Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part II: Reduction Semantics and Abstract Machines

Małgorzata Biernacka[1] and Olivier Danvy[2]

[1] Institute of Computer Science, University of Wroclaw
ul. Joliot-Curie 15, PL-50-383 Wroclaw, Poland
(mabi@ii.uni.wroc.pl)

[2] Department of Computer Science, Aarhus University
Aabogade 34, DK-8200 Aarhus N, Denmark
(danvy@brics.dk)

Abstract. We present a context-sensitive reduction semantics for a lambda-calculus with explicit substitutions and we show that the functional implementation of this small-step semantics mechanically corresponds to that of the abstract machine for Core Scheme presented by Clinger at PLDI'98, including first-class continuations. Starting from this reduction semantics, (1) we refocus it into a small-step abstract machine; (2) we fuse the transition function of this abstract machine with its driver loop, obtaining a big-step abstract machine which is staged; (3) we compress its corridor transitions, obtaining an eval/continue abstract machine; and (4) we unfold its ground closures, which yields an abstract machine that essentially coincides with Clinger's machine. This lambda-calculus with explicit substitutions therefore aptly accounts for Core Scheme, including Clinger's permutations and unpermutations.

1 Introduction

Motivation: Our motivation is the same as that of the second author in the companion paper "Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part I: Denotational Semantics, Natural Semantics, and Abstract Machines" [10]. We wish for semantic specifications that are mechanically interderivable, so that their compatibility is a corollary of the correctness of the derivations.

This work: We build on our previous work on the syntactic correspondence between context-sensitive reduction semantics and abstract machines for a λ-calculus with explicit substitutions [3,4]. Let us review each of these concepts in turn:

Abstract machines: An abstract machine is a state-transition system modeling the execution of programs. Typical abstract machines for lambda calculi treat substitution as a meta-operation and include it directly in the transitions of the machine. This approach is often used to faithfully model evaluation based on term rewriting. Alternatively, since Landin's SECD machine [12, 15], substitution is explicitly implemented in abstract machines using environments, for efficiency. The two approaches are used interchangeably in the literature (even for the same language), depending on the context of use, but their equivalence is rarely treated formally.

A λ-calculus with explicit substitutions: Since Plotkin's foundational work on λ-calculi and abstract machines [19], it has become a tradition to directly relate the result of abstract machines with the result of weak-head normalization, regardless of whether the abstract machines treat substitution implicitly as a meta-operation or explicitly with an environment. As an offshoot of his doctoral thesis [6, 7], Curien proposed a 'calculus of closures,' the $\lambda\rho$-calculus, that would, on the one hand, be faithful to the λ-calculus, and on the other hand, reflect more accurately the computational reality of abstract machines by delaying substitutions into environments. In so doing he gave birth to calculi of explicit substitutions [1], which promptly became a domain of research on their own.

In our thesis work [2,8], we revisited the $\lambda\rho$-calculus and proposed a minimal extension for it, the $\lambda\widehat{\rho}$-calculus, that is closed under one-step reduction. We then systematically applied Danvy and Nielsen's refocusing technique [13] on several reduction semantics and obtained a variety of known and new abstract machines with environments, including the Krivine machine for call by name and the CEK machine for call by value [3].

Context-sensitive reduction semantics: In his thesis work [14], Felleisen introduced a continuation-semantics analogue of structural operational semantics, reduction semantics: a small-step operational semantics with an explicit representation of the reduction context. As Strachey and Wadsworth originally did with continuation semantics [21], he then took advantage of this explicit representation of the rest of the reduction to make contraction rules context sensitive, and operate not just on a potential redex,[3] but also on its context, thereby providing the first small-step semantic account of control operators.

In our thesis work [2, 8], we considered context-sensitive contraction rules for $\lambda\widehat{\rho}$-calculi. We then systematically applied the refocusing technique on several context-sensitive reduction semantics and obtained a variety of known and new abstract machines with environments [4].

