From Krivine's machine
to the Caml implementations

Xavier Leroy

INRIA Rocquencourt

In this talk

An illustration of the strengths and limitations of abstract machines for the purpose of efficient execution of strict functional languages (Caml).

1. A retrospective on the ZAM, a call-by-value variant of Krivine's machine.

2. From abstract machines to efficient P.L. implementations: the example of Caml Light and Objective Caml.

3. Beyond abstract machines: push-enter models vs. eval-apply models.

Why are abstract machines interesting?

On the theory side (λ-calculus, semantics):

Abstract machines expose concepts lacking or implicit in the λ-calculus:

- Reduction strategy, evaluation order.
- Data representations, esp. closures and environments.

On the implementation side:

Compilation to abstract machine code offers much better performance than interpreters.

Abstract machine code as an intermediate language for machine-code generation.

Abstract machine code as an architecture-independent low-level format for distributing compiled programs (JVM, .Net).

There are better ways

In every area where abstract machines help, there are arguably better alternatives:

- Exposing reduction order: CPS transformation, structured operational semantics, reduction contexts.
- Closures and environments: explicit substitutions.
- Efficient execution: optimizing compilation to real machine code.
- Intermediate languages: RTL, SSA, CPS, A-normal forms, etc.
- Code distribution: ANDF (?).

Abstract machines do many things well but none very well.

This we will illustrate in the case of efficient execution of Caml.

A retrospective on the ZAM

The starting point

The Caml language and implementation circa 1989:

- Core ML (CBV λ-calculus) with primitives, data types and pattern-matching.
- Implemented using the CAM (Categorical Abstract Machine).
- On top of a Lisp run-time system (LeLisp).

The ZINC (ZINC Is Not Caml) experiment: exploration of alternative A.M., execution paths, and run-time systems.

Eventually grew into the Caml Light implementation.

Standard call-by-value abstract machines

The CAM (and SECD, and FAM, and . . .):

- Variables: look up value in environment (de Bruijn index).
- Abstractions: build a closure (code, current environment).
- Applications:
  evaluate function to closure (c, e);
  evaluate argument to value v;
  call code c with environment $v.e$.

Inessential variations in presentation, categorical folklore, etc.

More formally

$$\begin{aligned}
\mathcal{C}[\![n]\!] &= \texttt{ACCESS}(n) \\
\mathcal{C}[\![\lambda a]\!] &= \texttt{CLOSURE}(\mathcal{C}[\![a]\!];\texttt{RETURN}) \\
\mathcal{C}[\![a\ b]\!] &= \mathcal{C}[\![a]\!];\mathcal{C}[\![b]\!];\texttt{APPLY}
\end{aligned}$$

State before			State after		
Code	Env	Stack	Code	Env	Stack
$\texttt{ACCESS}(n); c$	e	s	c	e	$e(n).s$
$\texttt{CLOSURE}(c'); c$	e	s	c	e	$[c', e].s$
$\texttt{APPLY}; c$	e	$v.[c', e'].s$	c'	$v.e'$	$c.e.s$
$\texttt{RETURN}; c$	e	$v.c'.e'.s$	c'	e'	$v.s$

The problem with multiple-argument functions

λ-calculus has no built-in notion of function with several arguments. Encodes them in one of two ways:

- Tuples of arguments:
  $\lambda p.\ \texttt{match}\ p\ \texttt{with}\ (x_1, x_2, \ldots, x_n) \rightarrow a$
  $f\ (a_1, a_2, \ldots, a_n)$

- Currying:
  $\lambda x_1.\lambda x_2 \ldots \lambda x_n.\ a$
  $f\ a_1\ a_2\ \ldots\ a_n$

Currying enables partial applications and is the favored encoding in Caml and Haskell. (SML favours tuples.)

