Functional programming languages
Part II: abstract machines

Xavier Leroy

INRIA Rocquencourt

MPRI 2-4-2, 2006

Execution models for a programming 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 programming 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.

Outline

An abstract machine for arithmetic expressions
(Warm-up exercise)

Arithmetic expressions:

$$a ::= N \mid a_1 + a_2 \mid a_1 - a_2 \mid \ldots$$

The machine uses a stack to store intermediate results during expression evaluation. (Cf. Hewlett-Packard pocket calculators.)

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":

$$
\begin{aligned}
\mathcal{C}(N) &= \texttt{CONST}(N) \\
\mathcal{C}(a_1 + a_2) &= \mathcal{C}(a_1); \mathcal{C}(a_2); \texttt{ADD} \\
\mathcal{C}(a_1 - a_2) &= \mathcal{C}(a_1); \mathcal{C}(a_2); \texttt{SUB}
\end{aligned}
$$

Example 1

$\mathcal{C}(5 - (1 + 2)) = \texttt{CONST}(5); \texttt{CONST}(1); \texttt{CONST}(2); \texttt{ADD}; \texttt{SUB}$

Transitions of the abstract machine

The machine has two components:

- a code pointer c (the instructions yet to be executed)
- a stack s (holding intermediate results).

Machine state before		Machine state after	
Code	Stack	Code	Stack
$\texttt{CONST}(N); c$	s	c	$N.s$
$\texttt{ADD}; c$	$n_2.n_1.s$	c	$(n_1 + n_2).s$
$\texttt{SUB}; c$	$n_2.n_1.s$	c	$(n_1 - n_2).s$

Notations for stacks: top of stack is to the left.

push v on s: $s \longrightarrow v.s$ pop v off s: $v.s \longrightarrow s$

Evaluating expressions with the abstract machine

Initial state: code $= \mathcal{C}(a)$ and stack $= \varepsilon$.

Final state: code $= \varepsilon$ and stack $= n.\varepsilon$.
The result of the computation is the integer n (top of stack at end of execution).

Example 2

Code	Stack
CONST(5); CONST(1); CONST(2); ADD; SUB	ε
CONST(1); CONST(2); ADD; SUB	$5.\varepsilon$
CONST(2); ADD; SUB	$1.5.\varepsilon$
ADD; SUB	$2.1.5.\varepsilon$
SUB	$3.5.\varepsilon$
ε	$2.\varepsilon$

Executing abstract machine code: by interpretation

The interpreter is typically written in a low-level language such as C and executes 5 times faster than a term interpreter (typically).

```
int interpreter(int * code)
{
  int * s = bottom_of_stack;
  while (1) {
    switch (*code++) {
      case CONST:   *s++ = *code++; break;
      case ADD:     s[-2] = s[-2] + s[-1]; s--; break;
      case SUB:     s[-2] = s[-2] - s[-1]; s--; break;
      case EPSILON: return s[-1];
    }
  }
}
```

Executing abstract machine code: by expansion

Alternatively, abstract instructions can be expanded into canned sequences
for a real processor, giving an additional speedup by a factor of 5
(typically).

```
CONST(i)        --->    pushl $i

ADD             --->    popl %eax
                        addl 0(%esp), %eax

SUB             --->    popl %eax
                        subl 0(%esp), %eax

EPSILON         --->    popl %eax
                        ret
```

Outline

The Modern SECD: An abstract machine for call-by-value

Three components in this machine:

- a code pointer c (the instructions yet to be executed)
- an environment e (giving values to variables)
- a stack s (holding intermediate results and return addresses).

Instruction set ($+$ arithmetic operations as before):

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

Compilation scheme

Compilation scheme:

$$
\begin{aligned}
\mathcal{C}(\underline{n}) &= \text{ACCESS}(n) \\
\mathcal{C}(\lambda a) &= \text{CLOSURE}(\mathcal{C}(a); \text{RETURN}) \\
\mathcal{C}(\text{let } a \text{ in } b) &= \mathcal{C}(a); \text{LET}; \mathcal{C}(b); \text{ENDLET} \\
\mathcal{C}(a\ b) &= \mathcal{C}(a); \mathcal{C}(b); \text{APPLY}
\end{aligned}
$$

Constants and arithmetic: as before.

> **Example 3**
>
> Source term: $(\lambda x.\ x + 1)\ 2$.
> Code: CLOSURE(ACCESS(1); CONST(1); ADD; RETURN); CONST(2); APPLY.

Machine transitions

Machine state before			Machine state after		
Code	Env	Stack	Code	Env	Stack
$\texttt{ACCESS}(n); c$	e	s	c	e	$e(n).s$
$\texttt{LET}; c$	e	$v.s$	c	$v.e$	s
$\texttt{ENDLET}; c$	$v.e$	s	c	e	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'	$v.s$

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

Example of evaluation

Initial code $\texttt{CLOSURE}(c); \texttt{CONST}(2); \texttt{APPLY}$
where $c = \texttt{ACCESS}(1); \texttt{CONST}(1); \texttt{ADD}; \texttt{RETURN}$.

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

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 do except return 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, 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.

