On the Equivalence between Small-Step and Big-Step Abstract Machines: A Simple Application of Lightweight Fusion

Olivier Danvy
Kevin Millikin

BRICS Report Series RS-07-16

ISSN 0909-0878 November 2007

On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion

Olivier Danvy and Kevin Millikin
BRICS
Department of Computer Science
University of Aarhus[*]

November 2007

Abstract

We show how Ohori and Sasano's recent lightweight fusion by fixed-point promotion provides a simple way to prove the equivalence of the two standard styles of specification of abstract machines: (1) in small-step form, as a state-transition function together with a 'driver loop,' i.e., a function implementing the iteration of this transition function; and (2) in big-step form, as a tail-recursive function that directly maps a given configuration to a final state, if any. The equivalence hinges on our observation that for abstract machines, fusing a small-step specification yields a big-step specification. We illustrate this observation here with a recognizer for Dyck words, the CEK machine, and Krivine's machine with call/cc.

The need for such a simple proof is motivated by our current work on small-step abstract machines as obtained by refocusing a function implementing a reduction semantics (a syntactic correspondence), and big-step abstract machines as obtained by CPS-transforming and then defunctionalizing a function implementing a big-step semantics (a functional correspondence).

To appear in Information Processing Letters (extended version).

[*]IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: <danvy@brics.dk>, <kmillikin@brics.dk>

Contents

1 Introduction

Abstract machines come in two varieties:

A *small-step* abstract machine is defined as a state-transition function together with a 'driver loop,' i.e., a function implementing the iteration of this transition function towards a final state, if any.

A *big-step* abstract machine is defined as a collection of mutually tail-recursive transition functions mapping a given configuration to a final state, if any.

Small-step abstract machines are common in algorithmics (e.g., pushdown automata [17, Chapter 7]) and in semantics (e.g., the CEK machine [10]), and so are big-step abstract machines (e.g., finite automata [2, Figure 3.2.2, page 116] and the Krivine machine [13]). Their equivalence, however, is often given only intuitively.

In this article, we formally show that fusing the state-transition function and the driver loop of a small-step abstract machine yields the transition function of a big-step abstract machine. To this end, we use a particularly simple form of fusion: Ohori and Sasano's lightweight fusion by fixed-point promotion [18]. Ohori and Sasano presented a derivation method and proved its full correctness. The equivalence of the two abstract machines therefore follows as a corollary. We illustrate this equivalence with three examples: a recognizer for Dyck words (Section 2), the CEK machine (Section 3), and Krivine's machine with call/cc (Section 5). We use a pure and strict functional meta-language in the syntax of Standard ML and with no other effect than divergence. Ohori and Sasano's work therefore directly applies.

2 Example #1: a recognizer for Dyck words

Dyck words are well-balanced words of left and right parentheses, which we represent in ML as follows:

```
datatype parenthesis = L | R

type word = parenthesis list
```

For example, the list `[L, L, R, L, R, R]` represents a Dyck word whereas the list `[R, L]` does not.

Dyck words are classically recognized with an abstract machine implementing a push-down automaton. This state-transition system operates iteratively over a given list and a counter reflecting the number of open parentheses seen in the list so far:

```
datatype nat = ZERO | SUCC of nat
```

The machine starts with a given word and a zero counter. At each iteration, one of the following transitions takes place:

- if the list of parentheses is empty and the counter is zero, a final, accepting state is reached;
- if the list of parentheses is empty and the counter is positive, a final, non-accepting state is reached;
- if the first parenthesis is a left one, the tail of the list is taken and the counter is incremented;
- if the first parenthesis is a right one and if the counter is zero, a final, non-accepting state is reached;
- if the first parenthesis is a right one and if the counter is positive, the tail of the list is taken and the counter is decremented.