Currying in the CAM

Curried function application is costly in the CAM:

$$\begin{aligned}
\mathcal{C}[\![f\ a_1\ \ldots\ a_n]\!] &= \mathcal{C}[\![f]\!];\mathcal{C}[\![a_1]\!];\mathtt{APPLY};\ldots;\mathcal{C}[\![a_n]\!];\mathtt{APPLY} \\
\mathcal{C}[\![\lambda^n\ b]\!] &= \mathtt{CLOSURE}(\ldots(\mathtt{CLOSURE}(\mathcal{C}[\![b]\!];\mathtt{RETURN})\ldots);\mathtt{RETURN})
\end{aligned}$$

Before the body b of the function starts executing, the CAM

- constructs $n-1$ intermediate, short-lived closures;
- performs $n-1$ calls that return immediately.

Towards a better solution

Look at what other stack-based languages do:
(Forth, Postscript)

- Caller pushes the n arguments on the stack.
- Caller jumps to function code.
- Function accesses arguments from the stack when needed.
- Eventually, function pops arguments, pushes results and returns to caller.

No creation of $n-1$ intermediate closures, etc.

Why should it be any different for a functional language?

Partial and over applications

Problem: with functions as first-class value, the function being applied can receive less arguments than it expects

$$(\lambda x.\ \lambda y.\ b)\ a$$

or more arguments than it expects

$$(\lambda x.\ \lambda y.\ x)\ (\lambda z.z)\ 1\ 2$$

Towards a better solution

Revised approach:

- Caller pushes the n arguments on the stack and records that it provided n arguments.
- Caller sets environment from closure, jumps to function code.
- Function pops arguments off the stack and into its environment, checking the number of arguments provided.
- If not enough arguments: function returns closure corresponding to partial application.
- If too many arguments: function tail-calls its result value with the remaining arguments.

Still no creation of $n - 1$ intermediate closures, etc.

The ZAM (ZINC Abstract Machine)

$$\begin{aligned}
\mathcal{C}[\![n]\!] &= \texttt{ACCESS}(n) \\
\mathcal{C}[\![\lambda a]\!] &= \texttt{CLOSURE}(\mathcal{T}[\![\lambda a]\!]) \\
\mathcal{C}[\![b\ a_1\ \ldots\ a_n]\!] &= \texttt{PUSHMARK}; \mathcal{C}[\![a_n]\!]; \ldots; \mathcal{C}[\![a_1]\!]; \mathcal{C}[\![b]\!]; \texttt{APPLY} \\
\\
\mathcal{T}[\![n]\!] &= \texttt{ACCESS}(n); \texttt{RETURN} \\
\mathcal{T}[\![\lambda a]\!] &= \texttt{GRAB}; \mathcal{T}[\![a]\!] \\
\mathcal{T}[\![b\ a_1\ \ldots\ a_n]\!] &= \mathcal{C}[\![a_n]\!]; \ldots; \mathcal{C}[\![a_1]\!]; \mathcal{C}[\![b]\!]; \texttt{TAILAPPLY}
\end{aligned}$$

Notice the right-to-left evaluation order for function applications.

Transitions of the ZAM

State before				State after			
Code	Env	AStk	RStk	Code	Env	AStk	Rstk
$\mathtt{ACCESS}(n); c$	e	s	r	c	e	$e(n).s$	r
$\mathtt{CLOSURE}(c'); c$	e	s	r	c	e	$[c', e].s$	r
$\mathtt{TAILAPPLY}; c$	e	$[c', e'].s$	r	c'	e'	s	r
$\mathtt{APPLY}; c$	e	$[c', e'].s$	r	c'	e'	s	$c.e.r$
$\mathtt{PUSHMARK}; c$	e	s	r	c	e	$\varepsilon.s$	r
$\mathtt{GRAB}; c$	e	$\varepsilon.s$	$c'.e'.r$	c'	e'	$[(\mathtt{GRAB}; c), e].s$	r
$\mathtt{GRAB}; c$	e	$v.s$	r	c	$v.e$	s	r
$\mathtt{RETURN}; c$	e	$v.\varepsilon.s$	$c'.e'.r$	c'	e'	$v.s$	r
$\mathtt{RETURN}; c$	e	$[c', e'].s$	r	c'	e'	s	r

Nothing is new under the sun?

