Abstract machines

Outline

Execution models for a language

1. **Interpretation:** control (sequencing of computations) is expressed by a term of the source language, represented by a tree-shaped data structure. The interpreter traverses this tree during execution.
2. **Compilation to native code:** control is compiled to a sequence of machine instructions, before execution. These instructions are those of a real microprocessor and are executed in hardware.
3. **Compilation to abstract machine code:** control is compiled to a sequence of instructions. These instructions are those of an abstract machine. They do not correspond to that of an existing hardware processor, but are chosen close to the basic operations of the source language.

Execution models for a language

1. **Interpretation:** control (sequencing of computations) is expressed by a term of the source language, represented by a tree-shaped data structure. The interpreter traverses this tree during execution.
2. **Compilation to native code:** control is compiled to a sequence of machine instructions, before execution. These instructions are those of a real microprocessor and are executed in hardware.
3. **Compilation to abstract machine code:** control is compiled to a sequence of instructions. These instructions are those of an abstract machine. They do not correspond to that of an existing hardware processor, but are chosen close to the basic operations of the source language. Yet another example of program transformations between different languages

Execution models for a language

1. **Interpretation:** control (sequencing of computations) is expressed by a term of the source language, represented by a tree-shaped data structure. The interpreter traverses this tree during execution.
2. **Compilation to native code:** control is compiled to a sequence of machine instructions, before execution. These instructions are those of a real microprocessor and are executed in hardware.
3. **Compilation to abstract machine code:** control is compiled to a sequence of instructions. These instructions are those of an abstract machine. They do not correspond to that of an existing hardware processor, but are chosen close to the basic operations of the source language.Yet another example of program transformations between different languages

Next: short overview of abstract machines for functional languages

Outline

Abstract machine for arithmetic expressions

Arithmetic expressions:

$$a ::= N \mid a + a \mid a - a$$

Machine Components

1. A code pointer
2. A stack

Instruction set:

CONST(N)	push integer N on stack
ADD	pop two integers, push their sum
SUB	pop two integers, push their difference

Compilation scheme

Compilation (translation of expressions to sequences of instructions) is just translation to "reverse Polish notation":

$$\begin{aligned} [\![N]\!] &= \texttt{CONST}(N) \\ [\![a_1 + a_2]\!] &= [\![\mathtt{a}_1]\!];[\![\mathtt{a}_2]\!];\texttt{ADD} \\ [\![a_1 - a_2]\!] &= [\![\mathtt{a}_1]\!];[\![\mathtt{a}_2]\!];\texttt{SUB} \end{aligned}$$

Example

$[\![\ 5 - (1 + 2)]\!] = \texttt{CONST(5)};\ \texttt{CONST(1)};\ \texttt{CONST(2)};\ \texttt{ADD};\ \texttt{SUB}$

Transitions

BEFORE		AFTER	
Code	Stack	Code	Stack
CONST(N) ; c	s	c	$N.s$
ADD ; c	$n_2.n_1.s$	c	$(n_1 + n_2).s$
SUB ; c	$n_2.n_1.s$	c	$(n_1 - n_2).s$

Transitions

BEFORE		AFTER	
Code	Stack	Code	Stack
`CONST(`N`) ;` c	s	c	$N.s$
`ADD ;` c	$n_2.n_1.s$	c	$(n_1+n_2).s$
`SUB ;` c	$n_2.n_1.s$	c	$(n_1-n_2).s$

Let us try to execute the compilation of $5-(1+2)$ with an empty stack

Transitions

BEFORE		AFTER	
Code	Stack	Code	Stack
CONST(N) ; c	s	c	$N.s$
ADD ; c	$n_2.n_1.s$	c	$(n_1 + n_2).s$
SUB ; c	$n_2.n_1.s$	c	$(n_1 - n_2).s$

Let us try to execute the compilation of $5 - (1 + 2)$ with an empty stack

Code	Stack
CONST(5); CONST(1); CONST(2); ADD; SUB	ε
CONST(1); CONST(2); ADD; SUB	5
CONST(2); ADD; SUB	1.5
ADD; SUB	2.1.5
SUB	3.5
ε	2

Transitions

BEFORE		AFTER	
Code	Stack	Code	Stack
CONST(N) ; c	s	c	$N.s$
ADD ; c	$n_2.n_1.s$	c	$(n_1 + n_2).s$
SUB ; c	$n_2.n_1.s$	c	$(n_1 - n_2).s$

Let us try to execute the compilation of $5 - (1 + 2)$ with an empty stack

Code	Stack
CONST(5); CONST(1); CONST(2); ADD; SUB	ε
CONST(1); CONST(2); ADD; SUB	5
CONST(2); ADD; SUB	1.5
ADD; SUB	2.1.5
SUB	3.5
ε	2

Notice the right-to-left execution order

Outline

SECD: abstract-machine for call by value

Machine Components

1. A code pointer
2. An environment
3. A stack

Instruction set: (+ previous arithmetic operations)

`ACCESS(`n`)`	push n-th field of the environment
`CLOSURE(`c`)`	push closure of code c with current environment
`LET`	pop value and add it to environment
`ENDLET`	discard first entry of environment
`APPLY`	pop function closure and argument, perform application
`RETURN`	terminate current function, jump back to caller

Historical note: (S)tack, (E)nvironment, (C)ontrol, (D)ump. (SCD) are implemented by stacks, (E) is an array. (C) is our code pointer, (D) is the return stack as in the first version of the ZAM later on.

Compilation scheme

$$
\begin{aligned}
[\![\underline{n}]\!] &= \mathtt{ACCESS}(n) \\
[\![\lambda a]\!] &= \mathtt{CLOSURE}([\![a]\!]\ ;\ \mathtt{RETURN}) \\
[\![\mathtt{let}\ a\ \mathtt{in}\ b]\!] &= [\![a]\!]\ ;\ \mathtt{LET}\ ;\ [\![b]\!]\ ;\ \mathtt{ENDLET} \\
[\![ab]\!] &= [\![a]\!]\ ;\ [\![b]\!]\ ;\ \mathtt{APPLY}
\end{aligned}
$$

(constants and arithmetic as before)

Compilation scheme

$$
\begin{aligned}
[\![\underline{n}]\!] &= \texttt{ACCESS}(n) \\
[\![\lambda a]\!] &= \texttt{CLOSURE}([\![a]\!] \;;\; \texttt{RETURN}) \\
[\![\texttt{let}\ a\ \texttt{in}\ b]\!] &= [\![a]\!] \;;\; \texttt{LET} \;;\; [\![b]\!] \;;\; \texttt{ENDLET} \\
[\![ab]\!] &= [\![a]\!] \;;\; [\![b]\!] \;;\; \texttt{APPLY}
\end{aligned}
$$

(constants and arithmetic as before)

Example

Term:

$(\lambda(\underline{0}+1))2$ (i.e., $(\lambda x.x+1)2$)

Code:

`CLOSURE(ACCESS(0);CONST(1);ADD;RETURN) ; CONST(2) ; APPLY`

Transitions

BEFORE			AFTER		
Code	Env	Stack	Code	Env	Stack
ACCESS(n); c	e	s	c	e	$e(n).s$
LET; c	e	$v.s$	c	$v.e$	s
ENDLET; c	$v.e$	s	c	e	s
CLOSURE(c'); c	e	s	c	e	$c'[e].s$
APPLY; c	e	$v.c'[e'].s$	c'	$v.e'$	$c.e.s$
RETURN; c	e	$v.c'.e'.s$	c'	e'	$v.s$

where $c[e]$ denotes the closure of code c with environment e.

