BRICS

Basic Research in Computer Science

Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language

Part I:

Denotational Semantics, Natural Semantics, and Abstract Machines

Olivier Danvy

BRICS Report Series RS-08-7

ISSN 0909-0878 July 2008

Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part I: Denotational Semantics, Natural Semantics, and Abstract Machines

Olivier Danvy

Department of Computer Science
University of Aarhus *
danvy@brics.dk

Abstract

We derive two big-step abstract machines, a natural semantics, and the valuation function of a denotational semantics based on the small-step abstract machine for Core Scheme presented by Clinger at PLDI'98. Starting from a functional implementation of this small-step abstract machine, (1) we fuse its transition function with its driver loop, obtaining the functional implementation of a big-step abstract machine; (2) we adjust this big-step abstract machine so that it is in defunctionalized form, obtaining the functional implementation of a second big-step abstract machine; (3) we refunctionalize this adjusted abstract machine, obtaining the functional implementation of a natural semantics in continuation style; and (4) we closure-unconvert this natural semantics, obtaining a compositional continuation-passing evaluation function which we identify as the functional implementation of a denotational semantics in continuation style. We then compare this valuation function with that of Clinger's original denotational semantics of Scheme.

1. Introduction

Motivation: Somewhat facetiously, in an earlier work [7], Biernacka and the author concluded:

> Call/cc was introduced in Scheme [11] as a Church encoding of Reynolds's escape operator [48]. A typed version of it is available in Standard ML of New Jersey [29] and Griffin has identified its logical content [28]. It is endowed with a variety of specifications: a CPS transformation [18], a CPS interpreter [30, 48], a denotational semantics [33], a computational monad [55], a big-step operational semantics [29], the CEK machine [26], calculi in the form of reduction semantics [25], and a number of implementation techniques [12, 15, 31]—not to mention its call-by-name variant in the archival version of Krivine's machine [34].
>
> Question: How do we know that all the artifacts[1] in this semantic jungle define the same call/cc?

In some sense, the same can be said of the Scheme programming language today: it is independently specified with a denotational semantics [46], a structural operational semantics [45] and a reduction semantics [37], and its tail-recursion property is accounted for with a small-step abstract machine [14]. Which one specifies Scheme? The newest one? Or does one supersede a previous one of the same kind, just as the revised^{n+1} report supersedes the revised^{n} report [13, 33, 46, 50, 52, 53]? Or perhaps each semantics specifies one particular facet, however large that facet may be, of Scheme, irrespective of others? In that case, is the reference semantics the most complete one at any point of time?

Background: It is the author's thesis[2] that functional representations of small-step operational semantics (i.e., structured operational semantics), reduction semantics (i.e., small-step operational semantics with an explicit representation of the reduction context), small-step abstract machines, big-step abstract machines, big-step operational semantics (i.e., natural semantics), and denotational semantics are inter-derivable using elementary program transformations such as CPS transformation [18, 44, 47, 51], defunctionalization [21, 47], fixed-point fusion [19, 43], and refocusing [22].

* IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
http://www.brics.dk/~danvy

2008 Workshop on Scheme and Functional Programming

[1] "Artifact" means "man-made construct."

[2] A real one, actually [17], together with several others [1, 5, 9, 32, 38, 39, 41, 42], as a matter of fact.

This work: As a proof of concept, we derive the functional representation of the denotational semantics corresponding to Clinger's abstract machine as presented at PLDI'98 to specify the meaning of proper tail recursion [14]. We then compare it with Clinger's denotational semantics in the R^{3}RS [46]. Each of these semantics is significant: the denotational one was profoundly influential in that it revealed formal semantics in action to a whole generation of computer scientists, and the operational one was instrumental to substantiate the precise meaning of what it means for an implementation to be properly tail-recursive. They are also unique because of their use of permutation/unpermutation functions to account for the undetermined sequencing order of sub-expressions in an application.

Our starting point is that, as pioneered by Reynolds in "Definitional Interpreters" [47] and pursued by the author and his students [2–4, 6, 16], closure-converting and defunctionalizing a continuation-passing evaluation function yields a big-step abstract machine. Here, we restate Clinger's machine to operate with big steps, we adjust it so that it is in defunctionalized form, and we refunctionalize and then closure-unconvert the functional implementation of this adjusted machine, obtaining an evaluation function which is compositional and in continuation-passing style.

Prerequisites: Naturally, we assume the reader to be aware of Clinger's denotational semantics of Scheme in the R^{3}RS [46] and of his abstract machine for Core Scheme as presented at PLDI'98 [14]. In addition, we also expect a basic knowledge of Standard ML [40], a pure subset of which we use as a functional meta-language,[3] and a passing familiarity with defunctionalization [21, 47] and its left inverse, refunctionalization [19] as well as with closure conversion [35] and its left inverse, closure unconversion. As for the technique of fusing the transition function of a small-step abstract machine with its driver loop, it is described in a recent note by Millikin and the author [20].

Overview: We first review the domain of discourse (Section 2): stores, environments, and the permutation/unpermutation functions that are idiosyncratic to Clinger's semantic specifications of Scheme. We then specify the syntax (Section 3) and the semantics (Section 4) of Core Scheme, and a garbage-collection rule (Section 5). Thus equipped, we present Clinger's small-step abstract machine (Section 6) and then its big-step counterpart (Section 7). We put this big-step counterpart in defunctionalized form (Section 8), we present its refunctionalized counterpart (Section 9), and we closure-unconvert it (Section 10). We then compare the resulting compositional evaluation function in continuation-passing style to the valuation function of Clinger's denotational semantics (Section 11) and then conclude (Section 12).