The importance of tail call elimination

Tail call elimination is important for recursive functions of the following form — 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
```

The recursive call to fact is in tail position. With tail call elimination, this code runs in constant stack space. Without, it consumes $O(n)$ stack space and risks stack overflow.

Compare with the standard definition of fact, which is not tail recursive and runs in $O(n)$ stack space:

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

Tail call elimination in the Modern SECD

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

$$
\begin{aligned}
\mathcal{T}(\texttt{let } a \texttt{ in } b) &= \mathcal{C}(a); \texttt{LET}; \mathcal{T}(b) \\
\mathcal{T}(a\ b) &= \mathcal{C}(a); \mathcal{C}(b); \texttt{TAILAPPLY} \\
\mathcal{T}(a) &= \mathcal{C}(a); \texttt{RETURN} \qquad \text{(otherwise)}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}(\underline{n}) &= \texttt{ACCESS}(n) \\
\mathcal{C}(\lambda a) &= \texttt{CLOSURE}(\mathcal{T}(a)) \\
\mathcal{C}(\texttt{let } a \texttt{ in } b) &= \mathcal{C}(a); \texttt{LET}; \mathcal{C}(b); \texttt{ENDLET} \\
\mathcal{C}(a\ b) &= \mathcal{C}(a); \mathcal{C}(b); \texttt{APPLY}
\end{aligned}
$$

Tail call elimination in the Modern SECD

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

Machine state before			Machine state 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'	$v.e'$	$c.e.s$

Krivine's machine: An abstract machine for call-by-name

As for the Modern SECD, three components in this machine:

- Code c
- Environment e
- Stack s

However, stack and environment no longer contain values, but thunks: closures $c[e]$ representing expressions (function arguments) whose evaluations are delayed until their value is needed.

Consistent with the β-reduction rule for call by name:

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

Compilation scheme

$$
\begin{aligned}
\mathcal{C}(\underline{n}) &= \texttt{ACCESS}(n) \\
\mathcal{C}(\lambda a) &= \texttt{GRAB}; \mathcal{C}(a) \\
\mathcal{C}(a\ b) &= \texttt{PUSH}(\mathcal{C}(b)); \mathcal{C}(a)
\end{aligned}
$$

Instruction set:

ACCESS(N)	start evaluating the N-th thunk found in the environment
PUSH(c)	push a thunk for code c
GRAB	pop one argument and add it to the environment

Transitions of Krivine's machine

Machine state before			Machine state after		
Code	Env	Stack	Code	Env	Stack
$\texttt{ACCESS}(n); c$	e	s	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	$c'[e].s$

Initial state: $\text{code} = \mathcal{C}(a)$, stack $= \varepsilon$.
Final state: $\text{code} = \texttt{GRAB}; c$, stack $= \varepsilon$.

How does it work?

The stack encodes the spine of applications in progress.
The code and environment encode the term at the bottom left of the spine.
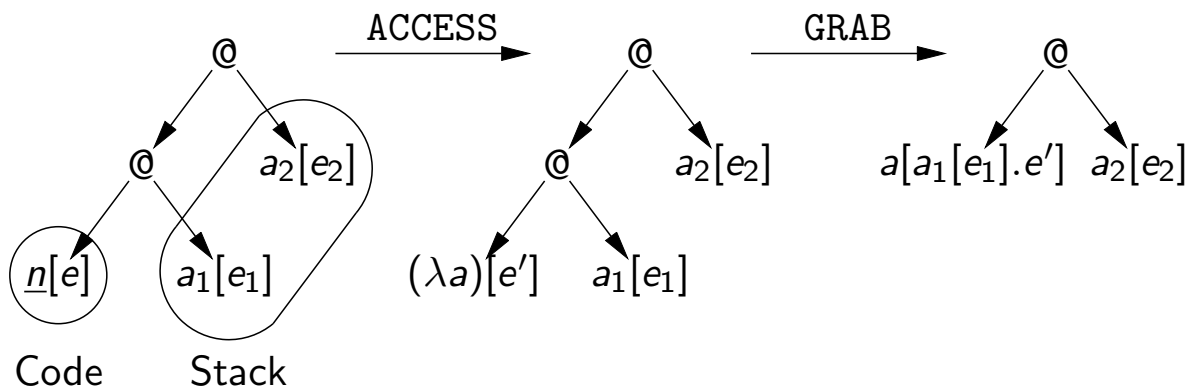
Call-by-name in practice

Realistic abstract machines for call-by-name are more complex than Krivine's machine in two respects:

- Constants and primitive operations:
  Operations such as addition are strict: they must evaluate fully their arguments before reducing. Extra mechanisms are needed to force evaluation of sub-expressions to values.

- Lazy evaluation, i.e. sharing of computations:
  Call-by-name evaluates an expression every time its value is needed. Lazy evaluation performs the evaluation the first time, then caches the result for later uses.

See: *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, S.L. Peyton Jones, Journal of Functional Programming 2(2), Apr 1992.

Eval-apply vs. push-enter

The SECD and Krivine's machine illustrate two subtly different ways to evaluate function applications $f\ a$:
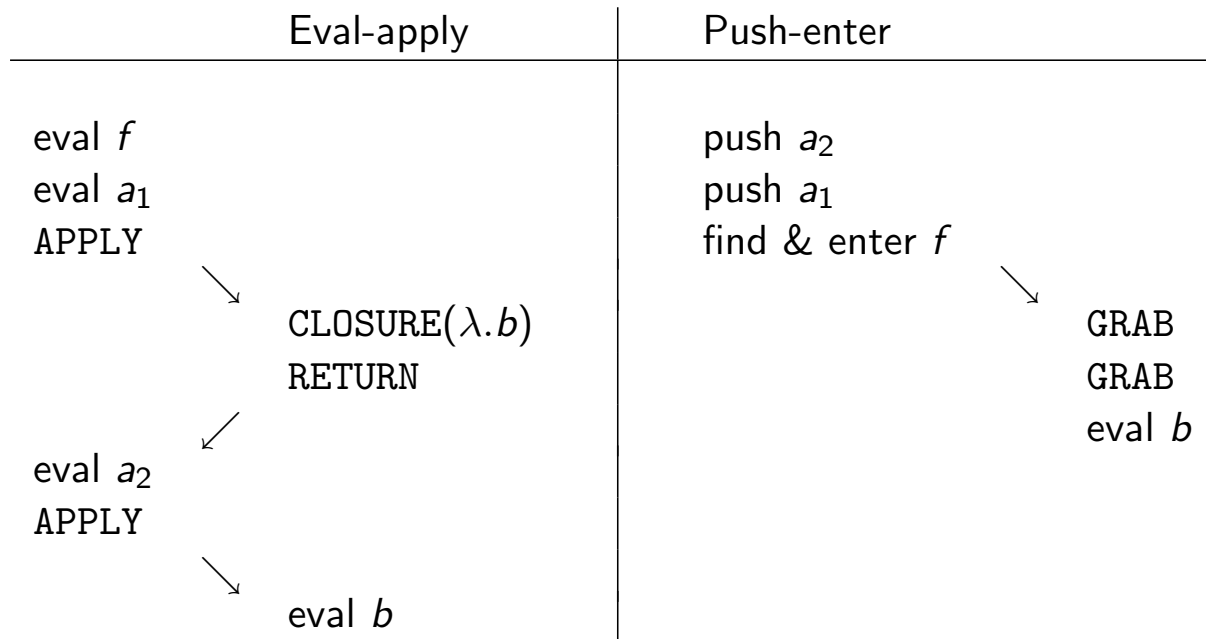
- Eval-apply: (e.g. SECD)
  Evaluate f to a closure $c[e]$, evaluate a, extend environment e', jump to c.
  The β-reduction is performed by the caller.

- Push-enter: (e.g. Krivine but also Postscript, Forth)
  Push a on stack, evaluate f to a closure $c[e]$, jump to c, pop argument, extend environment e with it.
  The β-reduction is performed by the callee.

The difference becomes significant for curried function applications

$$f\ a_1\ a_2\ \ldots\ a_n = \ldots ((f\ a_1)\ a_2)\ \ldots\ a_n \qquad \text{where } f = \lambda \ldots \lambda b$$

Eval-apply vs. push-enter for curried applications

Consider $f\ a_1\ a_2$ where $f = \lambda.\lambda.b$.