Pierre Crégut, the first person to whom I showed this design, immediately commented:

> "Oh, but this is just Krivine's machine with some more stuff for call-by-value."

It took me a long time to learn what Krivine's machine is, wrap my head around it, and fully understand Crégut's remark.

16 years later, I think I managed all three.

Krivine's machine hidden in the ZAM

Call-by-name evaluation in the ZAM can be achieved with the following compilation scheme, isomorphic to that of Krivine's machine:

$$\begin{array}{rcl}
\mathcal{N}[\![n]\!] & = & \mathtt{ACCESS}(n); \mathtt{TAILAPPLY} \\
\mathcal{N}[\![\lambda a]\!] & = & \mathtt{GRAB}; \mathcal{N}[\![a]\!] \\
\mathcal{N}[\![b\ a]\!] & = & \mathtt{CLOSURE}(\mathcal{N}[\![a]\!]); \mathcal{N}[\![b]\!]
\end{array}$$

The other ZAM instructions (and the mark ε, and the return stack) are just extra call-by-value baggage.

Representing the environment

Simple abstract machines such as the CAM or Krivine's represent the environment as a heap-allocated linked list.

Heap allocation enables saving the environment in a function closure at any time.
→ Inefficiency 1: many environments are never saved in a function closure.

The list structure is natural because environments are extended one value at a time ($e \rightarrow v.e$).
→ Inefficiency 2: access in linked lists is expensive.

A generic view of environments

An environment structure for an abstract machine is any data structure equipped with the following operations:

$\texttt{add}(v_1, \ldots, v_n, e)$	Set entries $0 \ldots n-1$ to $v_1 \ldots v_n$ and shift other entries of e by n
$\texttt{access}_n(e)$	Return the n-th entry of e
$\texttt{close}(c, e)$	Build a heap-allocated closure of code c with environment e. Return a pair (closure, environment equivalent to e).

Examples: linked lists; heap-allocated arrays (copy on `add`).

Mixed representation of environments

Idea: use a stack to record environment entries added since the last closure construction.

Environment =
volatile part S (in a stack) + persistent part E (heap-allocated).

Operations on this representation:

$$\begin{aligned}
\texttt{add}(v_1, \ldots, v_n, (S, E)) &= (v_1 \ldots v_n.S, E) \\
\texttt{access}_n(S, E) &= S[n] \quad \text{if } n < |S| \\
\texttt{access}_n(S, E) &= \texttt{access}_{n-|S|}(E) \quad \text{if } n \geq |S| \\
\texttt{close}(c, (v_1 \ldots v_n.\varepsilon, E)) &= (\texttt{close}(c, E'), (\varepsilon, E')) \\
& \text{where } E' = \texttt{add}(v_1, \ldots, v_n, E)
\end{aligned}$$

Application to the ZAM

In the ZAM: the volatile part is stored on the return stack. The persistent part is represented as a heap-allocated array, copied at `close` time.

In the common case of a n-argument function, fully applied and that does not build any closure:

- The volatile part contains the n arguments plus `let`-bound variables if any.
- Applications of the function proceed without heap-allocating any environments.

Validation in the Caml Light implementation

Caml Light = a bytecode interpreter (written in C) that interprets ZAM instructions.

Dramatic reduction in number of heap allocations and GC time (typically by a factor of 4), due both to the mixed environment representation and to better handling of multiple applications.

Some overhead on environment accesses (test $n < |S|$) and on partial applications (uncommon).

Overall performance better than the original Caml implementation, despite the latter performing JIT compilation of CAM code.

Second design iteration:
ZAM2 and Objective Caml

The ZAM2

An evolution of the ZAM, used by the bytecode interpreter of Objective Caml.

- Merges the return stack and the argument stack in one.

  (Facilitates exception handling, stack resizing, etc.)

- Compile-time determination of the limit between the volatile and persistent parts of the environment.

  (Avoids dynamic tests at access time.)

- Builds minimal closures containing only the variables actually free in the function body.

  (Slower to build, but important to avoid memory leaks.)