2.1 A small-step abstract machine

The following pure and total ML program implements the transition system above as a small-step abstract machine.[1] The machine has two states, a final one (a halting state) and an intermediate one (a configuration), and we define them with a data type. We define the state-transition function as a total function from intermediate states to states, and a driver loop as a function iterating the transition function until a final state is reached:

```
datatype         state = FINAL of halting_state | INTER of configuration
withtype halting_state = bool
     and configuration = word * nat

(*  move : configuration -> state  *)
fun move (    nil, ZERO  ) = FINAL true
  | move (    nil, SUCC c) = FINAL false
  | move (L :: ps,      c) = INTER (ps, SUCC c)
  | move (R :: ps, ZERO  ) = FINAL false
  | move (R :: ps, SUCC c) = INTER (ps,      c)

(*  drive : state -> halting_state  *)
fun drive (FINAL a) = a
  | drive (INTER g) = drive (move g)
```

A word is recognized by supplying the driver loop with an initial state:

```
(*  recognize : word -> halting_state  *)
fun recognize ps = drive (INTER (ps, ZERO))
```

2.2 A big-step abstract machine

The following pure and total ML program implements the transition system above as a big-step abstract machine.[2] We define a tail-recursive function mapping a configuration to a halting state:

```
(*  iterate : word * nat -> bool  *)
fun iterate (    nil, ZERO  ) = true
  | iterate (    nil, SUCC c) = false
  | iterate (L :: ps,      c) = iterate (ps, SUCC c)
  | iterate (R :: ps, ZERO  ) = false
  | iterate (R :: ps, SUCC c) = iterate (ps,      c)

(*  recognize : word -> bool  *)
fun recognize ps = iterate (ps, ZERO)
```

2.3 From small steps to big steps

Following Ohori and Sasano's steps, we consider the composition of `drive` and `move`:

```
fn g => drive (move g)
```

Step 1: Inline the definition of `move` in the composition.

```
fn g => drive (case g of (    nil, ZERO  ) => FINAL true
                       | (    nil, SUCC c) => FINAL false
                       | (L :: ps,      c) => INTER (ps, SUCC c)
                       | (R :: ps, ZERO  ) => FINAL false
                       | (R :: ps, SUCC c) => INTER (ps,      c))
```

[1]We used to refer to small-step abstract machines as 'pre-abstract machines' in our previous work [4, 5, 8].
[2]We used to refer to big-step abstract machines as 'abstract machines' in our previous work [1, 4, 5, 8].

Step 2: Distribute `drive` in the conditional branches.

```
fn g => (case g of (    nil, ZERO  ) => drive (FINAL true        )
                 | (    nil, SUCC c) => drive (FINAL false       )
                 | (L :: ps,      c) => drive (INTER (ps, SUCC c))
                 | (R :: ps, ZERO  ) => drive (FINAL false       )
                 | (R :: ps, SUCC c) => drive (INTER (ps,      c)))
```

Step 3: Simplify by inlining applications of `drive` to known arguments.

```
fn g => (case g of (    nil, ZERO  ) => true
                 | (    nil, SUCC c) => false
                 | (L :: ps,      c) => drive (move (ps, SUCC c))
                 | (R :: ps, ZERO  ) => false
                 | (R :: ps, SUCC c) => drive (move (ps,      c)))
```

Step 4: Use the result of Step 3 to define a new recursive function `drive_move` equal to `drive o move`.

```
fun drive_move (    nil, ZERO  ) = true
  | drive_move (    nil, SUCC c) = false
  | drive_move (L :: ps,      c) = drive_move (ps, SUCC c)
  | drive_move (R :: ps, ZERO  ) = false
  | drive_move (R :: ps, SUCC c) = drive_move (ps,      c)

fun recognize ps = drive_move (ps, ZERO)
```

The fused version coincides with the big-step abstract machine.