In this article, we present a variant of the $\lambda\widehat{\rho}$-calculus that, through refocusing, essentially corresponds to Clinger's abstract machine for Core Scheme as presented at PLDI'98 [5]. Curien's original point therefore applies and reductions in this calculus reflect the execution of Scheme programs accurately. We therefore put the $\lambda\widehat{\rho}$-calculus forward as an apt calculus for Core Scheme.

[3] A potential redex either is an actual one or is stuck.

Prerequisites and domain of discourse: Though one could of course use PLT Redex [17], we use a pure subset of Standard ML as a metalanguage here, for consistency with the companion paper. We otherwise expect some familiarity with programming a reduction semantics and with Clinger's PLDI'98 article [5]. For the rest, we have aimed for a stand-alone presentation but the reader might wish to consult our earlier work [3,4] or first flip through the pages of the second author's lecture notes for warm-up examples [9].

Terminology:

Notion of contraction: To alleviate the overloading of the term 'reduction' (as in, e.g., "reduction semantics," "notion of reduction," "reduction strategy," and "reduction step"), we refer to Barendregt's 'notion of reduction' as 'notion of contraction.' A notion of contraction is therefore the definition of a partial contraction function mapping a potential redex to a contractum.

Eval/continue abstract machine: As pointed out in the companion paper, an 'eval/apply' abstract machine [16] would be more accurately called 'eval/-continue' since the apply transition function, together with the data type of contexts, often form the defunctionalized representation of a continuation. We therefore use this term here.

Overview: We first present the signatures of the store, the environment, and the permutations (Section 2), and then the syntax (Section 3) and the reduction semantics (Sections 4 and 5) of a λ-calculus with explicit substitutions for Core Scheme. The resulting evaluation function is reduction-based in that it is defined as the iteration of a one-step reduction function that enumerates all the intermediate closures in the reduction sequence. We make it reduction-free by deforesting all these intermediate closures in the course of evaluation, using Danvy and Nielsen's refocusing technique (Section 6). We successively present an eval/continue abstract machine over closures that embodies the chosen reduction strategy (Section 6.1), and then an eval/continue abstract machine over terms and environments (Section 6.2). We then analyze this machine (Section 7) before concluding (Section 8).

2 Domain of discourse

In the interest of brevity and abstractness, and as in the companion paper, we only present ML signatures for the store (Section 2.1), the environment (Section 2.2) and the permutations (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 expressible values, dereference a given location in a given store, and update a given store at a given location with a given expressible value.

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

  val  empty : 'a option sto

  val    new : 'a option sto * 'a      -> loc      * 'a option sto
  val   news : 'a option sto * 'a list -> loc list * 'a option sto

  val  fetch : loc *      'a option sto -> 'a option option
  val update : loc * 'a * 'a option sto -> 'a option sto option
end


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

In his definition of Core Scheme [5, Fig. 4], Clinger lumps together storable values and expressible values [20]. In particular, he lets the undefined value be an expressible value even though in actuality it can only be a storable value (namely the value of a used, but not declared, variable). In point of fact, both dereferencing and assigning such an undeclared variable is an error. We therefore depart from Clinger's specification by defining storable values with an optional type: `NONE` denotes the undefined value and `SOME v` denotes the expressible value `v`. This way, we do not need to account for the undefined value at every turn in the derivation.

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, for a given type of identifiers.

```
type ide = string

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 definition of Scheme, the denotable values are store locations.

2.3 Permutations

The semantics of Scheme deliberately does not specify the order in which the subterms of an application are evaluated. This underspecification (also present in C for assignments) is meant to encourage programmers to explicitly sequence their side effects.

To this end, in his abstract machine, Clinger non-deterministically uses a pair of permutation functions: one over the subterms in an application, and the inverse one over the resulting values. We implement this non-determinism by threading a stream of pairs of permutations and unpermutations along with the store. We materialize this stream with the following polymorphic abstract data type.