- Optimized interpretation technology (threaded code).

Merging the two stacks

Return addresses can be put on the argument stack provided they are pushed **before** the arguments, along with the separation marks.

$$
\begin{array}{rcl}
\mathcal{C}[\![n]\!]\ k & = & \mathtt{ACCESS}(n); k \\
\mathcal{C}[\![\lambda a]\!]\ k & = & \mathtt{CLOSURE}(\mathcal{T}[\![\lambda a]\!]); k \\
\mathcal{C}[\![b\ a_1\ \ \ldots\ \ a_n]\!]\ k & = & \mathtt{PUSHRETADDR}(k); \\
 & & \mathcal{C}[\![a_n]\!]\ (\ldots\mathcal{C}[\![a_1]\!]\ (\mathcal{C}[\![b]\!]\ \mathtt{TAILAPPLY})) \\
 & & \\
\mathcal{T}[\![n]\!] & = & \mathtt{ACCESS}(n); \mathtt{RETURN} \\
\mathcal{T}[\![\lambda a]\!] & = & \mathtt{GRAB}; \mathcal{T}[\![a]\!] \\
\mathcal{T}[\![b\ a_1\ \ \ldots\ \ a_n]\!] & = & \mathcal{C}[\![a_n]\!]\ (\ldots\mathcal{C}[\![a_1]\!]\ (\mathcal{C}[\![b]\!]\ \mathtt{TAILAPPLY}))
\end{array}
$$

Relevant transitions of the ZAM2

State before			State after		
Code	Env	Stack	Code	Env	Stack
$\mathtt{PUSHRETADDR}(c'); c$	e	s	c	e	$\varepsilon.c'.e.s$
$\mathtt{TAILAPPLY}; c$	e	$[c', e'].s$	c'	e'	s
$\mathtt{RETURN}; c$	e	$v.\varepsilon.c'.e'.s$	c'	e'	$v.s$
$\mathtt{RETURN}; c$	e	$[c', e'].s$	c'	e'	s

Statically-known environment shape

In the ZAM, the threshold between stack- and heap-allocated environments varies depending on whether the function was fully or partially applied:

```
let f = λx. λy. x + y in

f 1 2;

let g = f 1 in g 2
```

In the first call, `x` and `y` are both on stack. In the second, `x` is in the heap, `y` on stack.

Statically-known environment shape

This variability can be avoided by treating partial applications as on-the-fly η-expansions:

$$f\ v \longrightarrow \lambda y.f\ v\ y$$

That is: a partial application $f\ v$, when applied, pushes back v on the stack as the first parameter, then falls through f.

Relevant transitions of the ZAM2

State before			State after		
Code	Env	Stack	Code	Env	Stack
$\mathtt{STACKACCESS}(n); c$	e	$v_1 \ldots v_n.s$	c	e	$v_n.v_1 \ldots v_n.s$
$\mathtt{ENVACCESS}(n); c$	e	s	c	e	$e(n).s$
$\mathtt{GRAB}(n); c$	e	$v_1 \ldots v_k.\varepsilon.c'.e'.s$	c'	e'	$cls.s$
$\mathtt{GRAB}(n); c$	e	$v_1 \ldots v_n.s$	c	e	$v_1 \ldots v_n.s$
$\mathtt{RETURN}(n); c$	e	$v.v_1 \ldots v_n.\varepsilon.c'.e'.s$	c'	e'	$v.s$
$\mathtt{RETURN}(n); c$	e	$[c', e'].v_1 \ldots v_n.s$	c'	e'	s
$\mathtt{RESTART}; c$	$[v_1, \ldots, v_k, e]$	s	c	e	$v_1 \ldots v_k.s$

with $cls = [(\mathtt{RESTART}; \mathtt{GRAB}(n); c), [v_1, \ldots, v_k, e]]$.