Eval-apply	Push-enter
eval f eval a_1 APPLY	push a_2 push a_1 find & enter f
CLOSURE($\lambda.b$) RETURN	GRAB GRAB eval b
eval a_2 APPLY	
eval b	

Eval-apply vs. push-enter for curried applications

Compared with push-enter, eval-apply of a n-argument curried application performs extra work:

- Jumps $n - 1$ times from caller to callee and back
  (the sequences APPLY; CLOSURE; RETURN).
- Builds $n - 1$ short-lived intermediate closures.

Can we combine push-enter and call-by-value? Yes, see the ZAM.

The ZAM (Zinc abstract machine)

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

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.

Since this is call-by-value, need special handling for

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

Compilation scheme

$\mathcal{T}$ for expressions in tail call position, $\mathcal{C}$ for other expressions.

$$
\begin{aligned}
\mathcal{T}(\lambda.a) &= \text{GRAB}; \mathcal{T}(a) \\
\mathcal{T}(\text{let } a \text{ in } b) &= \mathcal{C}(a); \text{GRAB}; \mathcal{T}(b) \\
\mathcal{T}(a\ a_1\ \ldots\ a_n) &= \mathcal{C}(a_n); \ldots; \mathcal{C}(a_1); \mathcal{T}(a) \\
\mathcal{T}(a) &= \mathcal{C}(a); \text{RETURN} \qquad \text{(otherwise)}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}(\underline{n}) &= \text{ACCESS}(n) \\
\mathcal{C}(\lambda.a) &= \text{CLOSURE}(\mathcal{T}(a)) \\
\mathcal{C}(\text{let } a \text{ in } b) &= \mathcal{C}(a); \text{GRAB}; \mathcal{C}(b); \text{ENDLET}; k \\
\mathcal{C}(a\ a_1\ \ldots\ a_n) &= \text{PUSHRETADDR}(k); \mathcal{C}(a_n); \ldots; \mathcal{C}(a_1); \mathcal{C}(a); \text{APPLY} \\
&\quad \text{where } k \text{ is the code that follows the APPLY}
\end{aligned}
$$

Note right-to-left evaluation of applications.

ZAM transitions

$\square$ is a special value ("marker") delimitating applications in the stack.

Machine state before			Machine state after		
Code	Env	Stack	Code	Env	Stack
$\texttt{GRAB}; c$	e	$v.s$	c	$v.e$	s
$\texttt{GRAB}; c$	e	$\square.c'.e'.s$	c'	e'	$(\texttt{GRAB}; c)[e].s$
$\texttt{RETURN}; c$	e	$v.\square.c'.e'.s$	c'	e'	$v.s$
$\texttt{RETURN}; c$	e	$c'[e'].s$	c'	e'	s
$\texttt{PUSHRETADDR}(c'); c$	e	s	c	e	$\square.c'.e.s$
$\texttt{APPLY}; c$	e	$c'[e'].s$	c'	e'	s

$\texttt{ACCESS, CLOSURE, ENDLET}$: like in the Modern SECD.

Handling of applications

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

$$\texttt{GRAB}; \texttt{GRAB}; \texttt{GRAB}; \mathcal{C}(a); \texttt{RETURN}$$

- Total application to 3 arguments:
  stack on entry is $v_1.v_2.v_3.\square.c'.e'$.
  The three GRAB succeed $\rightarrow$ environment $v_3.v_2.v_1.e$.
  RETURN sees the stack $v.\square.c'.e'$ and returns v to caller.
- Partial application to 2 arguments:
  stack on entry is $v_1.v_2.\square.c'.e'$.
  The third GRAB fails and returns $(\texttt{GRAB}; \mathcal{C}(a); \texttt{RETURN})[v_2.v_1.e]$,
  representing the result of the partial application.
- Over-application to 4 arguments:
  stack on entry is $v_1.v_2.v_3.v_4.\square.c'.e'$.
  RETURN sees the stack $v.v_4.\square.c'.e'$ and tail-applies v (which better has be a closure) to v_4.

Outline

Correctness proofs for abstract machines

At this point of the lecture, we have two ways to execute a given source term:

1. Evaluate directly the term: $a \xrightarrow{*} v$ or $\varepsilon \vdash a \Rightarrow v$.

2. Compile it, then execute the resulting code using the abstract machine:

$$\begin{pmatrix} \text{code} = \mathcal{C}(a) \\ \text{env} = \varepsilon \\ \text{stack} = \varepsilon \end{pmatrix} \xrightarrow{*} \begin{pmatrix} \text{code} = \varepsilon \\ \text{env} = e \\ \text{stack} = v.\varepsilon \end{pmatrix}$$

Do these two execution paths agree? Does the abstract machine compute the correct result, as predicted by the semantics of the source term?

Total correctness for Krivine's machine

We start with Krivine's machine because it enjoys a very nice property:

> *Every transition of Krivine's machine simulates one reduction step in the call-by-name λ-calculus with explicit substitutions.*

To make the simulation explicit, we first extend the compilation scheme $\mathcal{C}$ as follows:

$$\mathcal{C}(a[e]) = \mathcal{C}(a)[\mathcal{C}(e)]$$

(a term a viewed under substitution e compiles down to a machine thunk)