```
signature PERM = sig
  type 'a perm = 'a * 'a list -> 'a * 'a list
  type ('v, 'c) permgen

  val init : ('v, 'c) permgen
  val  new : ('v, 'c) permgen -> ('v perm * 'c perm) * ('v, 'c) permgen
end
```

```
structure Perm : PERM = struct
  (* deliberately omitted *)
end
```

3 Syntax

The following module implements the internal syntax of Core Scheme [5, Fig. 1], for a given type of identifiers.

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

  datatype      term = QUOTE of quotation
                     | VAR of ide
                     | LAM of ide list * term
                     | APP of term * term list
                     | COND of term * term * term
                     | SET of ide * term
end
```

Terms include all the constructs of Core Scheme considered by Clinger: quoted values, identifiers, lambda abstractions, applications, conditional expressions and assignments. Primitive operators such as call/cc are declared in the initial environment.

```
structure Sem = struct
  type env = Sto.loc Env.env

  datatype primop = CWCC (* | ... *)

  datatype clo =
      CLO_GND of Syn.term * env
    | CLO_QUOTE of Syn.quotation
    | CLO_LAM of ide list * Syn.term * env * Sto.loc
    | CLO_APP of clo * clo list * value list * value Perm.perm
    | CLO_CALL of value * value list
    | CLO_COND of clo * clo * clo
    | CLO_SET of ide * env * clo
    | CLO_UNSPECIFIED
    | CLO_PRIMOP of primop * Sto.loc
    | CLO_CONT of cont * Sto.loc
  and cont =
      HALT
    | SELECT of clo * clo * cont
    | ASSIGN of ide * env * cont
    | PUSH of clo list * value list * value Perm.perm * cont
    | CALL of value list * cont
  withtype value = clo

  type sto = value option Sto.sto

  type perms = (value, clo) Perm.permgen

  datatype answer = VALUE of value * sto * perms
                  | STUCK of string

  local val (l_primop, s1) = Sto.new (Sto.empty, CLO_UNSPECIFIED)
        val (l_cwcc, s2) = Sto.new (s1, CLO_PRIMOP (CWCC, l_primop))
  in val env_init = Env.extend ("call/cc", l_cwcc, Env.empty)
     val sto_init = s2
  end
end
```

Fig. 1. Lambda-calculus with explicit substitutions for Core Scheme

4 Semantics

We consider a language of closures built on top of terms. Fig. 1 displays the syntactic categories of closures (the data type `closure`) and of contexts (the data type `cont`) as well as the notion of environment, value, store, permutation generator, and answer. Let us review each of these in turn.

4.1 The environment

As described in Section 2.2, the environment maps identifiers to denotable values, and denotable values are store locations.

4.2 Closures

As initiated by Landin [15] and continued by Curien [6, 7], a ground closure is a term paired with a syntactic representation of its environment (this pairing is done with the constructor `CLO_GND`). In a calculus of closures, small-step evaluation is defined (e.g., by a set of rewriting rules) over closures rather than over terms. Ground closures, however, are usually not expressive enough to account for one-step reductions, though they may suffice for big-step evaluation. Indeed, one-step reduction can require the internal structure of a closure to be changed in such a way that it no longer conforms to the form "(term, environment)." The data type of closures therefore contains additional constructors to represent intermediate results of one-step reductions for all the language constructs.

- The `CLO_GND` constructor is used for ground closures.
- The `CLO_QUOTE` constructor accounts for Scheme's quotations.
- The `CLO_LAM` constructor accounts for user-defined procedures, and pairs lambda-abstractions (list of formal parameters and body) together with the environment of their definition.
- The `CLO_APP` constructor accounts for applications whose subcomponents are not completely reduced yet. The closure and list of closures still need to be reduced, and the list of values holds what has already been reduced. The value permutation will be used to unpermute the complete list of values and yield the `CLO_CALL` construction.
- The `CLO_CALL` constructor accounts for applications whose subcomponents are completely reduced.
- The `CLO_COND` constructor accounts for conditional closures.
- The `CLO_SET` constructor accounts for assignments.
- The `CLO_UNSPECIFIED` constructor accounts for the unspecified value yielded, e.g., by reducing an assignment.
- The `CLO_PRIMOP` constructor accounts for predefined procedures (i.e., primitive operators) such as "call-with-current-continuation" (commonly abbreviated "call/cc"), that captures the current context.
- The `CLO_CONT` constructor accounts for escape procedures, i.e., first-class continuations as yielded by call/cc. It holds a captured context (see Section 4.5).