Example

Code: CLOSURE(c); CONST(2); APPLY
where: c = ACCESS(0);CONST(1);ADD;RETURN

Code	Env	Stack
CLOSURE(c); CONST(2); APPLY	e	s
CONST(2); APPLY	e	$c[e].s$
APPLY	e	$2.c[e].s$
c	$2.e$	$\varepsilon.e.s$
CONST(1);ADD;RETURN	$2.e$	$2.\varepsilon.e.s$
ADD;RETURN	$2.e$	$1.2.\varepsilon.e.s$
RETURN	$2.e$	$3.\varepsilon.e.s$
ε	e	$3.s$

Soundness

Of course we always have to show that the compilation is correct, in the sense that it preserves the semantics of the reduction. This is stated as follows

Theorem (soundness of SECD)

If $e \Rightarrow v$ then the SECD machine in the state $([\![e]\!], \varepsilon, \varepsilon)$ reduces to the state $(\varepsilon, \varepsilon, \bar{v})$, where $\bar{v}$ is the machine value for v (the same integer for an integer, and the corresponding closure for a λ-abstraction.)

(where $\Rightarrow$ is the call-by-value, weak-reduction, big-step semantics defined in the “Refresher course on operational semantics”)

Outline

An optimization: tail call elimination

Consider:

```
f = λ. ... g 1 ...
g = λ. h(...)
h = λ. ...
```

The call from `g` to `h` is a tail call: when `h` returns, `g` has nothing more to compute, it just returns immediately to `f`.

At the machine level, the code of `g` is of the form `...;APPLY;RETURN`
When `g` calls `h`, it pushes a return frame on the stack containing the code `RETURN`. When `h` returns (e.g. a value v_h), it jumps to this `RETURN` in `g`, which jumps to the continuation in `f`.

Tail-call elimination consists in avoiding this extra return frame and this extra `RETURN` instruction, enabling `h` to return directly to `f`, and saving stack space.

An optimization: tail call elimination

```
f = λ. ... g 1 ...
g = λ. h(...)
h = λ. ...
```

Code	Env	Stack
APPLY;RETURN_g	e	$v.c_h[e_h].c_f.e_f.s$
c_h	$v.e_h$	$(\mathrm{RETURN}_g).e.c_f.e_f.s$
$\vdots$	$\vdots$	$\vdots$
RETURN_h	e''	$v_h.(\mathrm{RETURN}_g).e.c_f.e_f.s$
RETURN_g	e	$v_h.c_f.e_f.s$
c_f	e_f	$v_h.s$

An optimization: tail call elimination

```
f = λ. ... g 1 ...
g = λ. h(...)
h = λ. ...
```

Code	Env	Stack
APPLY;$\mathtt{RETURN}_g$	e	$v.c_h[e_h].c_f.e_f.s$
c_h	$v.e_h$	$(\mathtt{RETURN}_g).e.c_f.e_f.s$
⋮	⋮	⋮
$\mathtt{RETURN}_h$	e''	$v_h.(\mathtt{RETURN}_g).e.c_f.e_f.s$
$\mathtt{RETURN}_g$	e	$v_h.c_f.e_f.s$
c_f	e_f	$v_h.s$

Tail-call elimination consists in avoiding this extra return frame and this extra RETURN instruction, enabling h to return directly to f, and saving stack space.

The importance of tail call elimination

Tail call elimination is important for recursive functions whose recursive calls are in tail position — the functional equivalent to loops in imperative languages:

```
let rec fact n accu =
     if n = 0 then accu else fact (n-1) (accu*n)
 in fact 42 1
```

With tail call elimination, this code runs in constant stack space.

Without tail call elimination, it consumes $O(n)$ stack space exactly as

```
let rec fact n = if n = 0 then 1 else n * fact (n-1)
 in fact 42
```

Hello stack overflows!

SECD with tail-call elimination

Machine Components: as before

Instruction set: as before plus

`TAILAPPLY` perform application without pushing the return frame

Compilation scheme:

Split the compilation scheme in two functions: $\mathcal{T}$ for expressions in tail call position, $\mathcal{C}$ for other expressions.

$$
\begin{aligned}
\mathcal{T}[\![\texttt{let } a \texttt{ in } b]\!] &= \mathcal{C}[\![a]\!];\texttt{LET};\mathcal{T}[\![b]\!] \\
\mathcal{T}[\![ab]\!] &= \mathcal{C}[\![a]\!];\mathcal{C}[\![b]\!];\texttt{TAILAPPLY} \\
\mathcal{T}[\![a]\!] &= \mathcal{C}[\![a]\!];\texttt{RETURN} \qquad (\textit{otherwise}) \\
\\
\mathcal{C}[\![n]\!] &= \texttt{ACCESS}(n) \\
\mathcal{C}[\![\lambda a]\!] &= \texttt{CLOSURE}(\mathcal{T}[\![a]\!]) \\
\mathcal{C}[\![\texttt{let } a \texttt{ in } b]\!] &= \mathcal{C}[\![a]\!];\texttt{LET};\mathcal{C}[\![b]\!];\texttt{ENDLET} \\
\mathcal{C}[\![ab]\!] &= \mathcal{C}[\![a]\!];\mathcal{C}[\![b]\!];\texttt{APPLY}
\end{aligned}
$$

Transitions

The TAILAPPLY instruction behaves like APPLY, but does not bother pushing a return frame to the current function

BEFORE			AFTER		
Code	Env	Stack	Code	Env	Stack
TAILAPPLY; c	e	$v.c'[e'].s$	c'	$v.e'$	s
APPLY; c	e	$v.c'[e'].s$	c'	$v.e'$	$c.e.s$

Transitions

The `TAILAPPLY` instruction behaves like `APPLY`, but does not bother pushing a return frame to the current function

BEFORE			AFTER		
Code	Env	Stack	Code	Env	Stack
TAILAPPLY; c	e	$v.c'[e'].s$	c'	$v.e'$	s
APPLY; c	e	$v.c'[e'].s$	c'	$v.e'$	$c.e.s$

Note also that $\mathcal{T}[\![\texttt{let } a \texttt{ in } b]\!]$ does not end by `ENDLET`, since every code produced by $\mathcal{T}[\![]\!]$ ends either by `TAILAPPLY` and `RETURN`, and both `TAILAPPLY` and `RETURN` throw the current environment away.

Back to the example

Code	Env	Stack
APPLY;$\mathtt{RETURN}_g$	e	$v.c_h[e_h].c_f.e_f.s$
c_h	$v.e_h$	$(\mathtt{RETURN}_g).e.c_f.e_f.s$
$\vdots$	$\vdots$	$\vdots$
$\mathtt{RETURN}_h$	e''	$v_h.(\mathtt{RETURN}_g).e.c_f.e_f.s$
$\mathtt{RETURN}_g$	e	$v_h.c_f.e_f.s$
c_f	e_f	$v_h.s$

Code	Env	Stack
TAILAPPLY	e	$v.c_h[e_h].c_f.e_f.s$
c_h	$v.e_h$	$c_f.e_f.s$
$\vdots$	$\vdots$	$\vdots$
$\mathtt{RETURN}_h$	e''	$v_h.c_f.e_f.s$
c_f	e_f	$v_h.s$

Outline

The Krivine Machine $\mathcal{K}$

Machine Components

1. A code pointer c
2. An environment e
3. A stack s

Difference: stacks and environments no longer contain values but "thunks". These are closures $c[e]$ for generic expressions (not just λ's) and represent "frozen" expressions that are to be evaluated.