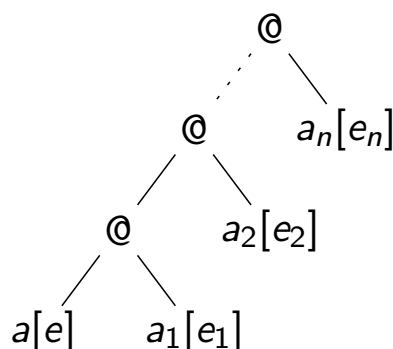$$\mathcal{C}(e) = \mathcal{C}(a_1[e_1])\ldots\mathcal{C}(a_n[e_n]) \quad \text{if } e = a_1[e_1]\ldots a_n[e_n]$$

(a substitution e of thunks for de Bruijn variables compiles down to a machine environment)
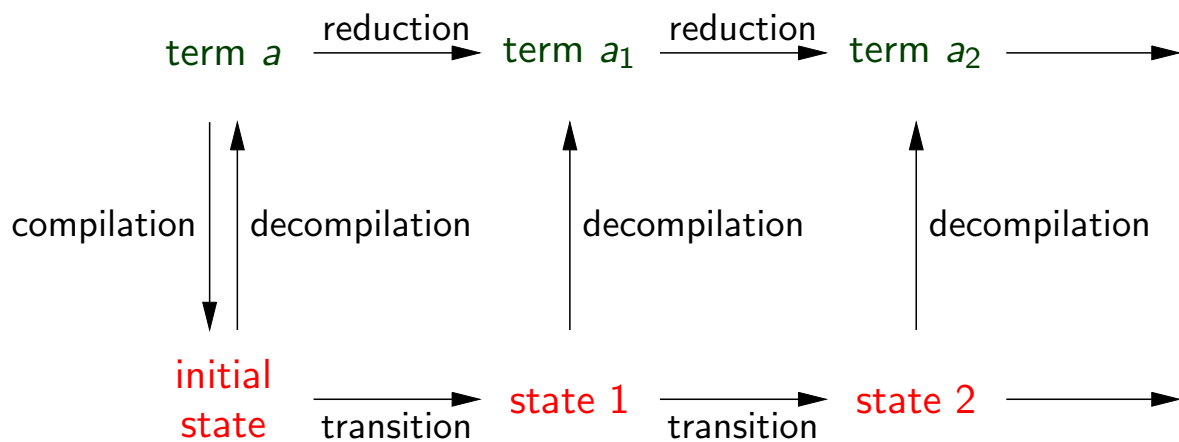
Decompiling states of Krivine's machine

A state of the machine of the following form

$$
\begin{aligned}
\text{code} &= \mathcal{C}(a) \\
\text{env} &= \mathcal{C}(e) \\
\text{stack} &= \mathcal{C}(a_1)[\mathcal{C}(e_1)]\ldots\mathcal{C}(a_n)[\mathcal{C}(e_n)]
\end{aligned}
$$

decompiles to the following source-level term:

Decompilation and simulation

The simulation lemma

Lemma 4 (Simulation)

If the machine state (c, e, s) decompiles to the source term a, and if the machine makes a transition $(c, e, s) \to (c', e', s')$, then there exists a term a' such that

1. $a \to a'$ *(reduction in the CBN λ-calculus with explicit substitutions)*
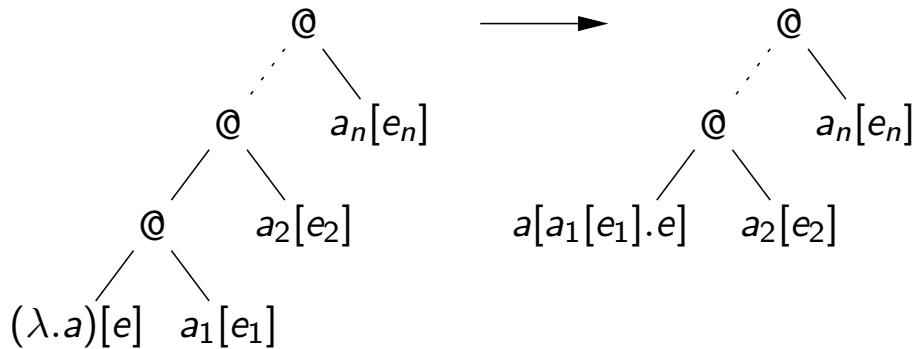2. (c', e', s') *decompiles to a'.*

Proof.

By case analysis on the machine transition. (Next 3 slides). □

The simulation lemma - GRAB case

The transition is:

$$(\texttt{GRAB}; \mathcal{C}(a), \quad \mathcal{C}(e), \quad \mathcal{C}(a_1)[\mathcal{C}(e_1)] \ldots \mathcal{C}(a_n)[\mathcal{C}(e_n)])$$
$$\downarrow$$
$$(\mathcal{C}(a), \quad \mathcal{C}(a_1[e_1].e), \quad \mathcal{C}(a_2)[\mathcal{C}(e_2)] \ldots \mathcal{C}(a_n)[\mathcal{C}(e_n)])$$

It corresponds to a β-reduction $(\lambda.a)[e]\ a_1[e_1] \to a[a_1[e_1].e]$:

The simulation lemma - PUSH case

The transition is:

$$(\texttt{PUSH}(\mathcal{C}(b)); \mathcal{C}(a), \quad \mathcal{C}(e), \quad \mathcal{C}(a_1)[\mathcal{C}(e_1)] \ldots \mathcal{C}(a_n)[\mathcal{C}(e_n)])$$
$$\downarrow$$
$$(\mathcal{C}(a), \quad \mathcal{C}(e), \quad \mathcal{C}(b)[\mathcal{C}(e)].\mathcal{C}(a_1)[\mathcal{C}(e_1)] \ldots \mathcal{C}(a_n)[\mathcal{C}(e_n)])$$

It corresponds to a reduction $(a\ b)[e] \to a[e]\ b[e]$:
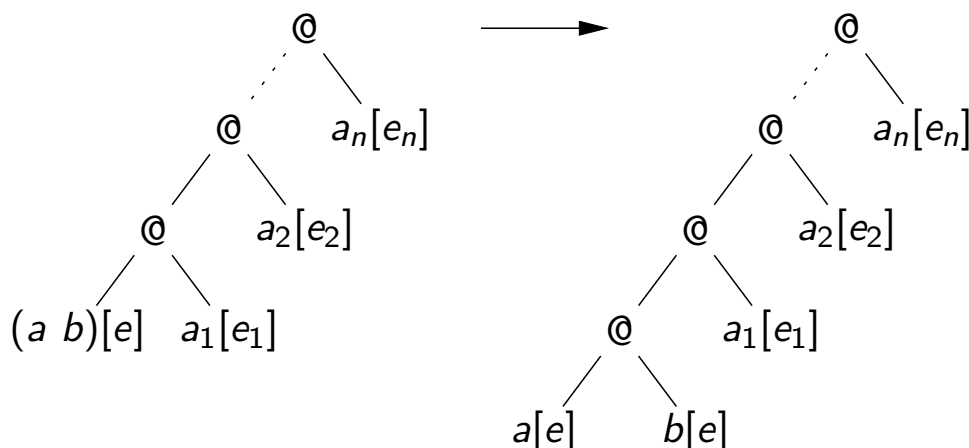
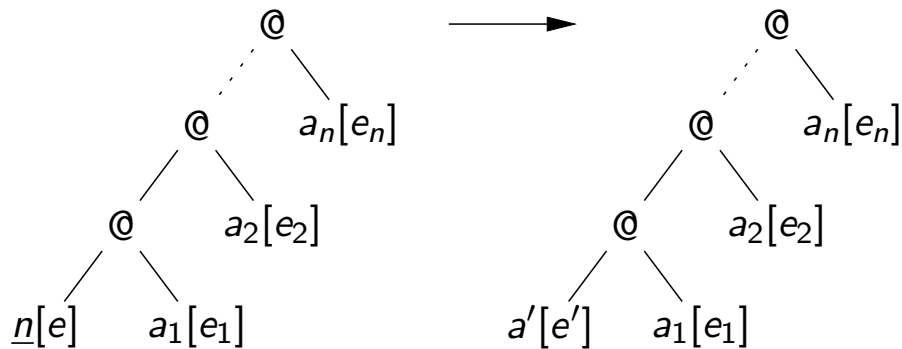# The simulation lemma - ACCESS case

The transition is:

$$(\mathtt{ACCESS}(n), \quad \mathcal{C}(e), \quad \mathcal{C}(a_1)[\mathcal{C}(e_1)]\ldots\mathcal{C}(a_n)[\mathcal{C}(e_n)])$$
$$\downarrow$$
$$(\mathcal{C}(a'), \quad \mathcal{C}(e'), \quad \mathcal{C}(a_1)[\mathcal{C}(e_1)]\ldots\mathcal{C}(a_n)[\mathcal{C}(e_n)])$$

if $e(n) = a'[e']$. It corresponds to a reduction $\underline{n}[e] \to e(n)$:

Other lemmas

Lemma 5 (Progress)

If the state (c, e, s) decompiles to the term a, and a can reduce, then the machine can make one transition from (c, e, s).

Lemma 6 (Initial states)

The initial state $(\mathcal{C}(a), \varepsilon, \varepsilon)$ decompiles to the term a.

Lemma 7 (Final state)

A final state of the form $(\mathtt{GRAB}; \mathcal{C}(a),\ \mathcal{C}(e),\ \varepsilon)$ decompiles to the value $(\lambda.a)[e]$.

The correctness theorem

Theorem 8 (Total correctness of Krivine's machine)

If we start the machine in initial state $(\mathcal{C}(a), \varepsilon, \varepsilon)$,

1. *the machine terminates on a final state (c, e, s) if and only if $a \xrightarrow{*} v$ and the final state (c, e, s) decompiles to the value v;*

2. *the machine performs an infinite number of transitions if and only if a reduces infinitely.*

Proof.

By the initial state and simulation lemmas, all intermediate machine states correspond to reducts of a. If the machine never stops, we are in case 2. If the machine stops, by the progress lemma, it must be because the corresponding term is irreducible. The final state lemma shows that we are in case 1. $\qquad\square$

Partial correctness for the Modern SECD

Total correctness for the Modern SECD is significantly harder to prove than for Krivine's machine. It is however straightforward to prove partial correctness, i.e. restrict ourselves to terminating source programs:

Theorem 9 (Partial correctness of the Modern SECD)

If $a \xrightarrow{*} v$ under call-by-value, then the machine started in state $(\mathcal{C}(a), \varepsilon, \varepsilon)$ terminates in state $(\varepsilon, \varepsilon, v'.\varepsilon)$, and the machine value v' corresponds with the source value v. In particular, if v is an integer N, then $v' = N$.

The key to a simple proof is to use natural semantics $e \vdash a \Rightarrow v$ instead of the reduction semantics $a[e] \xrightarrow{*} v$.

Compositionality and natural semantics