Note: In Scheme, procedures of course cannot be compared for mathematical equality, but they can be compared for representational identity. So for example, `(equal? (lambda (x) x) (lambda (x) x))` evaluates to `#f` but `(let ([identity (lambda (x) x)]) (equal? identity identity))` evaluates to `#t`. For better or for worse,[4] a unique location is associated to every applicable object, be it a user-defined procedure, a predefined procedure, or an escape procedure. This is the reason why the closure constructors `CLO_LAM`, `CLO_PRIMOP`, and `CLO_CONT` feature a store location.

4.3 Primitive operators

The data type `primop` groups all the predefined procedures. Here, we only consider one, call/cc.

[4] Will Clinger publically refers to this particular design as a "bug" in the semantics of Scheme.

4.4 Values

(Expressible) values are closures that cannot be decomposed into a potential redex and its reduction context, namely quotations, lambda-abstractions, primitive operators, and first-class continuations:

```
(*  valuep : Sem.clo -> bool  *)
fun valuep (CLO_QUOTE _)  = true
  | valuep (CLO_LAM _)    = true
  | valuep (CLO_PRIMOP _) = true
  | valuep (CLO_CONT _)   = true
  | valuep _              = false
```

4.5 Contexts

The data type faithfully reflects Clinger's grammar of contexts [5, Fig. 4]:

- The `HALT` constructor accounts for the empty context.
- The `SELECT` constructor accounts for the context of the test part of a conditional expression.
- The `ASSIGN` constructor accounts for the context of the subcomponent in an assignment.
- The `PUSH` constructor accounts for the context of a subcomponent in a permuted application.
- The `CALL` constructor accounts for the context of a value in position of function in an unpermuted application of values.

4.6 The store

As described in Section 2.1, the store maps locations to storable values, and storable values are either `NONE` for the undefined value or `SOME v` for the expressible value `v`.

4.7 The permutation generator

As described in Section 2.3, we thread a stream of pairs of permutations (of closures) and unpermutations (of values) along with the store.

4.8 Answers

Any non-diverging reduction sequence starting from a closure either leads to a value closure or becomes stuck. The result of a non-diverging evaluation is an answer, i.e., either an expressible value (as defined in Section 4.4) together with a store and a permutation generator, or an error message.

4.9 The initial store

The initial store holds the values of primitive operators such as call/cc.

4.10 The initial environment

The initial environment declares primitive operators such as call/cc.

```
structure Redexes = struct
  datatype potred =
      LOOKUP of ide * Sem.env
    | UNPERMUTE of Sem.value * Sem.value list * Sem.value Perm.perm
    | BETA of Sem.value * Sem.value list
    | UPDATE of ide * Sem.env * Sem.value
    | COND of Sem.value * Sem.clo * Sem.clo
    | PROC of ide list * Syn.term * Sem.env
    | PROP_APP of Syn.term * Syn.term list * Sem.env
    | PROP_COND of Syn.term * Syn.term * Syn.term * Sem.env
    | PROP_SET of ide * Syn.term * Sem.env

  datatype contractum =
      STUCK of string
    | NEXT of Sem.clo * Sem.cont * Sem.sto * Sem.perms

  (* ... *)
end
```

Fig. 2. Notion of contraction for Core Scheme (part 1/2)

5 A reduction semantics for Core Scheme

A reduction semantics is a small-step operational semantics with an explicit representation of the reduction context. It consists of a grammar of terms (here, the grammar of closures from Fig. 1), a notion of contraction specifying the basic computation steps, and a reduction strategy embodied by a grammar of reduction contexts (see Fig. 1). In this section, we present a reduction semantics for the calculus of closures introduced in Section 4.