Compilation of variable accesses

$$
\begin{array}{rcl}
\mathcal{C}[\![n]\!]\ k & = & \texttt{STACKACCESS}(n); k \quad \text{if } n < \mathit{threshold} \\
\mathcal{C}[\![n]\!]\ k & = & \texttt{ENVACCESS}(n - \mathit{threshold}); k \quad \text{if } n \geq \mathit{threshold}
\end{array}
$$

$\mathit{threshold}$ is the number of enclosing `let` bindings + the number of parameters of the enclosing function.

Eval-apply vs. push-enter

Two models for multiple-argument function applications

Simon Peyton Jones coined the terms **eval-apply** and **push-enter** to describe the two approaches to argument passing illustrated by the CAM/FAM/SECD and the KAM/ZAM.

This dichotomy is not specific to abstract machines and applies to other execution schemes for functional languages:

- Translation to native-code.
- Translation to an intermediate language featuring multiple-argument functions (C, C--, Scheme, etc).

The push-enter model

The caller stores the values of all arguments provided, typically on a stack, and records the number of arguments.

The callee tests the number of arguments provided and handles arity mismatches:

- Build and return closure if partial application.
- Tail-call result value if over-application.

Example: Krivine's machine, the ZAM.

Example: variable-arity functions in C.

The eval-apply model

The callee always receives exactly the number of arguments it expects.

Arity mismatches are handled by the caller, not the callee.

Example: CAM/FAM/SECD. Here, all functions have arity 1; handling arity mismatches is thus trivial:

$$f\ a_1\ \ldots\ a_n \rightarrow (\ldots(f(a_1))(a_2)\ldots)(a_n)$$

Example: Pascal, Java, etc. Static typing guarantees that the number of arguments provided is equal to the number of parameters expected. No partial applications, no need to handle arity mismatches.

Eval-apply, multiple arguments and currying

Eval-apply can also handle arity mismatches and multiple-argument functions, provided the arity of a function value is available at run-time (stored in the function closure):

$$f\ a_1\ \ldots\ a_n\ \rightarrow\ \mathtt{apply}_n(f, a_1, \ldots, a_n)$$

where the $\mathtt{apply}_n$ combinator is:

```
$\mathtt{apply}_n$ = $\lambda$ f $x_1$ ... $x_n$.
    match arity(f) with
    | 1   -> $\mathtt{apply}_{n-1}$(f($x_1$), $x_2$, ..., $x_n$)
    | ...
    | n-1 -> $\mathtt{apply}_1$(f($x_1$, ..., $x_{n-1}$), $x_n$)
    | n   -> f($x_1$, ..., $x_n$)
    | n+1 -> $\mathtt{papp}_{n+1,n}$(f, $x_1$, ..., $x_n$)
    | n+2 -> $\mathtt{papp}_{n+2,n}$(f, $x_1$, ..., $x_n$)
    | ...
```

Eval-apply, multiple arguments and currying

The partial application combinators $\texttt{papp}_{p,q}$, where p is the arity of the function and $q < p$ the number of arguments provided:

$$
\begin{array}{l}
\texttt{papp}_{p,q}\ \texttt{=} \\
\qquad \lambda\ \texttt{f x}_1\ \texttt{...}\ \texttt{x}_q\texttt{.}\ \texttt{(}\lambda\ \texttt{x}_{q+1}\ \texttt{...}\ \texttt{x}_p\texttt{.}\ \texttt{f(x}_1\texttt{, ..., x}_p\texttt{))}
\end{array}
$$

This is essentially what the GHC compiler (latest release) does.

A simplification

Since partial applications are uncommon, we can treat them as taking their arguments one at a time, CAM-style:

```
$\texttt{apply}_n$ = $\lambda$f $x_1$ ... $x_n$.
    match arity(f) with
    | 1   -> $\texttt{apply}_{n-1}$(f($x_1$), $x_2$, ..., $x_n$)
    | ...
    | n   -> f($x_1$, ..., $x_n$)
    | _   -> curry(f)($x_1$)($x_2$)...($x_n$)

curry = $\lambda$f.
    match arity(f) with
    | 1 -> f
    | 2 -> $\lambda$x. ($\lambda$y. f(x, y))
    | 3 -> $\lambda$x. ($\lambda$y. ($\lambda$z. f(x, y, z)))
    | ...
```

This is essentially what the OCaml native-code compiler does.