The compilation scheme is compositional: every sub-term a' of the program a is compiled to a code sequence that evaluates a and leaves its value on the top of the stack.

This follows exactly an evaluation derivation of $e \vdash a \Rightarrow v$ in natural semantics. This derivation contains sub-derivations $e' \vdash a' \Rightarrow v'$ for each sub-term a'.

Partial correctness using natural semantics

> **Theorem 10 (Partial correctness of the Modern SECD)**
>
> If $e \vdash a \Rightarrow v$, then
> $$\begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix} \xrightarrow{+} \begin{pmatrix} k \\ \mathcal{C}(e) \\ \mathcal{C}(v).s \end{pmatrix}$$

The compilation scheme $\mathcal{C}$ is extended to values and environments as follows:

$$\mathcal{C}(N) = N$$
$$\mathcal{C}((\lambda a)[e]) = (\mathcal{C}(a); \text{RETURN})[\mathcal{C}(e)]$$
$$\mathcal{C}(v_1 \ldots v_n.\varepsilon) = \mathcal{C}(v_1) \ldots \mathcal{C}(v_n).\varepsilon$$

Partial correctness using natural semantics

The proof of the partial correctness theorem proceeds by induction over the derivation of $e \vdash a \Rightarrow v$ and case analysis on the last rule used.

The cases $a = N$, $a = \underline{n}$ and $a = \lambda.b$ are straightforward: the machine performs exactly one CONST, ACCESS or CLOSURE transition in these cases.

The interesting case is that of function application:

$$\frac{e \vdash a \Rightarrow (\lambda c)[e'] \quad e \vdash b \Rightarrow v' \quad v'.e' \vdash c \Rightarrow v}{e \vdash a\ b \Rightarrow v}$$

(The let rule is similar.)

$$(\mathcal{C}(a); \mathcal{C}(b); \text{APPLY}; k \ | \ \mathcal{C}(e) \ | \ s)$$

$\downarrow +$ (induction hypothesis on first premise)

$$(\mathcal{C}(b); \text{APPLY}; k \ | \ \mathcal{C}(e) \ | \ (\mathcal{C}(c); \text{RETURN})[\mathcal{C}(e')].s)$$

$\downarrow +$ (induction hypothesis on second premise)

$$(\text{APPLY}; k \ | \ \mathcal{C}(e) \ | \ \mathcal{C}(v').(\mathcal{C}(c); \text{RETURN})[\mathcal{C}(e')].s)$$

$\downarrow$ (APPLY transition)

$$(\mathcal{C}(c); \text{RETURN} \ | \ \mathcal{C}(v'.e') \ | \ k.\mathcal{C}(e).s)$$

$\downarrow +$ (induction hypothesis on third premise)

$$(\text{RETURN} \ | \ \mathcal{C}(v'.e') \ | \ \mathcal{C}(v).k.\mathcal{C}(e).s)$$

$\downarrow$ (RETURN transition)

$$(k \ | \ \mathcal{C}(e) \ | \ \mathcal{C}(v).s)$$

Total correctness for the Modern SECD

The partial correctness theorem applies only to terminating source terms. But for terms a that diverge or get stuck, $e \vdash a \Rightarrow v$ does not hold for any e, v and the theorem does not apply.

We do not know what the machine is going to do when started on such terms.

(The machine could loop, could get stuck, but could as well stop and answer "42".)

Total correctness for the Modern SECD

To obtain a stronger correctness result, we can try to show a simulation result similar to that for Krivine's machine. However, decompilation of Modern SECD machine states is significantly complicated by the following fact:

> There are intermediate states of the Modern SECD where the code component is not the compilation of any source term, e.g.

$$code = \mathtt{APPLY}; k \neq \mathcal{C}(a) \quad \text{for all } a$$

$\Rightarrow$ Define decompilation by symbolic execution

Warm-up: symbolic execution for the HP calculator

Consider the following alternate semantics for the abstract machine:

Machine state before		Machine state after	
Code	Stack	Code	Stack
$\texttt{CONST}(N); c$	s	c	$N.s$
$\texttt{ADD}; c$	$a_2.a_1.s$	c	$+ \qquad .s$ $\swarrow \quad \searrow$ $a_1 \qquad a_2$
$\texttt{SUB}; c$	$a_2.a_1.s$	c	$- \qquad .s$ $\swarrow \quad \searrow$ $a_1 \qquad a_2$

The stack contains arithmetic expressions instead of integers.
The instruction ADD, SUB construct arithmetic expressions instead of performing integer computations.

Warm-up: symbolic execution for the HP calculator

To decompile the machine state (c, s), we execute the code c with the symbolic machine, starting in the stack s (viewed as a stack of expressions rather than a stack of integers).
If the symbolic machine stops with code $= \varepsilon$ and stack $= a.\varepsilon$, the decompilation is the expression a.

Example 11

Code	Stack
$\texttt{CONST}(3); \texttt{SUB}; \texttt{ADD}$	$2.1.\varepsilon$
$\texttt{SUB}; \texttt{ADD}$	$3.2.1.\varepsilon$
$\texttt{ADD}$	$(2-3).1.\varepsilon$