Instruction set:

ACCESS(n)	start evaluating the thunk at the n-th position of the environment
PUSH(c)	push a thunk for code c
GRAB	pop one argument and cons it to the environment

Compilation scheme

- Application pushes the argument as a thunk (i.e., current expression + its environment) on the stack and evaluates the function.
- λ-abstraction grabs its argument(s) from the stack and evaluates its body

$$
\begin{aligned}
[\![\underline{n}]\!] &= \mathtt{ACCESS(}n\mathtt{)} \\
[\![\lambda a]\!] &= \mathtt{GRAB}\ \mathtt{;}\ [\![a]\!] \\
[\![ab]\!] &= \mathtt{PUSH(}[\![b]\!]\mathtt{)}\ \mathtt{;}\ [\![a]\!]
\end{aligned}
$$

Nota bene

- Close to lambda calculus: three instructions for three terms
- Implements call-by-name

$$((\lambda.a)[e])(b[e']) \rightarrow a[b[e'].e]$$

(λ-calculus with explicit substitutions)

Transitions

BEFORE			AFTER		
Code	Env	Stack	Code	Env	Stack
$\texttt{ACCESS}(n); c$	e	s	$c'; c$	e'	s
$\texttt{GRAB}; c$	e	$c'[e'].s$	c	$c'[e'].e$	s
$\texttt{PUSH}(c'); c$	e	s	c	e	$c'[e].s$

if $e(n) = c'[e']$

Transitions

BEFORE			AFTER			
Code	Env	Stack	Code	Env	Stack	
$\texttt{ACCESS}(n); c$	e	s	$c'; c$	e'	s	if $e(n) = c'[e']$
$\texttt{GRAB}; c$	e	$c'[e'].s$	c	$c'[e'].e$	s	
$\texttt{PUSH}(c'); c$	e	s	c	e	$c'[e].s$	

In pure λ-calculus `ACCESS()` has no continuation c, so it is rather

BEFORE			AFTER			
Code	Env	Stack	Code	Env	Stack	
$\texttt{ACCESS}(n)$	e	s	c'	e'	s	if $e(n) = c'[e']$
⋮	⋮	⋮	⋮	⋮	⋮	

Soundness and efficiency

Soundness:

Krivine's machine is much closer to λ-calculus, so it has a stronger soundness result in the sense that every reduction step of the Krivine machine corresponds to a reduction step in the CBN λ-calculus (technically, to the CBN λ-calculus with explicit substitutions).

The soundness of SECD is stated just for the big-step semantics.

Soundness and efficiency

Soundness:

Krivine's machine is much closer to λ-calculus, so it has a stronger soundness result in the sense that every reduction step of the Krivine machine corresponds to a reduction step in the CBN λ-calculus (technically, to the CBN λ-calculus with explicit substitutions).
The soundness of SECD is stated just for the big-step semantics.

Efficiency:

Krivine's machine is highly inefficient

- Duplicated execution of the same expressions (call-by-value instead of call-by-need)
- Duplicated values stored on the heap (no mark compression)
- Redundant information for variables (it dumbly stores a variable with its closure, instead of storing directly the value the varible is bound to)
- Much more (see research papers).

Outline

Adding call-by-need

We add an indirection to a HEAP, which maps locations to closures.

Evironments map variables (De Brujin indexes) to locations of the heap.

A **value** ($\in \mathcal{V}al$) is a closure of the form $(\texttt{GRAB};c)[e]$ (compiles as before)

We split the rules for variables (ACCESS()) and lambda's (GRAB;c) in two:

BEFORE				AFTER				
Code	Env	Stack	Heap	Code	Env	Stack	Heap	
ACCESS(n)	e	s	h	c'	e'	s	h	(1)
ACCESS(n)	e	s	h	c'	e'	$\text{mrk}(\ell).s$	h	(2)
GRAB; c	e	$c'[e'].s$	h	c	$\ell.e$	s	$h\{\ell \mapsto c'[e']\}$	(3)
GRAB; c	e	$\text{mrk}(\ell).s$	h	GRAB;c	e	s	$h\{\ell \mapsto (\texttt{GRAB};c)[e]\}$	(4)
PUSH(c'); c	e	s	h	c	e	$c'[e].s$	h	

(1) if $e(n) = \ell$ and $h(\ell) = c'[e'] \in \mathcal{V}al$ — activate the value stored for n

(2) if $e(n) = \ell$ and $h(\ell) = c'[e'] \notin \mathcal{V}al$ — activate expr and mark the stack

(3) ℓ is fresh — grab the argument on the top of the stack and allocate on heap

(4) — store in the heap the value computed for the location ℓ

Mark compression

In some situations in the stack may contain sequences of markers.

For example for $(\lambda z.(\lambda y.z(yz))z)(\lambda x.x)$ the machine reduces at some point to a stack of the form mrk(ℓ_2).mrk(ℓ_1) and both locations contain the closure $(\lambda x.x)[\,]$ (try it)

When a sequence of markers is popped from the stack, the same value is assigned to each heap location pointed to by the markers

Optimization: avoid creating sequences of markers by sharing the first marker and result location among closures that receive the same value.

Mark compression

Solution: Add one level of indirection

Before: Environments map variables to pointers to closures

Now: Environments map variables to pointers to pointers to closures

BEFORE				AFTER				
Code	Env	Stack	Heap	Code	Env	Stack	Heap	
ACCESS(n)	e	s	h	c'	e'	s	h	(1)
ACCESS(n)	e	$\mathrm{mrk}(\ell').s$	h	c'	e'	$\mathrm{mrk}(\ell').s$	$h\{e(n) \mapsto \ell'\}$	(2a)
ACCESS(n)	e	s	h	c'	e'	$\mathrm{mrk}(\ell).s$	h	(2b)
GRAB; c	e	$c'[e'].s$	h	c	$\ell.e$	s	$h\{\ell \mapsto \ell'; \ell' \mapsto c'[e']\}$	(3)
GRAB; c	e	$\mathrm{mrk}(\ell).s$	h	GRAB;c	e	s	$h\{\ell \mapsto (\mathtt{GRAB}; c)[e]\}$	
PUSH(c'); c	e	s	h	c	e	$c'[e].s$	h	

(1) if $h(e(n))=\ell$ and $h(\ell)=c'[e'] \in \mathcal{V}al$

(2a) if $h(e(n))=\ell$ and $h(\ell)=c'[e'] \notin \mathcal{V}al$ — map $e(n)$ to ℓ' and dealloc ℓ

(2b) if $h(e(n))=\ell$ and $h(\ell)=c'[e'] \notin \mathcal{V}al$ and $s \neq \mathrm{mrk}(\ell').s'$ — proceed as before

(3) ℓ and ℓ' are fresh

Short circuiting for dereferencing

When the argument of a function is a variable deferencing is not efficient.

Consider $(\lambda x.Mx)N$.

1. We evaluate Mx in the environment $\{x \mapsto N[\,]\}$.
2. This pushes on the stack the closure $x[\{x \mapsto N[\,]\}]$: silly!
3. Much more efficient and clever to push directly on the stack the closure $N[\,]$ (i.e., the result of evaluating x in the environment $\{x \mapsto N[\,]\}$)

We short-circuit the deferencing of a variable in argument position.