[3] In that we keep in mind that the full name of the Scheme workshop is "Scheme *and Functional Programming*." (The emphasis is ours.)

Pictorially:

$$\begin{tikzcd}
\begin{array}{c}\text{Clinger's small-step abstract machine}\\ \text{(Figures 1 and 2 in Section 6)}\end{array} \arrow[d, "\text{lightweight fusion}"] \\
\begin{array}{c}\text{big-step abstract machine}\\ \text{(Figures 3 and 4 in Section 7)}\end{array} \arrow[d, "\text{adjustment}"] \\
\begin{array}{c}\text{big-step abstract machine in defunctionalized form}\\ \text{(Figures 5 and 6 in Section 8)}\end{array} \arrow[d, shift left, "\text{refunctionalization}"] \\
\begin{array}{c}\text{natural semantics in continuation style}\\ \text{(Figures 7 and 8 in Section 9)}\end{array} \arrow[u, shift left, "\text{defunctionalization}"] \arrow[d, shift left, "\text{closure unconversion}"] \\
\begin{array}{c}\text{denotational semantics}\\ \text{(Figure 9 in Section 10)}\end{array} \arrow[u, shift left, "\text{closure conversion}"]
\end{tikzcd}$$

2. Domain of discourse

In the interest of brevity and abstractness, we only present ML signatures for the store (Section 2.1) and the environment (Section 2.2). We also explicitly treat Clinger's permutations and unpermutations (Section 2.3).

2.1 Store

A store is a mapping from locations to storable values. We specify it as a polymorphic abstract data type with the usual algebraic operators to allocate fresh locations and initialize them with given storable values, dereference a given location in a given store, and update a given store at a given location with a given storable value.

```
signature STO = sig
  type 'a sto
  type loc

  val empty : 'a sto

  val new  : 'a sto * 'a      −¿ loc      * 'a sto
  val news : 'a sto * 'a list −¿ loc list * 'a sto

  val fetch  : loc *      'a sto −¿ 'a option
  val update : loc * 'a * 'a sto −¿ 'a sto
end

structure Sto : STO = struct
  (* deliberately omitted *)
end
```

At the price of modularity, we could define the allocation operators to also be given an environment and a context so that the fresh location(s) they return can be more precisely characterized as not occurring in the environment and in the context. This less modular definition would let us implement Clinger's small-step abstract machine even more faithfully.

2.2 Environment

An environment is a mapping from identifiers to denotable values. We specify it as a polymorphic abstract data type with the usual algebraic operators to extend a given environment with new bindings and to look up identifiers in a given environment. We also need a predicate testing whether a given environment is empty.

```
signature ENV = sig
  type 'a env

  val empty   : 'a env
  val emptyp  : 'a env −¿ bool
  val extend  : ide      ∗ 'a      ∗ 'a env
                −¿ ' a env
  val extends : ide list ∗ 'a list ∗ 'a env
                −¿ ' a env
  val lookup  : ide ∗ 'a env −¿ ' a option
end

structure Env : ENV = struct
  (∗ deliberately omitted ∗)
end
```

In the present study, the denotable values are locations in a store.

2.3 Permutations and unpermutations

Both in his denotational semantics and his small-step abstract machine, Clinger non-deterministically uses a pair of permutation functions: one over the sub-terms in an application, and the inverse one over the resulting values (see Note #1 in Section 6.1). We implement this non-determinism by threading a stream of pairs of permutations through Clinger's semantic specifications. We materialize this stream with the following polymorphic abstract data type.

```
signature PERMUTATION = sig
  type 'a permutation = 'a ∗ 'a list
                        −¿ ' a ∗ 'a list
  type ('v, 't) permutations

  val new : ('v, 't) permutations
          −¿ (' v permutation ∗ 't permutation)
             ∗ ('v, 't) permutations
  val init : ('v, 't) permutations
end

structure Permutation : PERMUTATION = struct
  (∗ deliberately omitted ∗)
end
```

3. Syntax

The following module implements the internal syntax of Core Scheme [14, Figure 1].

```
structure Syn = struct
  datatype quotation = QBOOL of bool
                     | QNUMB of int
                     | QSYMB of ide
                     | QPAIR of Sto.loc ∗ Sto.loc
                  (∗ | QVECT of ... ∗)
                  (∗ | ... ∗)

  datatype term = QUOTATION of quotation
                | VAR of ide
                | LAM of ide list ∗ term
                | APP of term ∗ term list
                | CND of term ∗ term ∗ term
                | SET of ide ∗ term
end
```

4. Semantics

The following module implements the expressible values and evaluation contexts [14, Figure 4]. Like Clinger, we focus on Core Scheme and do not consider primitive procedures and first-class continuations here. (These are unproblematic to add [4, 7].)

```
structure Sem = struct
(∗datatype primop = ...∗)

  datatype value = QUOTED of Syn.quotation
                 | UNSPECIFIED
                 | UNDEFINED
              (∗ | PRIMOP of primop ∗)
              (∗ | ESCAPE of Sto.loc ∗ cont ∗)
                 | CLOSURE of Sto.loc ∗
                         (ide list ∗ Syn.term) ∗
                              Sto.loc Env.env
        and cont = HALT
                 | SELECT of Syn.term ∗
                             Syn.term ∗
                             Sto.loc Env.env ∗
                             cont
                 | ASSIGN of ide ∗
                             Sto.loc Env.env ∗
                             cont
                 | PUSH of Syn.term list ∗
                           value list ∗
                value Permutation.permutation ∗
                           Sto.loc Env.env ∗
                           cont
                 | CALL of value list ∗ cont
end
```

Note: Compared to the original [14, Figure 4], we fixed one typo in the declaration of the evaluation-context constructor for calls, which holds a list of semantic values, not a list of syntactic terms.