ε	$1+(2-3).\varepsilon$

The decompilation is $1 + (2 - 3)$.

Decompilation by symbolic execution of the Modern SECD

Same idea: use a symbolic variant of the Modern SECD that operates over expressions rather than machine values.
Decompilation of machine values:

$$\mathcal{D}(N) = N \qquad\qquad \mathcal{D}(c[e]) = (\lambda a)[\mathcal{D}(e)] \text{ if } c = \mathcal{C}(a); \texttt{RETURN}$$

Decompilation of environments and stacks:

$$\mathcal{D}(v_1 \ldots v_n.\varepsilon) = \mathcal{D}(v_1) \ldots \mathcal{D}(v_n).\varepsilon$$
$$\mathcal{D}(\ldots v \ldots c.e \ldots) = \ldots \mathcal{D}(v) \ldots c.\mathcal{D}(e) \ldots$$

Decompilation of machine states: $\mathcal{D}(c, e, s) = a$ if the symbolic machine, started in state $(c, \mathcal{D}(e), \mathcal{D}(s))$, stops in state $(\varepsilon, e', a.\varepsilon)$.

Transitions for symbolic execution of the Modern SECD

Machine state before			Machine state after		
Code	Env	Stack	Code	Env	Stack
$\texttt{ACCESS}(n); c$	e	s	c	e	$e(n).s$
$\texttt{LET}; c$	e	$a.s$	c	$a.e$	s
$\texttt{ENDLET}; c$	$a.e$	$b.s$	c	e	$(\texttt{let } a \texttt{ in } b).s$
$\texttt{CLOSURE}(c'); c$	e	s	c	e	$\mathcal{D}(c)[e].s$
$\texttt{APPLY}; c$	e	$b.a.s$	c'	$v.e'$	$(a\ b).s$
$\texttt{RETURN}; c$	e	$a.c'.e'.s$	c'	e'	$a.s$

Simulation for the Modern SECD
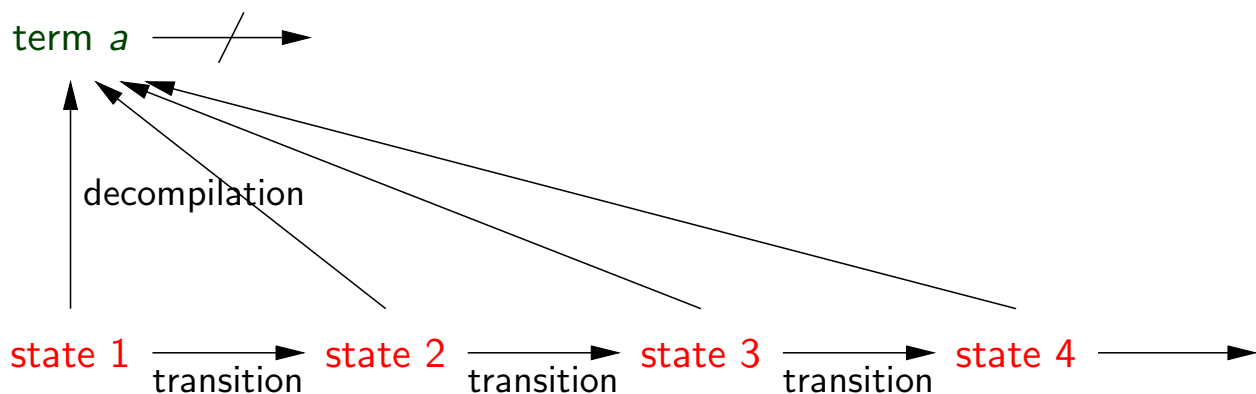
Lemma 12 (Simulation)

If the machine state (c, e, s) decompiles to the source term a, and if the machine makes a transition $(c, e, s) \rightarrow (c', e', s')$, then there exists a term a' such that

1 $a \xrightarrow{*} a'$

2 (c', e', s') *decompiles to a'.*

Note that we conclude $a \xrightarrow{*} a'$ instead of $a \rightarrow a'$ as in Krivine's machine. This is because many transitions of the Modern SECD correspond to no reductions: they move data around without changing the decompiled source term. Only the APPLY and LET transitions simulates one reduction step.

The stuttering problem

This makes it possible that the machine could "stutter": perform infinitely many transitions that correspond to zero reductions of the source term.

In this case, the machine could diverge even though the source term terminates (normally or on an error).

Simulation without stuttering

We can show that the stuttering problem does not occur by proving a stronger version of the simulation lemma:

Lemma 13 (Simulation without stuttering)

If the machine state (c, e, s) decompiles to the source term a, and if the machine makes a transition $(c, e, s) \to (c', e', s')$, then there exists a term a' such that

1. Either $a \to a'$, or $a = a'$ and $M(c', e', s') < M(c, e, s)$
2. (c', e', s') decompiles to a'.

Here, M is a measure associating nonnegative integers to machine states. A suitable definition of M is:

$$M(c, e, s) = \text{length}(c) + \sum_{c' \in s} \text{length}(c')$$

Total correctness for the Modern SECD

We can finish the proof by showing the Progress, Initial state and Final state lemmas with respect to CBV reduction semantics.
$\Rightarrow$ The Modern SECD is totally correct, after all.

But:

- The proofs are heavy.
- The definition of decompilation is complicated, hard to reason about, and hard to extend to more optimized compilation scheme.

Is there a better way?

Outline

1. Warm-up exercise: abstract machine for arithmetic expressions

2. Examples of abstract machines for functional languages
   - The Modern SECD
   - Tail call elimination
   - Krivine's machine
   - The ZAM

3. Correctness proofs for abstract machines
   - Total correctness for Krivine's machine
   - Partial correctness for the Modern SECD
   - Total correctness for the Modern SECD

4. Natural semantics for divergence
   - Definition and properties
   - Application to proofs of abstract machines

Reduction semantics versus natural semantics

Pros and cons of reduction semantics:

+ Accounts for all possible outcomes of evaluation:
  Termination: $a \xrightarrow{*} v$
  Divergence: $a \xrightarrow{*} a' \rightarrow \ldots$ (infinite sequence)
  Error: $a \xrightarrow{*} a' \nrightarrow$

− Compiler correctness proofs are painful.

Pros and cons of natural semantics:

− Describes only terminating evaluations $a \Rightarrow v$.
  If $a \nRightarrow v$ for all v, we do not know whether a diverges or causes an error.

+ Convenient for compiler correctness proofs

