substitutions*, Journal of Functional Programming, October 1991.

[TUR79] D.A. Turner, *A New Implementation Technique for Applicative Languages*, Software Practice and Experience Vol. 9, 1979.