5. Garbage collection

The following module is used to implement the garbage-collection rule [14, Figure 5].

```
signature GC = sig
  val gc : Sem.value ∗
           Sto.loc Env.env ∗
           Sem.cont ∗
           Sem.value Sto.sto
           −¿ Sem.value Sto.sto
end

structure Gc : GC = struct
  (∗ deliberately omitted ∗)
end
```

```
structure M_small_step = struct
  val r_init = Env.empty

  datatype term_or_value = TERM of Syn.term | VALUE of Sem.value

  datatype halting_state = RESULT of Sem.value * Sem.value Sto.sto | STUCK of string

  datatype state = FINAL of halting_state | INTER of configuration
  withtype configuration = term_or_value * Sto.loc Env.env * Sem.cont * Sem.value Sto.sto
                           * (Sem.value, Syn.term) Permutation.permutations

  fun move ...
      = ...

  fun drive (FINAL answer)
      = answer
    | drive (INTER configuration)
      = drive (move configuration)

  fun evaluate t
      = drive (INTER (TERM t, r_init, Sem.HALT, Sto.empty, Permutation.init))
end
```

Figure 1. Clinger's small-step abstract machine without the GC rule, part 1/2: configurations, driver loop and initialization

6. Clinger's small-step abstract machine

We first present the machine without the GC rule (Section 6.1) and then with a GC rule (Section 6.2).

6.1 Without the GC rule

The abstract machine is displayed in Figures 1 and 2. It operates on a quintuple: a term or a value, an environment mapping variables to store locations, a context, a store mapping locations to values, and a stream of permutations. When the first component is a term, this term is dispatched upon. When the first component is a value, the third component (i.e., the context) is dispatched upon.

Note #1: A new pair of permutations is explicitly allocated every time an application is evaluated. The second one operates over terms and is used immediately over the sub-terms of the application. The first one operates over values and it will be subsequently used over the results of evaluating each of these sub-terms.

Clinger's formulation uses an infinite set of rules generated by a rule schema that is parameterized by a permutation and its inverse. We model his non-deterministic choice of permutation with an oracle that picks in the current stream of permutations.

Note #2: Compared to the original [14, Figure 5] and to Clinger's denotational semantics, this semantics embodies the official specification of Scheme about assignments:

> When evaluating an assignment, the expression is evaluated, and the resulting value is stored in the location to which the variable is bound. This variable must be bound either in some region enclosing the assignment or at the top level.

Accordingly, in Figure 2, the expression is evaluated, and then the variable is checked to be bound.

For the rest, Figures 1 and 2 display a scrupulously faithful implementation of Clinger's small-step abstract machine.

6.2 With a GC rule

The following implementation deterministically applies the GC rule every time a value is returned to the evaluation context, i.e., at every reduction step. Compared to Figure 1, only the driver loop is changed.

```
structure M_small_step_with_gc_rule = struct
  ...
  fun drive (FINAL answer)
      = answer
    | drive (INTER (TERM t, r, c, s, p))
      = drive (move (TERM t, r, c, s, p))
    | drive (INTER (VALUE v, r, c, s, p))
      = let val s' = Gc.gc (v, r, c, s)
        in drive (move (VALUE v, r, c, s', p))
            end
  ...
end
```