5.1 Potential redexes and contraction

The notion of contraction is defined with two data types—one for potential redexes and one for the result of contraction (Figure 2)—and with a contraction function (Figure 3). Let us review each of these potential redexes and how they are contracted:

– `LOOKUP` – fetching the value of an identifier from the store (via its location in the environment); it succeeds only if the location corresponding to the identifier is defined in the store – otherwise reduction is stuck;
– `UNPERMUTE` – performing the unpermutation of a sequence of values before applying `BETA`-contraction;
– `BETA` – either performing the usual β-reduction for n-ary functions (when the operand is a user-defined procedure), or capturing the current context (when the operand is call/cc), or replacing the current context by a captured context (when the operand is an escape procedure);
– `UPDATE` – updating the value of an identifier in the store and returning the "unspecified" closure;
– `COND` – selecting one of the branches of a conditional expression, based on the value of its test;

```
structure Redexes = struct
  (* ... *)

  (*  contract : potred * Sem.cont * Sem.sto * Sem.perms -> contractum  *)
  fun contract (LOOKUP (i, r), rc, s, pg) =
      (case Env.lookup (i, r)
        of (SOME l)
           => (case Sto.fetch (l, s)
                of (SOME sv)
                   => (case sv
                        of (SOME v)
                           => NEXT (v, rc, s, pg)
                         | NONE
                           => STUCK "attempt to reference an undefined variable")
                 | NONE
                   => STUCK "attempt to read an invalid location")
         | NONE
           => STUCK "attempt to reference an undeclared variable")
      | contract (UNPERMUTE (v, vs, pi), rc, s, pg) =
        NEXT (Sem.CLO_CALL (pi (v, vs)), rc, s, pg)
      | contract (BETA (Sem.CLO_LAM (is, t, r, l), vs), rc, s, pg) =
        if List.length is = List.length vs
        then let val (ls, s') = Sto.news (s, vs)
             in NEXT (Sem.CLO_GND (t, Env.extends (is, ls, r)), rc, s', pg)
             end
        else STUCK "arity mismatch"
      | contract (BETA (Sem.CLO_PRIMOP (Sem.CWCC, _), vs), rc, s, pg) =
        if 1 = List.length vs
        then let val (l, s') = Sto.new (s, Sem.CLO_UNSPECIFIED)
             in NEXT (Sem.CLO_CALL (hd vs, [Sem.CLO_CONT (rc, l)]), rc, s', pg)
             end
        else STUCK "arity mismatch"
      | contract (BETA (Sem.CLO_CONT (rc', l), vs), rc, s, pg) =
        if 1 = List.length vs
        then NEXT (hd vs, rc', s, pg)
        else STUCK "arity mismatch"
      | contract (BETA (_, vs), rc, s, pg) =
        STUCK "attempt to apply a non-procedure"
      | contract (UPDATE (i, r, v), rc, s, pg) =
        (case Env.lookup (i, r)
           of (SOME l)
              => (case Sto.update (l, v, s)
                    of (SOME s')
                       => NEXT (Sem.CLO_UNSPECIFIED, rc, s', pg)
                     | NONE
                       => STUCK "attempt to write an invalid location")
            | NONE
              => STUCK "attempt to assign an undeclared variable")
      | contract (COND (Sem.CLO_QUOTE (Syn.QBOOL false), c1, c2), rc, s, pg) =
        NEXT (c2, rc, s, pg)
      | contract (COND (_, c1, c2), rc, s, pg) =
        NEXT (c1, rc, s, pg)
      | contract (PROC (is, t, r), rc, s, pg) =
        let val (l, s') = Sto.new (s, Sem.CLO_UNSPECIFIED)
        in NEXT (Sem.CLO_LAM (is, t, r, l), rc, s', pg) end
      | contract (PROP_APP (t, ts, r), rc, s, pg) =
        let val ((pi, rev_pi_inv), pg') = Perm.new pg
            val (c, cs) = rev_pi_inv (Sem.CLO_GND (t, r),
                                      map (fn t => Sem.CLO_GND (t, r)) ts)
        in NEXT (Sem.CLO_APP (c, cs, nil, pi), rc, s, pg') end
      | contract (PROP_COND (t0, t1, t2, r), rc, s, pg) =
        NEXT (Sem.CLO_COND (Sem.CLO_GND (t0, r),
                            Sem.CLO_GND (t1, r),
                            Sem.CLO_GND (t2, r)), rc, s, pg)
      | contract (PROP_SET (i, t, r), rc, s, pg) =
        NEXT (Sem.CLO_SET (i, r, Sem.CLO_GND (t, r)), rc, s, pg)
end
```