And the winner is. . . eval-apply!

Surprisingly, eval-apply turns out to be more efficient and easier to implement than push-enter in optimizing native-code compilers such as OCaml and GHC.

(Despite the overhead of the `apply`, `papp` and `curry` combinators.)

To understand why, we need to look at factors that are not apparent at the abstract machine level:

- Hardware architecture factors.
- Static analyses.

Hardware architecture factors

1. Direct procedure calls (to code addresses known statically) are 5–20 times faster than indirect procedure calls (to code addresses resulting from a run-time computation, e.g. loaded from a closure).

   (Comes from pipelining and instruction prefetching.)

2. Manipulating data that resides in processor registers is 5 times faster than data that resides on the stack.

   Manipulating data that resides on the stack is twice as fast as data that resides on the heap.

   (Comes from the cache hierarchy and the memory bottleneck.)

These are deep architectural limitations: even (non-existent) special-purpose processors would exhibit them.

Known function analysis

Optimizing functional compilers perform static analyses to determine which function is being called.

This is obvious in the case of `let`-bound functions:

```
let f = λx. a in
... f 42 ...
```

Here, the code for `f 42` can call directly the code for λ`x. a`, without fetching the code address from the closure of `f`.

More complex static analyses such as k-CFA can also determine this for functions passed as parameters.

Typically, 80% of function applications can be turned into direct calls.

Known functions and arity adaptation

When the called function is known, so is its arity. The code generated by the caller can be specialized:

$$\begin{aligned} f^2\ a &\rightarrow \texttt{let}\ x = a\ \texttt{in}\ \lambda y.\ f^2(x,y) \\ f^2\ a\ b &\rightarrow f^2(a,b) \\ f^2\ a\ b\ c &\rightarrow f^2(a,b)(c) \end{aligned}$$

(Assuming $f^2(a,b)$ is known to have arity 1.)

Making the common case efficient

Typical break-down for function applications:

Function statically known	80%
Unknown function, arity matches	10%
Unknown function, partial application	5%
Unknown function, over-application	5%

This explain why the overhead of `apply` and `curry` combinators does not kill eval-apply.

It still does not explain why eval-apply is more efficient than push-enter.

Register-based calling conventions

Push-enter requires that function arguments are passed in stack locations, and manipulated in accordance with the abstract machine scheme.

Eval-apply is compatible with any calling convention, in particular modern conventions where the first N parameters are passed in hardware registers.

- Passing arguments in hardware registers is more efficient.

  (This benefits all function applications, esp. the 80% of known calls.)

- The compiler can use any existing back-end (e.g. a C compiler or a C-- code generator) that imposes its own calling conventions.

  (Separation of concerns.)

Conclusions

A journey through implementation techniques for F.P.L

Using the Caml compilers as concrete examples. we have encountered the following implementation techniques:

- Theoretician's abstract machines, e.g. the CAM, with bytecode interpretation.
- Practitioner's abstract machines, e.g. the ZAM2, with bytecode interpretation.
- Practitioner's abstract machines, e.g. the ZAM2, with JIT native-code generation.
- Optimizing native-code compiler.

Each step represents a performance gain by a factor of 2 to 4.

On the usefulness of abstract machines

As a discovery tool: many of the key issues (e.g. eval-apply vs. push-enter) were discovered by thinking in terms of abstract machines, even though they are also apparent in terms of translations to lower-level functional languages.

As a tool to prove compiler correctness: both theoretician's abstract machines and practitioner's abstract machines can be proved correct w.r.t. the λ-calculus (using e.g. explicit substitutions), or derived in a systematic, semantics-preserving way.

As a cheap implementation device: bytecode interpreters offer 1/4 of the performance of optimizing native-code compilers, at 1/20 of the implementation cost.

Last words

In retrospect, the journey could have been shorter by not going through abstract machines at all.

But maybe it would never have taken place without the “detour” through abstract machines.