In contrast, Clinger's formulation of the GC rule is non-deterministic.

```
structure M_small_step = struct
  ...
  fun move (TERM (Syn.QUOTATION q), r, c, s, p)
      = INTER (VALUE (Sem.QUOTED q), r, c, s, p)
    | move (TERM (Syn.VAR i), r, c, s, p)
      = (case Env.lookup (i, r)
           of (SOME l)
              =¿ ( case Sto.fetch (l, s)
                    of (SOME v)
                       =¿ ( case v
                             of Sem.UNDEFINED
                                =¿ FINAL (STUCK "attempt to reference undefined variable")
                              | _
                                =¿ INTER (VALUE v, r, c, s, p))
                      | NONE
                       =¿ FINAL (STUCK "attempt to read an invalid location"))
            | NONE
              =¿ FINAL (STUCK "attempt to reference an undeclared variable"))
    | move (TERM (Syn.LAM (is, t)), r, c, s, p)
      = let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
        in INTER (VALUE (Sem.CLOSURE (l, (is, t), r)), r, c, s', p) end
    | move (TERM (Syn.CND (t0, t1, t2)), r, c, s, p)
      = INTER (TERM t0, r, Sem.SELECT (t1, t2, r, c), s, p)
    | move (TERM (Syn.SET (i, t)), r, c, s, p)
      = INTER (TERM t, r, Sem.ASSIGN (i, r, c), s, p)
    | move (TERM (Syn.APP (t0, ts)), r, c, s, p)
      = let val ((pi, rev_pi_inv), p') = Permutation.new p
            val (t0', ts') = rev_pi_inv (t0, ts)
        in INTER (TERM t0', r, Sem.PUSH (ts', nil, pi, r, c), s, p') end
    | move (VALUE v, r', Sem.HALT, s, p)
      = if Env.emptyp r'
        then FINAL (RESULT (v, s))
        else INTER (VALUE v, Env.empty, Sem.HALT, s, p)
    | move (VALUE (Sem.QUOTED (Syn.QBOOL false)), r', Sem.SELECT (t1, t2, r, c), s, p)
      = INTER (TERM t2, r, c, s, p)
    | move (VALUE _, r', Sem.SELECT (t1, t2, r, c), s, p)
      = INTER (TERM t1, r, c, s, p)
    | move (VALUE v, r', Sem.ASSIGN (i, r, c), s, p)
      = (case Env.lookup (i, r)
           of (SOME l)
              =¿ ( case Sto.fetch (l, s)
                    of (SOME v)
                       =¿ ( case v
                             of Sem.UNDEFINED
                                =¿ FINAL (STUCK "attempt to assign undefined variable")
                              | _
                                =¿ INTER (VALUE Sem.UNSPECIFIED, r, c, Sto.update (l, v, s), p))
                      | NONE
                       =¿ FINAL (STUCK "attempt to write an invalid location"))
            | NONE
              =¿ FINAL (STUCK "attempt to assign an undeclared variable"))
    | move (VALUE v0', r', Sem.PUSH (nil, vs', pi, r, c), s, p)
      = let val (v0, vs) = pi (v0', vs')
        in INTER (VALUE v0, r, Sem.CALL (vs, c), s, p) end
    | move (VALUE v0', r', Sem.PUSH (t1' :: ts', vs', pi, r, c), s, p)
      = INTER (TERM t1', r, Sem.PUSH (ts', v0' :: vs', pi, r, c), s, p)
    | move (VALUE (Sem.CLOSURE (l, (is, t), r)), r', Sem.CALL (vs, c), s, p)
      = let val (ls, s') = Sto.news (s, vs)
        in INTER (TERM t, Env.extends (is, ls, r), c, s', p) end
    | move (VALUE v, r', Sem.CALL (vs, c), s, p)
      = FINAL (STUCK "attempt to apply a non−procedure")
  ...
end
```

Figure 2. Clinger's small-step abstract machine without the GC rule, part 2/2: transition function

```
structure M_big_step = struct
  val r_init = Env.empty

  datatype term_or_value = TERM of Syn.term | VALUE of Sem.value

  datatype answer = RESULT of Sem.value * Sem.value Sto.sto | STUCK of string

  fun iterate ...
      = ...

  fun evaluate t
      = iterate (TERM t, r_init, Sem.HALT, Sto.empty, Permutation.init)
end
```

Figure 3. Big-step counterpart of Figure 1, part 1/2: configurations and initialization

7. Big-step version of Clinger's abstract machine

We first present the big-step version of the machine without any GC rule (Section 7.1) and then with a GC rule (Section 7.2).

7.1 Without the GC rule

In Figure 1, the 'drive' function iteratively calls the 'move' function until a final answer is obtained, if any. As pointed out by Millikin and the author [20], such small-step abstract machines are candidates for lightweight fusion by fixed-point promotion [43]: the composition of 'drive' and 'move' can be fused into an 'iterate' function where the outer recursive call to 'drive' has been distributed to all the return points in the definition of 'move.' The result is the big-step abstract machine displayed in Figures 3 and 4. Since Ohori and Sasano's fixed-point promotion is fully correct, this big-step abstract machine is also correct, by construction.