Ohori and Sasano have proved that this fixed-point promotion is correct if `drive` is strict, which it is here. (This kind of condition occurs frequently for fixed points of composite functions [22, Exercise 10.3, page 165]; see also Launchbury and Sheard's original work on warm fusion [15].) Their proof is based on a denotational semantics and shows that the denotation before and after fixed-point promotion are equal. The small-step abstract machine and the big-step abstract machine are therefore equivalent.

3 Example #2: the CEK machine

The CEK machine implements a weak-head normalization function for λ-terms using a left-to-right applicative-order reduction strategy [9–11]. Below, we represent λ-terms with de Bruijn indices (i.e., variables are represented with their lexical offset rather than with their names):

```
datatype term = VAR of int | LAM of term | APP of term * term
```

Terms are evaluated into a representation of their weak-head normal form, if there is one. The normal form takes the form of a closure, i.e., a construct pairing a term and an environment [14]:

```
datatype       value = CLO of term * environment
withtype environment = value list
```

The reduction contexts are the usual ones for left-to-right applicative order:

```
datatype reduction_context = RC0
                           | RC1 of reduction_context * term * environment
                           | RC2 of value * reduction_context
```

3.1 A small-step abstract machine

The CEK machine has two parameterized configurations: one with a term, an environment, and a reduction context where the term is dispatched upon; and one with a reduction context and a value where the reduction context is dispatched upon:

```
datatype configuration = EVAL of term * environment * reduction_context
                       | APPLY of reduction_context * value
```

The machine has two states: a final one and an intermediate one. A state-transition function `move` maps an intermediate state to a state, and as in Section 2.1, a driver loop maps a state to a halting state:

```
datatype         state = FINAL of halting_state | INTER of configuration
withtype halting_state = value option

(*  move : configuration -> state  *)
fun move (EVAL (VAR i,        e, c)) = if 0 <= i andalso i < (List.length e)
                                       then INTER (APPLY (c, List.nth (e, i)))
                                       else FINAL NONE
  | move (EVAL (LAM t,        e, c)) = INTER (APPLY (c, CLO (t, e)))
  | move (EVAL (APP (t0, t1), e, c)) = INTER (EVAL (t0, e, RC1 (c, t1, e)))
  | move (APPLY (RC0,               v)) = FINAL (SOME v)
  | move (APPLY (RC1 (c, t1, e),    v)) = INTER (EVAL (t1, e, RC2 (v, c)))
  | move (APPLY (RC2 (CLO (t, e), c), v)) = INTER (EVAL (t, v :: e, c))

(*  drive : state -> halting_state  *)
fun drive (FINAL a) = a
  | drive (INTER g) = drive (move g)
```

A closed term is evaluated by supplying the driver loop with an initial state:

```
(*  evaluate : term -> halting_state  *)
fun evaluate t = drive (INTER (EVAL (t, nil, RC0)))
```

3.2 A big-step abstract machine

The following ML program implements the usual big-step version of the CEK abstract machine [10], as obtained by CPS transformation and defunctionalization of a function implementing a big-step operational semantics [1, 11, 20]:

```
(*  eval : term * environment * reduction_context -> value option  *)
fun eval (VAR i,        e, c) = if 0 <= i andalso i < (List.length e)
                                then apply (c, List.nth (e, i))
                                else NONE
  | eval (LAM t,        e, c) = apply (c, CLO (t, e))
  | eval (APP (t0, t1), e, c) = eval (t0, e, RC1 (c, t1, e))
(*  apply : reduction_context * value -> value option  *)
and apply (RC0,                 v) = SOME v
  | apply (RC1 (c, t1, e),      v) = eval (t1, e, RC2 (v, c))
  | apply (RC2 (CLO (t, e), c), v) = eval (t , v :: e, c)

(*  evaluate : term -> value option  *)
fun evaluate t = eval (t, nil, RC0)
```

3.3 From small steps to big steps

Following Ohori and Sasano's steps, we consider the composition of `drive` and `move`:

```
fn g => drive (move g)
```