An optimization already present in early implementations of Algol 60.

Rationale: now expressions in closures are never variables. They are
- either lambdas (the closure is a value)
- or applications (the closure is a "thunk", a frozen expression).

Short circuiting for dereferencing

1. We split the rule for application (`PUSH()`) in two cases: when the argument is a variable and when it is not
2. We modify the rule for lambdas, since heap allocation is now performed at the application (instead of `GRAB`)

BEFORE				AFTER				
Code	Env	Stack	Heap	Code	Env	Stack	Heap	
$\texttt{ACC}(n)$	e	s	h	c'	e'	s	h	(1)
$\texttt{ACC}(n)$	e	$\mathsf{mrk}(\ell').s$	h	c'	e'	$\mathsf{mrk}(\ell').s$	$h\{e(n)\mapsto\ell'\}$	(2a)
$\texttt{ACC}(n)$	e	s	h	c'	e'	$\ell.s$	h	(2b)
$\texttt{GRAB};c$	e	$\mathsf{arg}(\ell).s$	h	c	$\ell.e$	s	h	
$\texttt{GRAB};c$	e	$\mathsf{mrk}(\ell).s$	h	$\texttt{GRAB};c$	e	s	$h\{\ell\mapsto(\texttt{GRAB};c)[e]\}$	
$\texttt{PUSH}(\texttt{ACC}(n));c$	e	s	h	c	e	$\mathsf{arg}(e(n)).s$	h	
$\texttt{PUSH}(c');c$	e	s	h	c	e	$\mathsf{arg}(\ell').s$	$h\{\ell\mapsto\ell';\ell'\mapsto c'[e]\}$	(3)

wrote `ACC()` instead of `ACCESS()` for space reasons

(1) if $h(e(n))=\ell$ and $h(\ell)=c'[e']\in\mathcal{V}al$

(2a) if $h(e(n))=\ell$ and $h(\ell)=c'[e']\notin\mathcal{V}al$

(2b) if $h(e(n))=\ell$ and $h(\ell)=c'[e']\notin\mathcal{V}al$ and $s\not\equiv\mathsf{mrk}(\ell').s'$

(3) ℓ and ℓ' are fresh and $c'\not\equiv\texttt{ACC}(n)$

Outline

Eval-apply vs. Push-enter

Real machines are more sophisticated (e.g., register allocation, garbage collection, ...) and there exist many more variants than the ones presented. See Marlow and Peyton Jones's JFP'06 paper for better approximation.

Peyton Jones classifies AM for functional languages based on two subtly different ways to evaluate a function application $f\ a\ b$:

- Push-enter: (e.g., Krivine)
  Push on stack the arguments a and b and *enter* the code of for f (that at some point will try to grab its arguments from the stack)
- Eval-apply: (e.g., SECD)
  Evaluate f (to a closure $c[e]$) and *apply* it to the right number of arguments (i.e., evaluate a and extend environment e with its result and, if f is binary, do the same with b)

Eval-apply vs. Push-enter

Real machines are more sophisticated (e.g., register allocation, garbage collection, ...) and there exist many more variants than the ones presented. See Marlow and Peyton Jones's JFP'06 paper for better approximation.

Peyton Jones classifies AM for functional languages based on two subtly different ways to evaluate a function application $f\ a\ b$:

- Push-enter: (e.g., Krivine)
  Push on stack the arguments a and b and *enter* the code of for f (that at some point will try to grab its arguments from the stack)
- Eval-apply: (e.g., SECD)
  Evaluate f (to a closure $c[e]$) and *apply* it to the right number of arguments (i.e., evaluate a and extend environment e with its result and, if f is binary, do the same with b)

The difference becomes significant for curried applications of functions whose arity is *not* statically known:

The problem with arity

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith k [] [] = []
zipWith k (x:xs) (y:ys) = k x y : zipWith k xs ys
```

The problem with arity

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith k [] [] = []
zipWith k (x:xs) (y:ys) = k x y : zipWith k xs ys
```

Here `k` can end up to be unary, binary, ternary ... or more:

1. $(\lambda x.x)(\lambda x.x)y$ (apply first to second)
2. $(\lambda x.\lambda y.x+y)xy$ (sum first and second)
3. $(\lambda x.\lambda y.\lambda z.z)xy$ (return a list of identities)

The arity of the function `k` is known only when it is bound to a closure.

The problem with arity

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith k [] [] = []
zipWith k (x:xs) (y:ys) = k x y : zipWith k xs ys
```

Here `k` can end up to be unary, binary, ternary ... or more:

1. $(\lambda x.x)(\lambda x.x)y$ (apply first to second)
2. $(\lambda x.\lambda y.x+y)xy$ (sum first and second)
3. $(\lambda x.\lambda y.\lambda z.z)xy$ (return a list of identities)

The arity of the function `k` is known only when it is bound to a closure.

Arity matching: match the function arity with the # of arguments available:

- Push-enter: the function, which statically knows its own arity, examines the stack to figure out how many arguments it has been passed, and where they are. the *callee* is responsible for arity matching
- Eval-apply: the caller, which statically knows what the arguments are, examines the function closure, extracts its arity, and makes an exact call to the function. the *caller* is responsible for arity matching

The problem with arity

Consider again `k x y`

- Push-enter:
  - if there are too few arguments, the function must construct a partial application and return.
  - if there are too many arguments, then only the required arguments are consumed, the rest of the arguments are left on the stack to be consumed later
- Eval-apply:
  - If k takes two arguments, call it straightforwardly.
  - If k takes one, call it passing x, and call the resulting function passing y;
  - if k takes more than two, build and return a closure for partial application k x y

The problem with arity

Consider again `k x y`

- Push-enter:
  - if there are too few arguments, the function must construct a partial application and return.
  - if there are too many arguments, then only the required arguments are consumed, the rest of the arguments are left on the stack to be consumed later
- Eval-apply:
  - If k takes two arguments, call it straightforwardly.
  - If k takes one, call it passing x, and call the resulting function passing y;
  - if k takes more than two, build and return a closure for partial application k x y

Nota bene:

This holds only for calls of *unknown* functions. For known functions such as:

```
let g x y = x*y
 in g 3 4
```

any decent compiler must load the arguments 3 and 4 into registers, or on the stack, and call the code for g directly (no closures created) both in push/enter and eval/apply

Outline

Curried functions

In eval-apply the application of curried functions is costly

$$
\begin{aligned}
[\![f\, a_1 ... a_n]\!] &= [\![f]\!]\ ;\ [\![a_1]\!]\ ;\ \texttt{APPLY}\ ;\ ...\ ;\ [\![a_n]\!]\ ;\ \texttt{APPLY} \\
[\![\lambda^n.b]\!] &= \texttt{CLOSURE(...(CLOSURE(}[\![b]\!]\texttt{;RETURN)...);RETURN)}
\end{aligned}
$$

Before the body b of the function starts executing, the SECD:

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

Curried functions

In eval-apply the application of curried functions is costly

$$
\begin{array}{rcl}
[\![f\, a_1 \ldots a_n]\!] & = & [\![f]\!]\ ;\ [\![a_1]\!]\ ;\ \texttt{APPLY}\ ;\ \ldots\ ;\ [\![a_n]\!]\ ;\ \texttt{APPLY} \\
[\![\lambda^n.b]\!] & = & \texttt{CLOSURE}(\ldots(\texttt{CLOSURE}([\![b]\!];\texttt{RETURN})\ldots);\texttt{RETURN})
\end{array}
$$

Before the body b of the function starts executing, the SECD:

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

In push-enter it is more efficient:

$$
\begin{array}{rcl}
[\![f\, a_1 \ldots a_n]\!] & = & \texttt{PUSH}([\![a_n]\!]);\ldots;\texttt{PUSH}([\![a_1]\!]);[\![\texttt{f}]\!] \\
[\![\lambda^n.b]\!] & = & \underbrace{\texttt{GRAB};\ldots;\texttt{GRAB}}_{n\ times};[\![\texttt{b}]\!]
\end{array}
$$

Push all the arguments, enter the function that grabs the needed arguments and executes the body.

Curried functions

In eval-apply the application of curried functions is costly

$$[\![f\,a_1 \ldots a_n]\!] = [\![f]\!]\ ;\ [\![a_1]\!]\ ;\ \mathtt{APPLY}\ ;\ \ldots\ ;\ [\![a_n]\!]\ ;\ \mathtt{APPLY}$$
$$[\![\lambda^n.b]\!] = \mathtt{CLOSURE(...(CLOSURE(}[\![b]\!]\mathtt{;RETURN)...);RETURN)}$$

Before the body b of the function starts executing, the SECD:

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

In push-enter it is more efficient:

$$[\![f\,a_1 \ldots a_n]\!] = \mathtt{PUSH(}[\![a_n]\!]\mathtt{)};\ldots;\mathtt{PUSH(}[\![a_1]\!]\mathtt{)};[\![\mathtt{f}]\!]$$
$$[\![\lambda^n.b]\!] = \underbrace{\mathtt{GRAB};\ldots;\mathtt{GRAB}}_{n\ times};[\![\mathtt{b}]\!]$$

Push all the arguments, enter the function that grabs the needed arguments and executes the body.

Let us try each technique on the application $(\lambda.\lambda.\lambda.\underline{0})2\,1\,0$

Curried function application in eval-apply

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!]$ =

```
CLOSURE(CLOSURE(CLOSURE(ACCESS(0);RETURN);RETURN);RETURN);
CONST(2) ; APPLY ; CONST(1) ; APPLY ; CONST(0) ; APPLY
```

Curried function application in eval-apply

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] =$

`CLOSURE(CLOSURE(CLOSURE(ACCESS(0);RETURN);RETURN);RETURN);`
`CONST(2) ; APPLY ; CONST(1) ; APPLY ; CONST(0) ; APPLY`

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] = \texttt{CLOSURE}(c_2);a_2$

where for $i = 1,2$

$$
\begin{array}{rcl}
c_0 & = & \texttt{ACCESS(0) ; RETURN} \\
c_i & = & \texttt{CLOSURE}(c_{i-1})\texttt{ ; RETURN} \\
a_0 & = & \texttt{CONST(0) ; APPLY} \\
a_i & = & \texttt{CONST(i) ; APPLY ; } a_{i-1}
\end{array}
$$

Curried function application in eval-apply

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] =$
`CLOSURE(CLOSURE(CLOSURE(ACCESS(0);RETURN);RETURN);RETURN);`
`CONST(2) ; APPLY ; CONST(1) ; APPLY ; CONST(0) ; APPLY`

Code	Env	Stack
$\mathtt{CLOSURE}(c_2);a_2$	[]	ε
a_2	[]	$c_2[]$
$\mathtt{APPLY};a_1$	[]	$2.c_2[]$
c_2	2	$a_1.[]$
RETURN	2	$c_1[2].a_1.[]$
a_1	[]	$c_1[2]$
$\mathtt{APPLY};a_0$	[]	$1.c_1[2]$
c_1	1.2	$a_0.[]$
RETURN	1.2	$c_0[1.2].a_0.[]$
a_0	[]	$c_0[1.2]$
APPLY	[]	$0.c_0[1.2]$
c_0	0.1.2	$\varepsilon.[]$
RETURN	0.1.2	$0.\varepsilon.[]$
ε	[]	0

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] = \mathtt{CLOSURE}(c_2);a_2$

where for $i = 1, 2$

$$
\begin{aligned}
c_0 &= \mathtt{ACCESS(0)\ ;\ RETURN} \\
c_i &= \mathtt{CLOSURE}(c_{i-1})\ ;\ \mathtt{RETURN} \\
a_0 &= \mathtt{CONST(0)\ ;\ APPLY} \\
a_i &= \mathtt{CONST(i)\ ;\ APPLY}\ ;\ a_{i-1}
\end{aligned}
$$

Curried function application in eval-apply

$[\![(\lambda.\lambda.\lambda.\underline{0}) 2\,1\,0]\!] =$

`CLOSURE(CLOSURE(CLOSURE(ACCESS(0);RETURN);RETURN);RETURN);`
`CONST(2) ; APPLY ; CONST(1) ; APPLY ; CONST(0) ; APPLY`

Code	Env	Stack
CLOSURE(c_2);a_2	[]	ε
a_2	[]	c_2[]
APPLY;a_1	[]	2.c_2[]
c_2	2	a_1.[]
RETURN	2	c_1[2].a_1.[]
a_1	[]	c_1[2]
APPLY;a_0	[]	1.c_1[2]
c_1	1.2	a_0.[]
RETURN	1.2	c_0[1.2].a_0.[]
a_0	[]	c_0[1.2]
APPLY	[]	0.c_0[1.2]
c_0	0.1.2	ε.[]
RETURN	0.1.2	0.ε.[]
ε	[]	0

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0}) 2\,1\,0]\!] =$ CLOSURE(c_2);a_2

where for $i = 1, 2$

$$
\begin{aligned}
c_0 &= \texttt{ACCESS(0) ; RETURN}\\
c_i &= \texttt{CLOSURE(}c_{i-1}\texttt{) ; RETURN}\\
a_0 &= \texttt{CONST(0) ; APPLY}\\
a_i &= \texttt{CONST(i) ; APPLY ; } a_{i-1}
\end{aligned}
$$

Curried function application in eval-apply

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] =$