7.2 With a GC rule

The following implementation deterministically applies the GC rule every time a function is about to be applied. Compared to Figure 4, only one clause is changed.

```
structure M_big_step_with_gc_rule = struct
  (* ... *)
    | iterate (VALUE v0',
               r',
               Sem.PUSH (nil, vs', pi, r, c),
               s,
               p)
      = let val (v0, vs) = pi (v0', vs')
            val s' = Gc.gc (v0,
                            r,
                            Sem.CALL (vs, c),
                            s)
        in iterate (VALUE v0,
                    r,
                    Sem.CALL (vs, c),
                    s',
                    p) end
  (* ... *)
end
```

```
structure M_big_step = struct
  ...
  fun iterate (TERM (Syn.QUOTATION q), r, c, s, p)
      = iterate (VALUE (Sem.QUOTED q), r, c, s, p)
    | iterate (TERM (Syn.VAR i), r, c, s, p)
      = (case Env.lookup (i, r)
           of (SOME l)
              =¿ ( case Sto.fetch (l, s)
                    of (SOME v)
                       =¿ ( case v
                             of Sem.UNDEFINED
                                =¿ STUCK "attempt to reference undefined variable"
                              | _
                                =¿ iterate (VALUE v, r, c, s, p))
                     | NONE
                       =¿ STUCK "attempt to read an invalid location")
            | NONE
              =¿ STUCK "attempt to reference an undeclared variable")
    | iterate (TERM (Syn.LAM (is, t)), r, c, s, p)
      = let val (l, s’) = Sto.new (s, Sem.UNSPECIFIED)
        in iterate (VALUE (Sem.CLOSURE (l, (is, t), r)), r, c, s’, p) end
    | iterate (TERM (Syn.CND (t0, t1, t2)), r, c, s, p)
      = iterate (TERM t0, r, Sem.SELECT (t1, t2, r, c), s, p)
    | iterate (TERM (Syn.SET (i, t)), r, c, s, p)
      = iterate (TERM t, r, Sem.ASSIGN (i, r, c), s, p)
    | iterate (TERM (Syn.APP (t0, ts)), r, c, s, p)
      = let val ((pi, rev_pi_inv), p’) = Permutation.new p
            val (t0’, ts’) = rev_pi_inv (t0, ts)
        in iterate (TERM t0’, r, Sem.PUSH (ts’, nil, pi, r, c), s, p’) end
    | iterate (VALUE v, r’, Sem.HALT, s, p)
      = if Env.emptyp r’
        then RESULT (v, s)
        else iterate (VALUE v, Env.empty, Sem.HALT, s, p)
    | iterate (VALUE (Sem.QUOTED (Syn.QBOOL false)), r’, Sem.SELECT (t1, t2, r, c), s, p)
      = iterate (TERM t2, r, c, s, p)
    | iterate (VALUE _, r’, Sem.SELECT (t1, t2, r, c), s, p)
      = iterate (TERM t1, r, c, s, p)
    | iterate (VALUE v, r’, Sem.ASSIGN (i, r, c), s, p)
      = (case Env.lookup (i, r)
           of (SOME l)
              =¿ ( case Sto.fetch (l, s)
                    of (SOME v)
                       =¿ ( case v
                             of Sem.UNDEFINED
                                =¿ STUCK "attempt to assign undefined variable"
                              | _
                                =¿ iterate (VALUE Sem.UNSPECIFIED, r, c, Sto.update (l, v, s), p))
                     | NONE
                       =¿ STUCK "attempt to write an invalid location")
            | NONE
              =¿ STUCK "attempt to assign an undeclared variable")
    | iterate (VALUE v0’, r’, Sem.PUSH (nil, vs’, pi, r, c), s, p)
      = let val (v0, vs) = pi (v0’, vs’)
        in iterate (VALUE v0, r, Sem.CALL (vs, c), s, p) end
    | iterate (VALUE v0’, r’, Sem.PUSH (t1’ :: ts’, vs’, pi, r, c), s, p)
      = iterate (TERM t1’, r, Sem.PUSH (ts’, v0’ :: vs’, pi, r, c), s, p)
    | iterate (VALUE (Sem.CLOSURE (l, (is, t), r)), r’, Sem.CALL (vs, c), s, p)
      = let val (ls, s’) = Sto.news (s, vs)
        in iterate (TERM t, Env.extends (is, ls, r), c, s’, p) end
    | iterate (VALUE v, r’, Sem.CALL (vs, c), s, p)
      = STUCK "attempt to apply a non−procedure"
  ...
end
```

Figure 4. Big-step counterpart of Figure 2, part 2/2: transition function

```
structure M_big_step_defunct = struct
  val r_init = Env.empty

  datatype term_or_value = TERM of Syn.term | VALUE of Sem.value

  datatype answer = RESULT of Sem.value * Sem.value Sto.sto | STUCK of string

  fun eval ...
      = ...
  and continue ...
      = ...

  fun evaluate t
      = eval (TERM t, r_init, Sem.HALT, Sto.empty, Permutation.init)
end
```

Figure 5. Version of Figure 3 in defunctionalized form, part 1/2: configurations and initialization

8. Big-step version of Clinger's abstract machine in defunctionalized form

8.1 Without the GC rule

Like the SECD machine [16, 35], the big-step version of Clinger's abstract machine is not in defunctionalized form. Fortunately, it can easily made to be so [19], by using the type isomorphism between the transition function

```
 iterate : term_or_value * ... -¿ answer
```

and two mutually recursive transition functions

```
    eval :  term * ... -¿ answer
continue : value * ... -¿ answer.
```

The reformulated version is displayed in Figures 5 and 6, and can readily be recognized as an 'eval/apply' abstract machine [36]:[4] the 'eval' transition function dispatches on terms and the 'apply' transition function (or more accurately, the 'continue' transition function) dispatches on (the top constructor of) the context. This abstract machine is in defunctionalized form in that the evaluation context and the second transition function are the defunctionalized counterpart of a function. As shown in the next section, this function is a continuation since the refunctionalized abstract machine is in CPS.[5]

8.2 With a GC rule

As in Section 7.2, it is simple to deterministically apply the GC rule, e.g., every time a function is about to be applied.

[4] A more accurate term than 'eval/apply', though, would be 'eval/continue.'