Step 1: Inline the definition of `move` in the composition.

```
fn g => drive (case g
                of (EVAL (VAR i,        e, c)) => if 0 <= i andalso i < (List.length e)
                                                  then INTER (APPLY (c, List.nth (e, i)))
                                                  else FINAL NONE
                 | (EVAL (LAM t,        e, c)) => INTER (APPLY (c, CLO (t, e)))
                 | (EVAL (APP (t0, t1), e, c)) => INTER (EVAL (t0, e, RC1 (c, t1, e)))
                 | (APPLY (RC0,                v)) => FINAL (SOME v)
                 | (APPLY (RC1 (c, t1, e),     v)) => INTER (EVAL (t1, e, RC2 (v, c)))
                 | (APPLY (RC2 (CLO (t, e), c), v)) => INTER (EVAL (t, v :: e, c)))
```

Step 2: Distribute `drive` in the conditional branches.

```
fn g => case g
         of (EVAL (VAR i,        e, c)) => if 0 <= i andalso i < (List.length e)
                                           then drive (INTER (APPLY (c, List.nth (e, i))))
                                           else drive (FINAL NONE)
          | (EVAL (LAM t,        e, c)) => drive (INTER (APPLY (c, CLO (t, e))))
          | (EVAL (APP (t0, t1), e, c)) => drive (INTER (EVAL (t0, e, RC1 (c, t1, e))))
          | (APPLY (RC0,                v)) => drive (FINAL (SOME v))
          | (APPLY (RC1 (c, t1, e),     v)) => drive (INTER (EVAL (t1, e, RC2 (v, c))))
          | (APPLY (RC2 (CLO (t, e), c), v)) => drive (INTER (EVAL (t, v :: e, c))))
```

Step 3: Simplify by inlining applications of `drive` to known arguments.

```
fn g => case g
         of (EVAL (VAR i,        e, c)) => if 0 <= i andalso i < (List.length e)
                                           then drive (move (APPLY (c, List.nth (e, i))))
                                           else NONE
          | (EVAL (LAM t,        e, c)) => drive (move (APPLY (c, CLO (t, e))))
          | (EVAL (APP (t0, t1), e, c)) => drive (move (EVAL (t0, e, RC1 (c, t1, e))))
          | (APPLY (RC0,                v)) => SOME v
          | (APPLY (RC1 (c, t1, e),     v)) => drive (move (EVAL (t1, e, RC2 (v, c))))
          | (APPLY (RC2 (CLO (t, e), c), v)) => drive (move (EVAL (t, v :: e, c))))
```

Step 4: Use the result of Step 3 to define a new recursive function `drive_move` equal to `drive o move`.

```
fun drive_move (EVAL (VAR i,        e, c)) = if 0 <= i andalso i < (List.length e)
                                             then drive_move (APPLY (c, List.nth (e, i)))
                                             else NONE
  | drive_move (EVAL (LAM t,        e, c)) = drive_move (APPLY (c, CLO (t, e)))
  | drive_move (EVAL (APP (t0, t1), e, c)) = drive_move (EVAL (t0, e, RC1 (c, t1, e)))
  | drive_move (APPLY (RC0,                v)) = SOME v
  | drive_move (APPLY (RC1 (c, t1, e),     v)) = drive_move (EVAL (t1, e, RC2 (v, c)))
  | drive_move (APPLY (RC2 (CLO (t, e), c), v)) = drive_move (EVAL (t, v :: e, c))

  fun evaluate t = drive_move (EVAL (t, nil, RC0))
```

Using the type isomorphism between $(A + B) \to C$ and $(A \to C) \times (B \to C)$ and disentangling `drive_move` into two mutually recursive functions, which is another case of fusion (a trivial one), yields the big-step version of the CEK machine displayed in Section 3.2. The small-step abstract machine and the big-step abstract machine are therefore equivalent.

4 From reduction semantics to abstract machine

The present work is part of our current study of a syntactic correspondence between reduction semantics [9, 10] and abstract machines, using the refocusing technique [8]. The idea is as follows.