```
CLOSURE(CLOSURE(CLOSURE(ACCESS(0);RETURN);RETURN);RETURN);
CONST(2) ; APPLY ; CONST(1) ; APPLY ; CONST(0) ; APPLY
```

Code	Env	Stack
CLOSURE(c_2);a_2	[]	ε
a_2	[]	$c_2[]$
APPLY;a_1	[]	$2.c_2[]$
c_2	2	$a_1.[]$
RETURN	2	$c_1[2].a_1.[]$
a_1	[]	$c_1[2]$
APPLY;a_0	[]	$1.c_1[2]$
c_1	1.2	$a_0.[]$
RETURN	1.2	$c_0[1.2].a_0.[]$
a_0	[]	$c_0[1.2]$
APPLY	[]	$0.c_0[1.2]$
c_0	0.1.2	$\varepsilon.[]$
RETURN	0.1.2	$0.\varepsilon.[]$
ε	[]	0

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] =$ CLOSURE(c_2);a_2

where for $i = 1, 2$

c_0 = ACCESS(0) ; RETURN

c_i = CLOSURE(c_{i-1}) ; RETURN

a_0 = CONST(0) ; APPLY

a_i = CONST(i) ; APPLY ; a_{i-1}

Curried function application in eval-apply

$[\![(\lambda.\lambda.\lambda.\underline{0})\,2\,1\,0]\!] =$

`CLOSURE(CLOSURE(CLOSURE(ACCESS(0);RETURN);RETURN);RETURN);`
`CONST(2) ; APPLY ; CONST(1) ; APPLY ; CONST(0) ; APPLY`

Code	Env	Stack
CLOSURE(c_2);a_2	[]	ε
a_2	[]	c_2[]
APPLY;a_1	[]	2.c_2[]
c_2	2	a_1.[]
RETURN	2	c_1[2].a_1.[]
a_1	[]	c_1[2]
APPLY;a_0	[]	1.c_1[2]
c_1	1.2	a_0.[]
RETURN	1.2	c_0[1.2].a_0.[]
a_0	[]	c_0[1.2]
APPLY	[]	0.c_0[1.2]
c_0	0.1.2	ε.[]
RETURN	0.1.2	0.ε.[]
ε	[]	0

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})\,2\,1\,0]\!] =$ CLOSURE(c_2);a_2

where for $i = 1, 2$

c_0 = ACCESS(0) ; RETURN
c_i = CLOSURE(c_{i-1}) ; RETURN
a_0 = CONST(0) ; APPLY
a_i = CONST(i) ; APPLY ; a_{i-1}

Curried function application in eval-apply

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!]$ =
`CLOSURE(CLOSURE(CLOSURE(ACCESS(0);RETURN);RETURN);RETURN);`
`CONST(2) ; APPLY ; CONST(1) ; APPLY ; CONST(0) ; APPLY`

Code	Env	Stack
CLOSURE(c_2);a_2	[]	ε
a_2	[]	$c_2.[]$
APPLY;a_1	[]	$2.c_2.[]$
c_2	2	$a_1.[]$
RETURN	2	$c_1[2].a_1.[]$
a_1	[]	$c_1[2]$
APPLY;a_0	[]	$1.c_1[2]$
c_1	1.2	$a_0.[]$
RETURN	1.2	$c_0[1.2].a_0.[]$
a_0	[]	$c_0[1.2]$
APPLY	[]	$0.c_0[1.2]$
c_0	0.1.2	$\varepsilon.[]$
RETURN	0.1.2	$0.\varepsilon.[]$
ε	[]	0

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!]$ = CLOSURE(c_2);a_2

where for $i = 1, 2$

c_0 = ACCESS(0) ; RETURN
c_i = CLOSURE(c_{i-1}) ; RETURN
a_0 = CONST(0) ; APPLY
a_i = CONST(i) ; APPLY ; a_{i-1}

Curried function application in eval-apply

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] =$

`CLOSURE(CLOSURE(CLOSURE(ACCESS(0);RETURN);RETURN);RETURN);`
`CONST(2) ; APPLY ; CONST(1) ; APPLY ; CONST(0) ; APPLY`

Code	Env	Stack
CLOSURE(c_2);a_2	[]	ε
a_2	[]	c_2[]
APPLY;a_1	[]	2.c_2[]
c_2	2	a_1.[]
RETURN	2	c_1[2].a_1.[]
a_1	[]	c_1[2]
APPLY;a_0	[]	1.c_1[2]
c_1	1.2	a_0.[]
RETURN	1.2	c_0[1.2].a_0.[]
a_0	[]	c_0[1.2]
APPLY	[]	0.c_0[1.2]
c_0	0.1.2	ε.[]
RETURN	0.1.2	0.ε.[]
ε	[]	0

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] =$ CLOSURE(c_2);a_2

where for $i = 1, 2$

c_0 = ACCESS(0) ; RETURN

c_i = CLOSURE(c_{i-1}) ; RETURN

a_0 = CONST(0) ; APPLY

a_i = CONST(i) ; APPLY ; a_{i-1}

Curried function application in eval-apply

$[\![(\lambda.\lambda.\lambda.\underline{0})\,2\,1\,0]\!]$ =

CLOSURE(CLOSURE(CLOSURE(ACCESS(0);RETURN);RETURN);RETURN);
CONST(2) ; APPLY ; CONST(1) ; APPLY ; CONST(0) ; APPLY

Code	Env	Stack
CLOSURE(c_2);a_2	[]	ε
a_2	[]	c_2.[]
APPLY;a_1	[]	2.c_2.[]
c_2	2	a_1.[]
RETURN	2	c_1[2].a_1.[]
a_1	[]	c_1[2]
APPLY;a_0	[]	1.c_1[2]
c_1	1.2	a_0.[]
RETURN	1.2	c_0[1.2].a_0.[]
a_0	[]	c_0[1.2]
APPLY	[]	0.c_0[1.2]
c_0	0.1.2	ε.[]
RETURN	0.1.2	0.ε.[]
ε	[]	0

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})\,2\,1\,0]\!]$ = CLOSURE(c_2);a_2

where for $i = 1, 2$

$$c_0 = \texttt{ACCESS(0) ; RETURN}$$
$$c_i = \texttt{CLOSURE(}c_{i-1}\texttt{) ; RETURN}$$
$$a_0 = \texttt{CONST(0) ; APPLY}$$
$$a_i = \texttt{CONST(i) ; APPLY ; }a_{i-1}$$

Curried function application in push-enter

$[\![(\lambda.\lambda.\lambda.\underline{0})\,2\,1\,0]\!]$ = `PUSH(0);PUSH(1);PUSH(2);GRAB;GRAB;GRAB;ACCESS(0)`

Curried function application in push-enter

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!]$ = PUSH(0);PUSH(1);PUSH(2);GRAB;GRAB;GRAB;ACCESS(0)

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!]$ = PUSH(0); p_1

where for $i = 1, 2, 3$

$$
\begin{array}{rcl}
g_0 & = & \texttt{ACCESS(0)} \\
g_i & = & \texttt{GRAB} \; ; \; g_{i-1} \\
p_0 & = & \texttt{PUSH(2)} \; ; \; g_3 \\
p_1 & = & \texttt{PUSH(1)} \; ; \; p_0
\end{array}
$$

Curried function application in push-enter

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] = \texttt{PUSH(0);PUSH(1);PUSH(2);GRAB;GRAB;GRAB;ACCESS(0)}$

Code	Env	Stack
$\texttt{PUSH(0)};p_1$	[]	ε
$\texttt{PUSH(1)};p_0$	[]	0[]
$\texttt{PUSH(2)};g_3$	[]	1[].0[]
$\texttt{GRAB};g_2$	[]	2[].1[].0[]
$\texttt{GRAB};g_1$	2[]	1[].0[]
$\texttt{GRAB};g_0$	1[].2[]	0[]
$\texttt{ACCESS(0)}$	0[].1[].2[]	0[]
0	[]	ε

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] = \texttt{PUSH(0)};p_1$

where for $i = 1, 2, 3$

$$
\begin{aligned}
g_0 &= \texttt{ACCESS(0)} \\
g_i &= \texttt{GRAB}\ ;\ g_{i-1} \\
p_0 &= \texttt{PUSH(2)}\ ;\ g_3 \\
p_1 &= \texttt{PUSH(1)}\ ;\ p_0
\end{aligned}
$$

Curried function application in push-enter

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!]$ = PUSH(0);PUSH(1);PUSH(2);GRAB;GRAB;GRAB;ACCESS(0)

Code	Env	Stack
PUSH(0);p_1	[]	ε
PUSH(1);p_0	[]	0[]
PUSH(2);g_3	[]	1[].0[]
GRAB;g_2	[]	2[].1[].0[]
GRAB;g_1	2[]	1[].0[]
GRAB;g_0	1[].2[]	0[]
ACCESS(0)	0[].1[].2[]	0[]
0	[]	ε

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!]$ = PUSH(0);p_1

where for $i = 1, 2, 3$

$$
\begin{array}{rcl}
g_0 & = & \texttt{ACCESS(0)} \\
g_i & = & \texttt{GRAB ; } g_{i-1} \\
p_0 & = & \texttt{PUSH(2) ; } g_3 \\
p_1 & = & \texttt{PUSH(1) ; } p_0
\end{array}
$$

Push-enter clearly wins

Curried function application in push-enter

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] = \texttt{PUSH(0);PUSH(1);PUSH(2);GRAB;GRAB;GRAB;ACCESS(0)}$

Code	Env	Stack
$\texttt{PUSH(0)};p_1$	[]	ε
$\texttt{PUSH(1)};p_0$	[]	0[]
$\texttt{PUSH(2)};g_3$	[]	1[].0[]
$\texttt{GRAB};g_2$	[]	2[].1[].0[]
$\texttt{GRAB};g_1$	2[]	1[].0[]
$\texttt{GRAB};g_0$	1[].2[]	0[]
$\texttt{ACCESS(0)}$	0[].1[].2[]	0[]
0	[]	ε

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] = \texttt{PUSH(0)};p_1$

where for $i = 1,2,3$

$$\begin{array}{rcl} g_0 & = & \texttt{ACCESS(0)} \\ g_i & = & \texttt{GRAB} \ ; \ g_{i-1} \\ p_0 & = & \texttt{PUSH(2)} \ ; \ g_3 \\ p_1 & = & \texttt{PUSH(1)} \ ; \ p_0 \end{array}$$

Push-enter clearly wins

ZAM

Combine the call-by-value semantics with the push-enter model

The ZAM (Zinc Abstract Machine)

(The model underlying the bytecode interpretors of Caml Light and OCaml.)

A call-by-value, push-enter model where the caller pushes one or several arguments on the stack and the callee pops them and put them in its environment.

Needs special handling for

- partial applications: $(\lambda x.\lambda y.b)\ a$
- over-applications: $(\lambda x.x)\ (\lambda x.x)\ a$

The ZAM

Machine Components: as the SECD but where the stack is split into a argument stack and a return stack (as in Landin's original SECD)

Instruction set: as the SECD plus