Fig. 3. Notion of contraction for Core Scheme (part 2/2)

– PROC – allocating a fresh location in the store (with an unspecified value) for a source lambda abstraction and converting this abstraction to CLO_LAM;
– PROP_APP, PROP_COND, and PROP_SET – propagating the environment into all subterms of an application, a conditional expression, and an assignment, respectively. Beside environment propagation, in PROP_APP all components of the application are permuted.

While most of the contractions account directly for the reductions in the language, the last three – the propagation contractions – are "administrative" reductions necessary to maintain the proper syntactic structure of closures after each reduction step. In addition, PROP_APP includes a permutation of all components of an application before they are evaluated in turn.

The notion of contraction depends not only on closures but also on the reduction context that can be captured by call/cc, on the store, and on the permutation generator. Therefore, all three are supplied as arguments to the contract function.

5.2 Reduction strategy

The reduction strategy is embodied in the grammar of reduction contexts defined by the data type cont in Fig. 1.

Recomposition: The function recompose reconstructs a closure given a reduction context and a (sub)closure. Its definition is displayed in Fig. 4.

Decomposition: The role of the decomposition function is to traverse a closure in a context according to the given reduction strategy and to locate the first redex to be contracted, if there is any. The decomposition function is total: it returns the closure if this closure is a value, and otherwise it returns a potential redex together with its reduction context. Its implementation is displayed in Fig. 5. In particular, decompose is called at the top level and its role is to call an auxiliary function, decompose_clo, with a closure to decompose and the

```
structure Recomposition = struct
  (*  recompose : Sem.cont * Sem.clo -> Sem.clo  *)
  fun recompose (Sem.HALT, c) =
      c
    | recompose (Sem.SELECT (c1, c2, rc), c) =
      recompose (rc, Sem.CLO_COND (c, c1, c2))
    | recompose (Sem.ASSIGN (i, r, rc), c) =
      recompose (rc, Sem.CLO_SET (i, r, c))
    | recompose (Sem.PUSH (cs, vs, p, rc), c) =
      recompose (rc, Sem.CLO_APP (c, cs, vs, p))
    | recompose (Sem.CALL (vs, rc), c) =
      recompose (rc, Sem.CLO_CALL (c, vs))
end
```