In a reduction semantics, a one-step reduction function is defined as the composition of

1. a total 'decomposition' function from non-value terms to potential redexes[3] and reduction contexts;
2. a partial 'contraction' function[4] mapping actual redexes and their reduction context to a contractum and a reduction context (possibly a different one, to account for control effects [5, 9]); and
3. a total 'plug' function filling the reduction context with the contractum.

Graphically:

$$\begin{tikzcd}
\circ \arrow[dr, "\text{decompose}"] & & & \circ \\
& \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] &
\end{tikzcd}$$

Evaluation is defined as iterated reduction:

$$\begin{tikzcd}
\circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & & \circ \\
& \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] & & \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] & & \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] &
\end{tikzcd}$$

Danvy and Nielsen [8] pointed out that the composition of the two total functions decompose and plug could be fused into a 'refocus' function that continues the decomposition from the current reduction context to the next one, if there is one, and they presented an algorithm to construct an optimal such refocus function:

$$\begin{tikzcd}
\circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & & \circ & \\
& \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] \arrow[rr, dashed, "\text{refocus}"'] & & \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] \arrow[rr, dashed, "\text{refocus}"'] & & \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] \arrow[rr, dashed, dash] & & {}
\end{tikzcd}$$

A refocused evaluation function takes the form of a small-step abstract machine: refocus is a (total) state-transition function that is iterated by a (strict) function implementing a driver loop. Lightweight fusion yields the corresponding big-step abstract machine, and in our experience, compressing corridor transitions in this big-step abstract machine often leads to an abstract machine that was independently known [4, 5, 8], as illustrated next.

5 Example #3: Krivine's machine with call/cc

Let us specify the $\lambda\widehat{\rho}\mathcal{K}$-calculus, i.e., Curien's original calculus of closures [4, 7] with generalized β-reduction and call/cc, as recently defined by Biernacka and Danvy [5, Section 4].

We start from the usual λ-terms with de Bruijn indices and the control operator call/cc [6]:

```
datatype term = VAR of int | LAM of term | APP of term * term | CCC of term
```

[3]Potential redexes are partitioned into actual redexes, which can be contracted, and stuck redexes, which cannot.
[4]The contraction function is partial because stuck redexes are not contracted.

A calculus of closures embodies Landin's original idea [14] that the substitution generally associated with β-reduction should be delayed, and that instead, each term should be associated with an environment reflecting what should have been substituted in this term. As specified in the ML data type below, a closure can be ground, i.e., it can simply pair a term and an environment; it can be the application of one closure to another; it can be the application of call/cc to a closure; or it can be a captured context as provided by call/cc. As for the reduction contexts, they are inductively defined as usual for normal order: `RC0` corresponds the empty context, `RC1` to the context of an application, and `RC2` to the context of call/cc.

```
datatype      closure = CLO_GND of term * environment
                      | CLO_APP of closure * closure
                      | CLO_CCC of closure
                      | CLO_CTX of reduction_context
and reduction_context = RC0
                      | RC1 of reduction_context * closure
                      | RC2 of reduction_context
withtype environment = closure list
```

We define potential redexes and the corresponding contraction function in the separate structure below. There are five potential redexes:

- The `ACCESS` redex concerns a de Bruijn index and an environment. It is stuck if the index is out of the domain of the environment. Otherwise, this actual redex is contracted in a given context by fetching the closure held in the given environment at the given index.
- The `BETA` redex concerns a closure applied in context. It is contracted in a non-empty context that corresponds to an application (and its contaction is therefore context-sensitive). If the closure corresponds to a curried λ-abstraction, the matching number of actual parameters is popped from the context (which implements the so-called 'generalized' β-reduction). If the closure is a captured context, this context replaces the current one.
- The `PROP_APP` and `PROP_CCC` redexes propagate the current environment downwards in the abstract syntax tree.
- The `CWCC` redex corresponds to applying call/cc to a closure. It is contracted by capturing the current context and applying this closure to it.

```
structure Redexes
= struct
    datatype potential_redex = ACCESS of int * environment
                             | BETA of closure
                             | PROP_APP of term * term * environment
                             | PROP_CCC of term * environment
                             | CWCC of closure

    (*  contract : potential_redex * reduction_context
                -> (closure * reduction_context) option  *)
    fun contract (ACCESS (i, e), rc)
        = if 0 <= i andalso i < (List.length e)
          then SOME (List.nth (e, i), rc)
          else NONE
      | contract (BETA (CLO_GND (LAM t, e)), RC1 (rc, c))
        = let fun traverse (LAM t, e, RC1 (rc, c))
                  = traverse (t, c :: e, rc)
                | traverse (t, e, rc)
                  = SOME (CLO_GND (t, e), rc)
          in traverse (t, c :: e, rc)
          end
        | contract (BETA (CLO_CTX rc'), RC1 (rc, c))
          = SOME (c, rc')
        | contract (PROP_APP (t0, t1, e), rc)
          = SOME (CLO_APP (CLO_GND (t0, e), CLO_GND (t1, e)), rc)
        | contract (PROP_CCC (t, e), rc)
          = SOME (CLO_CCC (CLO_GND (t, e)), rc)
        | contract (CWCC c, rc)
          = SOME (CLO_APP (c, CLO_CTX rc), rc)
        | contract _
          = NONE
    end
```

The function refocus is given a closure and a reduction context and navigates through the shortest path towards the next potential redex and its reduction context, if there is one [8]:

```
  datatype val_or_dec = VAL of closure
                      | DEC of Redexes.potential_redex * reduction_context

  (*  refocus : configuration -> val_or_dec  *)
  fun refocus (CLO_GND (VAR i, e),        rc         ) = DEC (Redexes.ACCESS (i, e), rc)
    | refocus (c as CLO_GND (LAM t, e),   RC0        ) = VAL c
    | refocus (c as CLO_GND (LAM t, e),   rc as RC1 _) = DEC (Redexes.BETA c, rc)
    | refocus (c as CLO_GND (LAM t, e),   RC2 rc     ) = DEC (Redexes.CWCC c, rc)
    | refocus (CLO_GND (APP (t0, t1), e), rc         ) = DEC (Redexes.PROP_APP (t0, t1, e), rc)
    | refocus (CLO_GND (CCC t, e),        rc         ) = DEC (Redexes.PROP_CCC (t, e), rc)
    | refocus (CLO_APP (c0, c1),          rc         ) = refocus (c0, RC1 (rc, c1))
    | refocus (CLO_CCC c,                 rc         ) = refocus (c , RC2 rc)
    | refocus (c as CLO_CTX rc',          RC0        ) = VAL c
    | refocus (c as CLO_CTX rc',          rc as RC1 _) = DEC (Redexes.BETA c, rc)
    | refocus (c as CLO_CTX rc',          RC2 rc     ) = DEC (Redexes.CWCC c, rc)
```

As usual, the iterate function acts as a trampoline [12] and repeatedly calls refocus until a final state is reached, if any:

```
  (*  iterate : val_or_dec -> closure option  *)
  fun iterate (VAL c)
      = SOME c
    | iterate (DEC (pr, rc))
      = (case Redexes.contract (pr, rc)
           of NONE
              => NONE
            | SOME (c, rc)
              => iterate (refocus (c, rc)))
```

The evaluation of a given closed term is initiated by manufacturing an initial state (the corresponding ground closure with an empty environment and the empty context) and launching the iteration:

```
  (*  evaluate : term -> closure option  *)
  fun evaluate t = iterate (refocus (CLO_GND (t, nil), RC0))
```

In this small-step abstract machine, the state is val_or_dec, the transition function is refocus and the driver loop is iterate.