[5] Hence the point about terminology in Footnote 4.

```
structure M_big_step_defunct = struct
  ...
  fun eval (Syn.QUOTATION q, r, c, s, p)
      = continue (Sem.QUOTED q, r, c, s, p)
    | eval (Syn.VAR i, r, c, s, p)
      = (case Env.lookup (i, r)
           of (SOME l)
              =¿ ( case Sto.fetch (l, s)
                    of (SOME v)
                       =¿ ( case v
                             of Sem.UNDEFINED
                                =¿ STUCK "attempt to reference undefined variable"
                              | _
                                =¿ continue (v, r, c, s, p))
                     | NONE
                       =¿ STUCK "attempt to read an invalid location")
            | NONE
              =¿ STUCK "attempt to reference an undeclared variable")
    | eval (Syn.LAM (is, t), r, c, s, p)
      = let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
        in continue (Sem.CLOSURE (l, (is, t), r), r, c, s', p) end
    | eval (Syn.CND (t0, t1, t2), r, c, s, p)
      = eval (t0, r, Sem.SELECT (t1, t2, r, c), s, p)
    | eval (Syn.SET (i, t), r, c, s, p)
      = eval (t, r, Sem.ASSIGN (i, r, c), s, p)
    | eval (Syn.APP (t0, ts), r, c, s, p)
      = let val ((pi, rev_pi_inv), p') = Permutation.new p
            val (t0', ts') = rev_pi_inv (t0, ts)
        in eval (t0', r, Sem.PUSH (ts', nil, pi, r, c), s, p') end
  and continue (v, r', Sem.HALT, s, p)
      = if Env.emptyp r'
        then RESULT (v, s)
        else continue (v, Env.empty, Sem.HALT, s, p)
    | continue (Sem.QUOTED (Syn.QBOOL false), r', Sem.SELECT (t1, t2, r, c), s, p)
      = eval (t2, r, c, s, p)
    | continue (_, r', Sem.SELECT (t1, t2, r, c), s, p)
      = eval (t1, r, c, s, p)
    | continue (v, r', Sem.ASSIGN (i, r, c), s, p)
      = (case Env.lookup (i, r)
           of (SOME l)
              =¿ ( case Sto.fetch (l, s)
                    of (SOME v)
                       =¿ ( case v
                             of Sem.UNDEFINED
                                =¿ STUCK "attempt to assign undefined variable"
                              | _
                                =¿ continue (Sem.UNSPECIFIED, r, c, Sto.update (l, v, s), p))
                     | NONE
                       =¿ STUCK "attempt to write an invalid location")
            | NONE
              =¿ STUCK "attempt to assign an undeclared variable")
    | continue (v0', r', Sem.PUSH (nil, vs', pi, r, c), s, p)
      = let val (v0, vs) = pi (v0', vs')
        in continue (v0, r, Sem.CALL (vs, c), s, p) end
    | continue (v0', r', Sem.PUSH (t1' :: ts', vs', pi, r, c), s, p)
      = eval (t1', r, Sem.PUSH (ts', v0' :: vs', pi, r, c), s, p)
    | continue (Sem.CLOSURE (l, (is, t), r), r', Sem.CALL (vs, c), s, p)
      = let val (ls, s') = Sto.news (s, vs)
        in eval (t, Env.extends (is, ls, r), c, s', p) end
    | continue (v, r', Sem.CALL (vs, c), s, p)
      = STUCK "attempt to apply a non—procedure"
  ...
end
```

Figure 6. Version of Figure 4 in defunctionalized form, part 2/2: transition functions

```
structure Sem = struct
  datatype value = QUOTED of Syn.quotation
                 | UNSPECIFIED
                 | UNDEFINED
                 | CLOSURE of Sto.loc * (ide list * Syn.term) * Sto.loc Env.env
      and answer = RESULT of value * value Sto.sto
                 | STUCK of string
  withtype cont = value * Sto.loc Env.env * value Sto.sto * value Permutation.permutation -¿ answer
end

structure M_big_step_refunct = struct
  val r_init = Env.empty

  datatype term_or_value = TERM of Syn.term | VALUE of Sem.value

  fun eval ...
      = ...

  fun evaluate t
      = eval (t, r_init, fn (v, _, s, p) =¿ Sem.RESULT (v, s), Sto.empty, Permutation.init)
end
```

Figure 7. Refunctionalized version of Figure 5, part 1/2: configurations and initialization

9. Big-step version of Clinger's abstract machine, refunctionalized

9.1 Without the GC rule

The refunctionalized version is displayed in Figures 7 and 8: defunctionalizing it yields back Figures 5 and 6. It is the evaluation function in continuation-passing style of a natural semantics [23, 29]. As exploited in Section 10, it is also in the range of closure conversion.