Idea: try to describe either divergence or errors using natural semantics.

Natural semantics for erroneous terms

Describing erroneous evaluations in natural semantics is easy but not very interesting.

$$x \Rightarrow \texttt{err} \qquad \frac{a \Rightarrow \texttt{err}}{a\ b \Rightarrow \texttt{err}} \qquad \frac{a \Rightarrow v \quad b \Rightarrow \texttt{err}}{a\ b \Rightarrow \texttt{err}}$$

$$\frac{a \Rightarrow v \quad b \Rightarrow v' \quad v \text{ is not a } \lambda}{a\ b \Rightarrow \texttt{err}}$$

Natural semantics for divergence

More challenging but more interesting is the description of divergence in natural semantics.

Idea: what are terms that diverge in reduction semantics?

They must be applications $a\ b$ — other terms do not reduce.

An infinite reduction sequence for $a\ b$ is necessarily of one of the following three forms:

1. $a\ b \to a_1\ b \to a_2\ b \to a_3\ b \to \ldots$
   i.e. a reduces infinitely.

2. $a\ b \xrightarrow{*} v\ b \to v\ b_1 \to v\ b_2 \to v\ b_3 \to \ldots$
   i.e. a terminates, but b reduces infinitely.

3. $a\ b \xrightarrow{*} (\lambda x.c)\ b \xrightarrow{*} (\lambda x.c)\ v \to c[x \leftarrow v] \to \ldots$
   i.e. a and b terminate, but the term after β-reduction reduces infinitely.

Natural semantics for divergence

Transcribing these three cases of divergence as inference rules in the style of natural semantics, we get the following rules for $a \Rightarrow \infty$ (read: "the term a diverges").

$$\frac{a \Rightarrow \infty}{a\ b \Rightarrow \infty} \qquad\qquad \frac{a \Rightarrow v \qquad b \Rightarrow \infty}{a\ b \Rightarrow \infty}$$

$$\frac{a \Rightarrow \lambda x.c \qquad b \Rightarrow v \qquad c[x \leftarrow v] \Rightarrow \infty}{a\ b \Rightarrow \infty}$$

To make sense, these rules must be interpreted coinductively.

Inductive and coinductive interpretations

A set of axioms and inference rules define not one but two logical predicates of interest:

- Inductive interpretation / smallest fixpoint:
  the predicate holds iff it is the conclusion of a finite derivation tree.

- Coinductive interpretation / greatest fixpoint:
  the predicate holds iff it is the conclusion of a finite or infinite derivation tree.

Example of coinductive interpretation

Consider the following inference rules for the predicate $\text{even}(n)$

$$\text{even}(0) \qquad\qquad \frac{\text{even}(n)}{\text{even}(n+2)}$$

Assume that n ranges over $\mathbb{N} \cup \{\infty\}$, with $\infty + 2 = \infty$.

With the inductive interpretation of the rules, the even predicate holds on the following numbers: 0, 2, 4, ..., but $\text{even}(\infty)$ does not hold.

With the coinductive interpretation, even holds on 0, 2, 4, ..., and also on ∞. This is because we have an infinite derivation tree $\mathcal{T}$ that concludes $\text{even}(\infty)$:

$$\mathcal{T} = \frac{\dfrac{\mathcal{T}}{}}{\text{even}(\infty)}$$

Example of diverging evaluation

The inductive interpretation of $a \Rightarrow \infty$ is always false: there are no axioms, hence no finite derivations.

The coinductive interpretation captures classic examples of divergence. Taking e.g. $\delta = \lambda x.\ x\ x$, we have the following infinite derivation:

$$\frac{\delta \Rightarrow \lambda x.\ x\ x \quad \delta \Rightarrow \delta \quad \dfrac{\delta \Rightarrow \lambda x.\ x\ x \quad \delta \Rightarrow \delta \quad \dfrac{\delta \Rightarrow \lambda x.\ x\ x \quad \delta \Rightarrow \delta \quad \dfrac{\vdots}{\delta\ \delta \Rightarrow \infty}}{\delta\ \delta \Rightarrow \infty}}{\delta\ \delta \Rightarrow \infty}}{\delta\ \delta \Rightarrow \infty}$$

Equivalence between $\Rightarrow \infty$ and infinite reductions

Theorem 14

If $a \Rightarrow \infty$, then a reduces infinitely.

Proof.

We show that for all n and a, if $a \Rightarrow \infty$, then there exists a reduction sequence of length n starting with a. The proof is by induction over n, then induction over a, then case analysis on the rule used to conclude $a \Rightarrow \infty$. (Exercise.) □

Equivalence between $\Rightarrow \infty$ and infinite reductions

Theorem 15

If a reduces infinitely, then $a \Rightarrow \infty$.

Proof.

The coinduction principle applied to the rules defining $\Rightarrow \infty$ tells us that:
If T is a set of terms such that for all $a \in T$,

- either $a = b\ c$ and $b \in T$;