As in Sections 2.3 and 3.3, lightweight fusion of iterate o refocus yields a function implementing a big-step abstract machine. Compressing corridor transitions in this abstract machine, disentangling it into two mutually recursive transition functions, and unfolding the data type of closures mechanically yields the latest version of Krivine's machine with call/cc [5, Section 4]—a machine that was conceived independently of any calculus [13, Section 3].

6 Conclusion

We have reported the simple observation that applying Ohori and Sasano's lightweight fusion to a small-step abstract machine yields a big-step abstract machine. We have illustrated it through three examples: a recognizer for Dyck words, the CEK machine, and Krivine's machine with call/cc. The usefulness of this observation stems from it being systematic and from lightweight fusion being correct: there is thus no need anymore to prove the equivalence of small-step and big-step abstract machines on a case-by-case basis.

The present work is part of an investigation of a syntactic correspondence between a reduction semantics (i.e., a calculus together with a reduction strategy) specifying a one-step reduction function and an abstract machine implementing an evaluation function [3–5, 8]. The relation between reduction strategies (for calculi) and evaluation orders (for abstract machines) was made by Plotkin in his seminal article "Call-by-name, Call-by-value and the λ-calculus" [19]: normal order corresponds to call by name, and applicative order to call by value. Today this relation is taken for granted: for example, Krivine spontaneously characterized his machine as a "call-by-name" one [13]. Our syntactic correspondence between reduction semantics and abstract machines not only mechanizes Plotkin's relation between reduction strategies and evaluation orders; it also connects architectural designs in an abstract machine with properties of the corresponding calculus. For example, Krivine's machine, with or without call/cc, is written in the so-called 'push/enter' style [16],[5] due to the fact that when a (curried) λ-abstraction is applied, its arguments are available on the control stack, i.e., in the context. As shown by the syntactic correspondence, this implementation design is foreshadowed by the generalized β-reduction in the corresponding calculus. The same goes for right-to-left call by value [4, Section 5.2], as in the ZINC abstract machine for Caml, and for call by need, as in the Three-Instruction Machine.

Finally, the present work suggests a useful extension to Ohori and Sasano's lightweight fusion by fixed-point promotion: to work for mutually recursive functions instead of only for single recursive functions. Indeed big-step abstract machines are often defined with transition functions that are mutually recursive.

Acknowledgments: Thanks are due to Małgorzata Biernacka, Kristian Støvring, and the anonymous reviewers of IPL for comments. This work is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* World Student Series. Addison-Wesley, Reading, Massachusetts, 1986.

[3] Małgorzata Biernacka and Dariusz Biernacki. Formalizing constructions of abstract machines for functional languages in Coq. In Jürgen Giesl, editor, *Preliminary proceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming (WRS'07)*, Paris, France, June 2007.

[4] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 2006. To appear. Available as the research report BRICS RS-06-3.

[5] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007. Extended version available as the research report BRICS RS-06-18.

[5] In contrast, the CEK machine is written in the so-called 'eval/apply' style (see Section 3.1).

[6] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[7] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[8] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[9] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987.

[10] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `<http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html>`, 1989-2003.

[11] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[12] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 34, No. 9, pages 18–27, Paris, France, September 1999. ACM Press.

[13] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[14] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[15] John Launchbury and Tim Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 314–323, La Jolla, California, June 1995. ACM Press.

[16] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In Kathleen Fisher, editor, *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, SIGPLAN Notices, Vol. 39, No. 9, pages 4–15, Snowbird, Utah, September 2004. ACM Press.

[17] John C. Martin. *Introduction to Languages and the Theory of Computation*. Programming Language Series. McGraw-Hill International Editions, second edition, 1997.

[18] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In Matthias Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 42, No. 1, pages 143–154, New York, NY, USA, January 2007. ACM Press.

[19] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[20] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [21].

[21] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[22] Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundation of Computing Series. The MIT Press, 1993.