Fig. 4. The recomposition function for Core Scheme

```
structure Decomposition = struct
  datatype decomposition = VAL of Sem.value
                         | DEC of Redexes.potred * Sem.cont

  (*  decompose_clo : Sem.clo * Sem.cont -> decomposition  *)
  fun decompose_clo (Sem.CLO_GND (Syn.QUOTE q, r), rc) =
      decompose_cont (rc, Sem.CLO_QUOTE q)
    | decompose_clo (Sem.CLO_GND (Syn.VAR i, r), rc) =
      DEC (Redexes.LOOKUP (i, r), rc)
    | decompose_clo (Sem.CLO_GND (Syn.LAM (is, t), r), rc) =
      DEC (Redexes.PROC (is, t, r), rc)
    | decompose_clo (Sem.CLO_GND (Syn.APP (t, ts), r), rc) =
      DEC (Redexes.PROP_APP (t, ts, r), rc)
    | decompose_clo (Sem.CLO_GND (Syn.COND (t0, t1, t2), r), rc) =
      DEC (Redexes.PROP_COND (t0, t1, t2, r), rc)
    | decompose_clo (Sem.CLO_GND (Syn.SET (i, t), r), rc) =
      DEC (Redexes.PROP_SET (i, t, r), rc)
    | decompose_clo (v as Sem.CLO_QUOTE _, rc) =
      decompose_cont (rc, v)
    | decompose_clo (v as Sem.CLO_LAM _, rc) =
      decompose_cont (rc, v)
    | decompose_clo (Sem.CLO_APP (c, cs, vs, p), rc) =
      decompose_clo (c, Sem.PUSH (cs, vs, p, rc))
    | decompose_clo (Sem.CLO_CALL (v, vs), rc) =
      decompose_cont (Sem.CALL (vs, rc), v)
    | decompose_clo (Sem.CLO_COND (c0, c1, c2), rc) =
      decompose_clo (c0, Sem.SELECT (c1, c2, rc))
    | decompose_clo (Sem.CLO_SET (i, r, c1), rc) =
      decompose_clo (c1, Sem.ASSIGN (i, r, rc))
    | decompose_clo (v as Sem.CLO_UNSPECIFIED, rc) =
      decompose_cont (rc, v)
    | decompose_clo (v as Sem.CLO_PRIMOP (Sem.CWCC, _), rc) =
      decompose_cont (rc, v)
    | decompose_clo (v as Sem.CLO_CONT _, rc) =
      decompose_cont (rc, v)

  (*  decompose_cont : Sem.cont * Sem.value -> decomposition  *)
  and decompose_cont (Sem.HALT, v) =
      VAL v
    | decompose_cont (Sem.SELECT (c1, c2, rc), c) =
      DEC (Redexes.COND (c, c1, c2), rc)
    | decompose_cont (Sem.ASSIGN (i, r, rc), c) =
      DEC (Redexes.UPDATE (i, r, c), rc)
    | decompose_cont (Sem.PUSH (nil, vs, p, rc), v) =
      DEC (Redexes.UNPERMUTE (v, vs, p), rc)
    | decompose_cont (Sem.PUSH (c :: cs, vs, p, rc), v) =
      decompose_clo (c, Sem.PUSH (cs, v :: vs, p, rc))
    | decompose_cont (Sem.CALL (vs, rc), v) =
      DEC (Redexes.BETA (v, vs), rc)

  (*  decompose : Sem.clo -> decomposition  *)
  fun decompose c =
      decompose_clo (c, Sem.HALT)
end
```

Fig. 5. The decomposition function for Core Scheme

empty context. In turn, `decompose_clo` traverses a closure and accumulates the current context until a potential redex or a value closure is found; in the latter case, `decompose_cont` is called in order to dispatch on the accumulated context for this given value.

The decomposition function can be expressed in a variety of ways. In Fig. 5, we have conveniently specified it as a big-step abstract machine with two transition functions, `decompose_clo` and `decompose_cont`.

5.3 One-step reduction

One-step reduction can now be defined with the following steps: (a) decomposing a non-value closure into a potential redex and a reduction context, (b) contracting the potential redex if it is an actual one, and (c) recomposing the contractum into the context.

```
(*  reduce : Clo.clo * Clo.sto -> Clo.clo option  *)
fun reduce (c, s, pg) =
    (case Decomposition.decompose c
       of (Decomposition.VAL v)
          => SOME (v, s, pg)
        | (Decomposition.DEC (pr, rc))
          => (case Redexes.contract (pr, rc, s, pg)
                of (Redexes.NEXT (c', rc', s', pg'))
                   => SOME (Recomposition.recompose (rc', c'), s', pg')
                 | (Redexes.STUCK msg)
                   => NONE))
```

5.4 Reduction-based evaluation

Finally, we can define evaluation as the iteration of one-step reduction. We implement it as the iteration of (a) decomposition, (b) contraction, and (c) recomposition.

```
(*  iterate : Decomposition.decomposition * Sem.sto  -> Sem.answer  *)
fun iterate (Decomposition.VAL v, s, pg)
    = Sem.VALUE (v, s, pg)
  | iterate (Decomposition.DEC (pr, rc), s, pg)
    = (case Redexes.contract (pr, rc, s, pg)
         of Redexes.NEXT (c', rc', s', pg')
            => let val c = Recomposition.recompose (rc', c')
                   val d = Decomposition.decompose c
               in iterate (d, s', pg')
               end
          | Redexes.STUCK msg
            => Sem.STUCK msg)

(*  evaluate : Syn.term -> Sem.answer  *)
fun evaluate t
    = iterate (Decomposition.decompose (Sem.CLO_GND (t, Sem.env_init)),
               Sem.sto_init,
               Perm.init)
```

6 Refocusing for reduction-free evaluation

We use Danvy and Nielsen's refocusing technique to mechanically transform the iteration of one-step reduction implemented in Section 5.4 into an abstract machine. In this section we show the main steps of this transformation and their effect on the Core Scheme calculus of closures.