- or $a = b\ c$ and $b \Rightarrow v$ and $c \in T$;
- or $a = b\ c$ and $b \Rightarrow \lambda x.d$ and $c \Rightarrow v$ and $d[x \leftarrow v] \in T$.

Then, for all $a \in T$, we have $a \Rightarrow \infty$.
The expected result follows from this coinduction theorem applied to the set $T = \{a \mid a \text{ reduces infinitely}\}$.

□

Divergence rules with environments and closures

We can follow the same approach for evaluations using environments and closures, obtaining the following rules for $e \vdash a \Rightarrow \infty$
(read: "in environment e, the term a diverges").

$$\frac{e \vdash a \Rightarrow \infty}{e \vdash a\ b \Rightarrow \infty} \qquad \frac{e \vdash a \Rightarrow v \quad e \vdash b \Rightarrow \infty}{e \vdash a\ b \Rightarrow \infty}$$

$$\frac{e \vdash a \Rightarrow (\lambda.c)[e'] \quad e \vdash b \Rightarrow v \quad v.e' \vdash c \Rightarrow \infty}{e \vdash a\ b \Rightarrow \infty}$$

(Again: coinductive interpretation.)

Back to the total correctness of the Modern SECD

We can now use the $e \vdash a \Rightarrow \infty$ predicate to obtain a simpler proof that the Modern SECD executes correctly terms that diverge:

Theorem 16

If $e \vdash a \Rightarrow \infty$, then for all k and s, the Modern SECD performs infinitely many transitions starting from the state

$$\begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix}$$

Proof principle

Lemma 17

Let X be a set of machine states such that

$$\forall S \in X, \ \exists S' \in X, \ S \overset{+}{\rightarrow} S'$$

Then, the machine, started in a state $S \in X$, performs infinitely many transitions.

Proof.

Assume the lemma is false and consider a minimal counterexample, that is, $S \in X \overset{*}{\rightarrow} S' \not\rightarrow$ and the number of transitions from S to S' is minimal among all such counterexamples.

By hypothesis over X and determinism of the machine, there exists a state S_1 such that $S \overset{+}{\rightarrow} S_1 \in X \overset{*}{\rightarrow} S' \not\rightarrow$. But then S_1 is a counterexample smaller than S. Contradiction. □

Application to the theorem

Consider

$$X = \left\{ \left(\begin{array}{c} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{array} \right) \ \middle| \ e \vdash a \Rightarrow \infty \right\}$$

It suffices to show $\forall S \in X, \ \exists S' \in X, \ S \overset{+}{\rightarrow} S'$ to establish the theorem.

The proof

Take $S \in X$, that is, $S = \begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix}$ with $e \vdash a \Rightarrow \infty$.

We show $\exists S' \in X, \; S \xrightarrow{+} S'$ by induction over a.

- First case: $a = a_1 \; a_2$ and $e \vdash a_1 \Rightarrow \infty$.
  $\mathcal{C}(a); k = \mathcal{C}(a_1); (\mathcal{C}(a_2); \texttt{APPLY}; k)$. The result follows by induction hypothesis

- Second case: $a = a_1 \; a_2$ and $e \vdash a_1 \Rightarrow v$ and $e \vdash a_2 \Rightarrow \infty$.

$$S = \begin{pmatrix} \mathcal{C}(a_1); \mathcal{C}(a_2); \texttt{APPLY}; k \\ \mathcal{C}(e) \\ s \end{pmatrix} \xrightarrow{+} \begin{pmatrix} \mathcal{C}(a_2); \texttt{APPLY}; k \\ \mathcal{C}(e) \\ \mathcal{C}(v).s \end{pmatrix} = S'$$

and we have $S' \in X$.

The proof

- Third case: $a = a_1 \; a_2$ and $e \vdash a_1 \Rightarrow (\lambda c)[e']$ and $e \vdash a_2 \Rightarrow v$ and $v.e' \vdash c \Rightarrow \infty$

$$S = \begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix} \xrightarrow{+} \begin{pmatrix} \mathcal{C}(a_2); \texttt{APPLY}; k \\ \mathcal{C}(e) \\ \mathcal{C}(\lambda c[e']).s \end{pmatrix}$$

$$\xrightarrow{+} \begin{pmatrix} \texttt{APPLY}; k \\ \mathcal{C}(e) \\ \mathcal{C}(v).\mathcal{C}(\lambda c[e']).s \end{pmatrix}$$

$$\rightarrow \begin{pmatrix} \mathcal{C}(c); \texttt{RETURN} \\ \mathcal{C}(v.e') \\ k.\mathcal{C}(e).s \end{pmatrix} = S'$$

and we have $S' \in X$, as expected.

Summary

Combining theorems 10 and 16, we obtain the following total correctness theorem for the Modern SECD:

Theorem 18

Let a be a closed program. Starting the Modern SECD in state
$(\mathcal{C}(a), \varepsilon, \varepsilon)$,

- *If $\varepsilon \vdash a \Rightarrow v$, the machine executes a finite number of transitions and stops on the final state $(\varepsilon, \varepsilon, \mathcal{C}(v).\varepsilon)$.*

- *If $\varepsilon \vdash a \Rightarrow \infty$, the machine executes an infinite number of transitions.*