9.2 With a GC rule

Refunctionalization has made us cross a line: continuations are now higher-order, which prevents us to implement the GC rule as directly as in Section 7.2.

```
structure M_big_step_refunct = struct
  ...
  fun eval (Syn.QUOTATION q, r, c, s, p)
      = c (Sem.QUOTED q, r, s, p)
    | eval (Syn.VAR i, r, c, s, p)
      = (case Env.lookup (i, r)
           of (SOME l)
              =¿ ( case Sto.fetch (l, s)
                    of (SOME v)
                       =¿ ( case v
                             of Sem.UNDEFINED
                                =¿ Sem.STUCK "attempt to reference undefined variable"
                              | _
                                =¿ c (v, r, s, p))
                     | NONE
                       =¿ Sem.STUCK "attempt to read an invalid location")
            | NONE
              =¿ Sem.STUCK "attempt to reference an undeclared variable")
    | eval (Syn.LAM (is, t), r, c, s, p)
      = let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
        in c (Sem.CLOSURE (l, (is, t), r), r, s', p) end
    | eval (Syn.CND (t0, t1, t2), r, c, s, p)
      = eval (t0, r, fn (Sem.QUOTED (Syn.QBOOL false), r', s, p)
                        =¿ eval (t2, r, c, s, p)
                      | (_, r', s, p)
                        =¿ eval (t1, r, c, s, p),
              s, p)
    | eval (Syn.SET (i, t), r, c, s, p)
      = eval (t, r, fn (v, r', s, p)
                       =¿ ( case Env.lookup (i, r)
                             of (SOME l)
                                =¿ ( case Sto.fetch (l, s)
                                      of (SOME v)
                                         =¿ ( case v
                                               of Sem.UNDEFINED
                                                  =¿ Sem.STUCK "attempt to assign undefined variable"
                                                | _
                                                  =¿ c (Sem.UNSPECIFIED, r, Sto.update (l, v, s), p))
                                       | NONE
                                         =¿ Sem.STUCK "attempt to write an invalid location")
                              | NONE
                                =¿ Sem.STUCK "attempt to assign an undeclared variable"),
              s, p)
    | eval (Syn.APP (t0, ts), r, c, s, p)
      = let val ((pi, rev_pi_inv), p') = Permutation.new p
            val (t0', ts') = rev_pi_inv (t0, ts)
        in eval (t0', r, fn (v0', r', s, p) =¿ evlis (ts', v0', nil, pi, r, c, s, p), s, p') end
  and evlis (nil, v0', vs', pi, r, c, s, p)
      = let val (v0, vs) = pi (v0', vs')
        in case v0
             of (Sem.CLOSURE (l, (is, t), r))
                =¿ let val (ls, s') = Sto.news (s, vs)
                   in eval (t, Env.extends (is, ls, r), c, s', p)
                   end
              | _
                =¿ Sem.STUCK "attempt to apply a non-procedure" end
    | evlis (t1' :: ts', v0', vs', pi, r, c, s, p)
      = eval (t1', r, fn (v1', r', s, p) =¿ evlis (ts', v1', v0' :: vs', pi, r', c, s, p), s, p)
  ...
end
```

Figure 8. Refunctionalized version of Figure 6, part 2/2: evaluation functions

```
structure Sem = struct
  datatype value = ...
                 | CLOSURE of Sto.loc * (value list * cont * value Sto.sto *                 (* ¡¡   *)
                                         (value, Syn.term) Permutation.permutations –¿ answer) (* ¡¡   *)
      and answer = RESULT of value * value Sto.sto
                 | STUCK of string
  withtype cont = value * Sto.loc Env.env * value Sto.sto * (value, Syn.term) Permutation.permutations
                  –¿ answer
end

structure M_big_step_refunct_higher_order = struct
  (* ... *)
    | eval (Syn.LAM (is, t), r, c, s, p)
      = let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
        in c (Sem.CLOSURE (l, fn (vs, c, s, p)                                               (* ¡¡   *)
                              =¿ let val (ls, s') = Sto.news (s, vs)                          (* ¡¡   *)
                                 in eval (t, Env.extends (is, ls, r), c, s', p)              (* ¡¡   *)
                                 end),                                                        (* ¡¡   *)
              r, s', p) end
  (* ... *)
  and evlis (nil, v0', vs', pi, r, c, s, p)
      = let val (v0, vs) = pi (v0', vs')
        in case v0
             of (Sem.CLOSURE (l, f))
                =¿ f (vs, c, s, p)                                                            (* ¡¡   *)
              | _
                =¿ Sem.STUCK "attempt to apply a non–procedure" end
  (* ... *)
end
```

Figure 9. Closure-unconverted version of Figures 7 and 8 (the modified parts are marked on the right)

10. Big-step version of Clinger's abstract machine, refunctionalized and closure-unconverted

10.1 Without the GC rule

Figure 9 displays the higher-order counterpart of Figures 7 and 8: closure-converting it yields back these two figures. It is a compositional evaluation function in continuation-passing style.

10.2 With a GC rule

Closure unconversion has made us cross another line: the higher-order functions in the domain of values further prevent us to implement the GC rule as directly as in Section 7.2.

11. Analysis

11.1 From abstract machine to denotational semantics

The evaluation function of Figure 9 differs from the one in Clinger's denotational semantics in two ways:

- the continuation domains are not the same: the operational continuations are passed an environment whereas the denotational ones are not;
- the operational treatment of evlis tail-recursion differs from the denotational one: in the operational one, the sub-terms of an application are not only permuted but the result of this permutation is reversed, so that the resulting values can be iteratively accumulated in the 'evlis' function.

11.2 From denotational semantics to abstract machine

Conversely, defunctionalizing Clinger's denotational semantics yields a big-step abstract machine that also differs from the one presented at PLDI'98: it is a traditional eval/continue abstract machine where the 'continue' transition function is not passed any environment. In particular, the permuted sub-terms are not reversed prior to be evaluated and the corresponding list of values is not accumulated as in Figure 2.

11.3 Related work

Earlier on, Biernacki and the author carried out the same experiment for Propositional Prolog with cut [10]. Comparing de Bruin and de Vink's operational and denotational semantics [24], we found mismatches that are similar to the ones reported in this section.

For the rest, the literature is rich with connections and derivations between semantic artifacts. To the best of our knowledge, none are as simple and as effective as the ones pioneered by Reynolds and used here.

12. Conclusion and perspectives

We have presented new semantic specifications for Core Scheme as specified by Clinger in his PLDI'98 article. These semantic specifications are compatible and inter-derived. It is our analysis that structurally, they differ from similar semantics that have independently been published.