GRAB	grab argument on the stack OR create a closure
PUSHMARK	push a mark to signal the last argument

minus LET, which is replaced by a GRAB.

Compilation scheme:

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

Notice the left to right evaluation order for function application

Transitions

BEFORE				AFTER			
Code	Env	ArgStack	RetStack	Code	Env	ArgStack	RetStack
$\mathtt{ACCESS}(n);c$	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	$\boxed{*}.s$	r
$\mathtt{GRAB};c$	e	$\boxed{*}.s$	$c'.e'.r$	c'	e'	$(\mathtt{GRAB};c)[e].s$	r
$\mathtt{GRAB};c$	e	$v.s$	r	c	$v.e$	s	r
$\mathtt{RETURN};c$	e	$v.\boxed{*}.s$	$c'.e'.r$	c'	e'	$v.s$	r
$\mathtt{RETURN};c$	e	$c'[e'].s$	r	c'	e'	s	r
$\mathtt{ENDLET};c$	$v.e$	s	r	c	e	s	r

Transitions

BEFORE				AFTER			
Code	Env	ArgStack	RetStack	Code	Env	ArgStack	RetStack
ACCESS(n);c	e	s	r	c	e	$e(n).s$	r
CLOSURE(c');c	e	s	r	c	e	$c'[e].s$	r
TAILAPPLY;c	e	$c'[e'].s$	r	c'	e'	s	r
APPLY;c	e	$c'[e'].s$	r	c'	e'	s	$c.e.r$
PUSHMARK;c	e	s	r	c	e	$\boxed{*}.s$	r
GRAB;c	e	$\boxed{*}.s$	$c'.e'.r$	c'	e'	$(\mathtt{GRAB};c)[e].s$	r
GRAB;c	e	$v.s$	r	c	$v.e$	s	r
RETURN;c	e	$v.\boxed{*}.s$	$c'.e'.r$	c'	e'	$v.s$	r
RETURN;c	e	$c'[e'].s$	r	c'	e'	s	r
ENDLET;c	$v.e$	s	r	c	e	s	r

Nota Bene

1. Having a separate TAILAPPLY command no longer is strictly necessary since it has same behaviour as RETURN and could be replaced by it (we keep it to stress the places where only TAILAPPLY applies).

Transitions

BEFORE				AFTER			
Code	Env	ArgStack	RetStack	Code	Env	ArgStack	RetStack
ACCESS(n);c	e	s	r	c	e	$e(n).s$	r
CLOSURE(c');c	e	s	r	c	e	$c'[e].s$	r
TAILAPPLY;c	e	$c'[e'].s$	r	c'	e'	s	r
APPLY;c	e	$c'[e'].s$	r	c'	e'	s	$c.e.r$
PUSHMARK;c	e	s	r	c	e	$\boxed{*}.s$	r
GRAB;c	e	$\boxed{*}.s$	$c'.e'.r$	c'	e'	$(\texttt{GRAB};c)[e].s$	r
GRAB;c	e	$v.s$	r	c	$v.e$	s	r
RETURN;c	e	$v.\boxed{*}.s$	$c'.e'.r$	c'	e'	$v.s$	r
RETURN;c	e	$c'[e'].s$	r	c'	e'	s	r
ENDLET;c	$v.e$	s	r	c	e	s	r

Nota Bene

1. Having a separate TAILAPPLY command no longer is strictly necessary since it has same behaviour as RETURN and could be replaced by it (we keep it to stress the places where only TAILAPPLY applies).
2. The code produced by $\mathcal{T}[\![a]\!]$ always ends either by RETURN or (equivalently) by TAILAPPLY

Krivine 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:

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

The other ZAM instructions (and the mark $\boxed{*}$, and the return stack) are just extra call-by-value baggage.

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.
For that we compile using a continuation passing style:

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

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

(Facilitates exception handling, stack resizing, etc.)

Transitions

BEFORE			AFTER		
Code	Env	Stack	Code	Env	Stack
GRAB; c	e	$v.s$	c	$v.e$	s
GRAB; c	e	$\boxed{*}.c'.e'.s$	c'	e'	$(\texttt{GRAB}; c)[e].s$
RETURN; c	e	$v.\boxed{*}.c'.e'.s$	c'	e'	$v.s$
RETURN; c	e	$c'[e'].s$	c'	e'	s
PUSHRETADDR(c'); c	e	s	c	e	$\boxed{*}.c'.e.s$
TAILAPPLY; c	e	$c'[e'].s$	c'	e'	s
ACCESS(n); c	e	s	c	e	$e(n).s$
ENDLET; c	$v.e$	s	c	e	s
CLOSURE(c'); c	e	s	c	e	$c'[e].s$

Once more we can use RETURN instead of TAILAPPLY

Handling of curried applications

Consider the code in the closure for $\lambda.\lambda.\lambda.a$:

$$\texttt{GRAB; GRAB; GRAB; } \mathcal{T}[\![a]\!]$$

and recall that $\mathcal{T}[\![a]\!]$ finishes by a RETURN (or equivalently by a TAILAPPLY)

- **Total application to 3 arguments:**
  - The stack on entry is $v_1.v_2.v_3.\boxed{*}.c'.e'$
  - The three GRABs succeed yielding an environment $v_3.v_2.v_1.e$.
  - $\mathcal{T}[\![a]\!]$ is executed. It produces a value v and finishes by a RETURN
  - RETURN sees the stack $v.\boxed{*}.c'.e'$, reinstalls the caller $c'[e']$, and returns v to it
- **Partial application to 2 arguments:**
  - The stack on entry is $v_1.v_2.\boxed{*}.c'.e'$
  - The third GRAB fails and returns $(\texttt{GRAB};\mathcal{T}[\![a]\!])[v_2.v_1.e]$, representing the result of the partial application.
- **Over-application to 4 arguments:**
  The stack on entry is $v_1.v_2.v_3.v_4.\boxed{*}.c'.e'$
  - The three GRABs succeed yielding an environment $v_3.v_2.v_1.e$.
  - $\mathcal{T}[\![a]\!]$ is executed. It produces a value v and finishes by a RETURN
  - RETURN sees the stack $v.v_4.\boxed{*}.c'.e'$, and tail-applies v to v_4

  (v should be a closure or otherwise the over-application would be wrong and the machine stuck).

Outline

Stackeless Machine for CPS terms

The λ-terms produced by the CPS transformation have the following form:

$$\begin{array}{lcll} a & ::= & \underline{n} \mid N \mid \lambda.b \mid \lambda\lambda.b & \textbf{CPS atom} \\ b & ::= & a \mid a_1\, a_2 \mid a_1\, a_2\, a_3 & \textbf{CPS body} \end{array}$$

Machine Components:

A stackless abstract machine with:

- a code pointer c
- an environment e
- three registers R_1, R_2, R_3.

Instruction set:

$\texttt{ACCESS}_i\texttt{(n)}$	store n-th field of the environment in R_i
$\texttt{CONST}_i(N)$	store the integer N in R_i
$\texttt{CLOSURE}_i\texttt{(c)}$	store closure of c in R_i
TAILAPPLY1	apply closure in R_1 to argument R_2
TAILAPPLY2	apply closure in R_1 to arguments R_2, R_3

Compilation scheme

Compilation of atoms $\mathcal{A}_i[\![a]\!]$ (leaves the value of a in R_i):

$$
\begin{array}{rcl}
\mathcal{A}_i[\![\underline{n}]\!] & = & \mathtt{ACCESS}_i\mathtt{(}n\mathtt{)} \\
\mathcal{A}_i[\![N]\!] & = & \mathtt{CONST}_i\mathtt{(}N\mathtt{)} \\
\mathcal{A}_i[\![\lambda.b]\!] & = & \mathtt{CLOSURE}_i\mathtt{(}\mathcal{B}[\![b]\!]\mathtt{)} \\
\mathcal{A}_i[\![\lambda\lambda.b]\!] & = & \mathtt{CLOSURE}_i\mathtt{(}\mathcal{B}[\![b]\!]\mathtt{)}
\end{array}
$$

Compilation of bodies $\mathcal{B}[\![b]\!]$:

$$
\begin{array}{rcl}
\mathcal{B}[\![a]\!] & = & \mathcal{A}_1[\![a]\!] \\
\mathcal{B}[\![a_1\,a_2]\!] & = & \mathcal{A}_1[\![a_1]\!];\mathcal{A}_2[\![a_2]\!];\mathtt{TAILAPPLY1} \\
\mathcal{B}[\![a_1\,a_2\,a_3]\!] & = & \mathcal{A}_1[\![a_1]\!];\mathcal{A}_2[\![a_2]\!];\mathcal{A}_3[\![a_3]\!];\mathtt{TAILAPPLY2}
\end{array}
$$

Transitions

BEFORE					AFTER				
Code	Env	R_1	R_2	R_3	Code	Env	R_1	R_2	R_3
TAILAPPLY1; c	e	$c'[e']$	v	-	c'	$v.e'$	-	-	-
TAILAPPLY2; c	e	$c'[e']$	v_1	v_2	c'	$v_2.v_1.e'$	-	-	-
ACCESS$_1$$(n)$; c	e	—	v_2	v_3	c	e	$e(n)$	v_2	v_3
CONST$_1$$(N)$; c	e	—	v_2	v_3	c	e	N	v_2	v_3
CLOSURE$_1$$(c')$; c	e	—	v_2	v_3	c	e	$c'[e]$	v_2	v_3
ACCESS$_2$$(n)$; c	e	—	v_2	v_3	c	e	v_1	$e(n)$	v_3
CONST$_2$$(N)$; c	e	—	v_2	v_3	c	e	v_1	N	v_3
CLOSURE$_2$$(c')$; c	e	—	v_2	v_3	c	e	v_1	$c'[e]$	v_3
ACCESS$_3$$(n)$; c	e	—	v_2	v_3	c	e	v_1	v_2	$e(n)$
CONST$_3$$(N)$; c	e	—	v_2	v_3	c	e	v_1	v_2	N
CLOSURE$_3$$(c')$; c	e	—	v_2	v_3	c	e	v_1	v_2	$c'[e]$

That CPS terms can be executed without a stack is not surprising, given that the stack of a machine such as the SECD is isomorphic to the current continuation in a CPS-based approach.

```
f x = 1 + g x      g x = 2 - h x      h x = ...
```

Consider the execution point where h is entered. In the CPS model, the continuation at this point is

$$k = \lambda v.k'(2-v) \text{ with } k' = \lambda v.k''(1+v) \text{ and } k'' = \lambda v.v$$

In the SECD model, the stack at this point is

$$\underbrace{(\mathtt{SUB}\ ;\ \mathtt{RETURN}).e_g.2}_{\simeq k}.\underbrace{(\mathtt{ADD}\ ;\ \mathtt{RETURN}).e_f.1}_{\simeq k'}.\underbrace{\varepsilon}_{\simeq k''}$$

Continuations vs. stacks

At the machine level, stacks and continuations are two ways to represent the call chain: the chain of function calls currently active.

- Continuations: as a singly-linked list of heap-allocated closures, each closure representing a function activation (in the example of the previous slide $k \mapsto \lambda v.k'(2-v)[k' \mapsto \lambda v.k''(1+v)[k'' \mapsto \lambda v.v]]$).
  These closures are reclaimed by the garbage collector.
- Stacks: as contiguous blocks in a memory area outside the heap, each block representing a function activation. These blocks are explicitly deallocated by RETURN instructions.

Stacks are more efficient in terms of GC costs and memory locality, but need to be copied in full to implement `callcc`.

References

- Jean-Louis Krivine: A call-by-name lambda-calculus machine. Higher-Order and Symbolic Computation 20(3):199-207 (2007)
- Improving the lazy Krivine machine, by D. Friedman, A. Ghuloum, J. Siek, O. Winebarger. Higher-Order Symb Comput (2007)20:271-293.
- Making a fast curry: push/enter vs. eval/apply for higher-order languages by Simon Marlow and Simon Peyton Jones. ICFP '04 and JFP '06
- A. Appel. Compiling with continuations (Chapter 13).
- Simon Peyton Jones. The Spineless Tagless G machine. 1992 (the original STG virtual machine for Haskell, now outdated).
- Slides of the course *Functional Programming Languages* by Xavier Leroy (from which the slides of this and the following part **heavily** borrowed) available on the web: `https://xavierleroy.org/mpri/2-4/machines.2up.pdf`