There are two next logical steps to this preliminary work, an analytical one and a constructive one:

- Redo this experiment on a larger scale with the other semantic specifications of Scheme. We will then be in position to compare all the small-step semantics, all the abstract machines, and all the big-step semantics of Scheme with respect to each other, and verify whether Scheme is uniformly and uniquely specified.
- Specify Scheme in part or in toto with inter-derivable specifications so that the compatibility of these specifications is a corollary of the correctness of the derivations. This corollary would let us flesh out, for example, Gasbichler, Knauel, and Sperber's conjecture of equivalence for their operational and denotational semantics of Scheme with multiple threads [27, Section 6.4].

The author does not have any opinion about the particular goodness of one or another semantic specification of Scheme. He however feels strongly that Scheme's semantic artifacts should be compatible with each other. PhD students are thus invited to apply for six-months visits to the BRICS PhD School at the University of Aarhus to help carrying out the experiments above together with the author. Any other input or collaboration is also cordially welcome.

Acknowledgments: The author is grateful to the anonymous reviewers for their comments, and to Will Clinger for extra explanations about his use of an infinite set of rules generated by a rule schema that is parameterized by a permutation and its inverse. Special thanks to Kevin Millikin and Ian Zerny for timely comments and to Małgorzata Biernacka for joining forces in a 'Part II' companion article about reduction semantics and abstract machines [8].

This work is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545.

References

[1] Mads Sig Ager. *Partial Evaluation of String Matchers & Constructions of Abstract Machines.* PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

[2] 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.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the research report BRICS RS-04-3.

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the research report BRICS RS-04-28.

[5] Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

[6] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).

[7] 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.

[8] Małgorzata Biernacka and Olivier Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II: Reduction semantics and abstract machines. In Will Clinger, editor, *Proceedings of the 2008 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Victoria, British Columbia, September 2008. Available in the present proceedings.

[9] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations.* PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, December 2005.

[10] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003,* number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[11] 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.

[12] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999. A preliminary version was presented at the 1988 ACM Conference on Lisp and Functional Programming.

[13] William Clinger and Jonathan Rees, editors. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[14] William D. Clinger. Proper tail recursion and space efficiency. In Keith D. Cooper, editor, *Proceedings of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998. ACM Press.

[15] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Gert Smolka, editor, *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 88–103, Berlin, Germany, March 2000. Springer-Verlag.

[16] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, September 2004. Springer-Verlag. Recipient of the 2004 Peter Landin prize. Extended version available as the research report BRICS RS-03-33.

[17] Olivier Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2006.

[18] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[19] Olivier Danvy and Kevin Millikin. Refunctionalization at work. Research Report BRICS RS-08-4, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, August 2007. To appear in Science of Computer Programming, extended version.

[20] Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.

[21] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.

[22] 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.

[23] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 224–242, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[24] Arie de Bruin and Erik P. de Vink. Continuation semantics for Prolog with cut. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, number 351 in Lecture Notes in Computer Science, pages 178–192, Barcelona, Spain, March 1989. Springer-Verlag.

[25] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes available at `<http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html>` and last accessed in April 2008, 1989-2001.

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

[27] Martin Gasbichler, Eric Knauel, and Michael Sperber. How to add threads to a sequential language without getting tangled up. In Matthew Flatt, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Scheme and Functional Programming*, Technical report UUCS-03-023, School of Computing, University of Utah, pages 30–47, Boston, Massachusetts, November 2003.

[28] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[29] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993. A preliminary version was presented at the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1991).

[30] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, Texas, August 1984. ACM Press.

[31] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[32] Jacob Johannsen. An investigation of Abadi and Cardelli's untyped calculus of objects. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2008. BRICS research report RS-08-6.

[33] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

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

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

[36] 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.

[37] Jacob Matthews and Robert Bruce Findler. An operational semantics for R5RS Scheme. In J. Michael Ashley and Michael Sperber, editors, *Proceedings of the Sixth ACM SIGPLAN Workshop on Scheme and Functional Programming*, Technical report TR619, Computer Science Department, Indiana University, pages 41–54, Tallinn, Estonia, September 2005.

[38] Jan Midtgaard. *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, June 2007.

[39] Kevin Millikin. *A Structured Approach to the Transformation, Normalization and Execution of Computer Programs*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, May 2007.

[40] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[41] Johan Munk. A study of syntactic and semantic artifacts and its application to lambda definability, strong normalization, and weak normalization in the presence of state. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 2007. BRICS research report RS-08-3.

[42] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-7.

[43] 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.

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

[45] John D. Ramsdell. An operational semantics for Scheme. *LISP Pointers*, V(2):6–10, April-June 1992.

[46] Jonathan Rees and William Clinger, editors. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[47] 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 [49].

[48] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword [49].

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

[50] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten, editors. Revised^6 report on the algorithmic language Scheme. Available online at `<http://www.r6rs.org/>`, September 2007.

[51] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[52] Guy L. Steele Jr. and Gerald J. Sussman. The revised report on Scheme, a dialect of Lisp. AI Memo 452, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1978.

[53] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. AI Memo 349, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1975. Reprinted in Higher-Order and Symbolic Computation 11(4):405–439, 1998, with a foreword [54].

[54] Gerald J. Sussman and Guy L. Steele Jr. The first report on Scheme revisited. *Higher-Order and Symbolic Computation*, 11(4):399–404, 1998.

[55] Philip Wadler. The essence of functional programming (invited talk). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.