The reduction sequence as described in Section 5 consists in repeating the following steps: decomposing a closure into a potential redex and a context, contracting the redex when it is an actual one, and recomposing the context with the contractum, thereby obtaining the next closure in the reduction sequence. The recomposition operation creates an intermediate closure (the next one in the reduction sequence) which is then immediately decomposed in the next iteration. Using refocusing, we can bypass the creation of intermediate closures and proceed directly from one redex to the next. The method is based on the observation that the composition of functions `recompose` and `decompose` can be replaced by a more efficient function, called `refocus`, which is extensionally equal to (and optimally implemented by) `decompose_clo`. The situation is depicted in the following diagram:

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

6.1 An eval/continue abstract machine over closures

First, we fuse the functions `recompose` and `decompose` into one function `refocus` that given a closure and its surrounding context, searches for the next redex according to the given reduction strategy. The result is a small-step state-transition system, where `refocus` performs a single transition to the next redex site, if there is one, and `iterate` implements its iteration (after performing the contraction).

Next, we distribute the calls to `iterate` in the definition of `refocus` in order to obtain a big-step state-transition system [11]. The difference between the big-step and the small-step transition system is that in the former, the function `refocus` does not stop on encountering a redex site; it calls the function `iterate` directly. The resulting big-step transition system is presented in Figs. 6 and 7, where `refocus_clo` is an alias for the `refocus` function described above. (The definition of `refocus_clo` and `refocus_cont` is a clone of the definition of `decompose_clo` and `decompose_cont` in Figure 5.)

This resulting transition system is *staged* in that the call to the contraction function is localized in `iterate` whereas `refocus_clo` and `refocus_cont` implement the congruence rules, i.e., the navigation in a closure towards the next redex. Inlining the definition of `iterate` (and thus making do without the data type `decomposition`) yields an eval/continue abstract machine with two mutually recursive transition functions: `refocus_clo` that dispatches on closures, and `refocus_cont` that dispatches on contexts.

```
structure EAC_AM = struct
  datatype decomposition = VAL of Sem.value
                         | DEC of Redexes.potred * Sem.cont

  (*  refocus_clo : Sem.clo * Sem.cont * Sem.sto * Sem.perms -> Sem.answer  *)
  fun refocus_clo (Sem.CLO_GND (Syn.QUOTE q, r), rc, s, pg) =
      refocus_cont (rc, Sem.CLO_QUOTE q, s, pg)
    | refocus_clo (Sem.CLO_GND (Syn.VAR i, r), rc, s, pg) =
      iterate (DEC (Redexes.LOOKUP (i, r), rc), s, pg)
    | refocus_clo (Sem.CLO_GND (Syn.LAM (is, t), r), rc, s, pg) =
      iterate (DEC (Redexes.PROC (is, t, r), rc), s, pg)
    | refocus_clo (Sem.CLO_GND (Syn.APP (t, ts), r), rc, s, pg) =
      iterate (DEC (Redexes.PROP_APP (t, ts, r), rc), s, pg)
    | refocus_clo (Sem.CLO_GND (Syn.COND (t0, t1, t2), r), rc, s, pg) =
      iterate (DEC (Redexes.PROP_COND (t0, t1, t2, r), rc), s, pg)
    | refocus_clo (Sem.CLO_GND (Syn.SET (i, t), r), rc, s, pg) =
      iterate (DEC (Redexes.PROP_SET (i, t, r), rc), s, pg)
    | refocus_clo (v as Sem.CLO_QUOTE _, rc, s, pg) =
      refocus_cont (rc, v, s, pg)
    | refocus_clo (v as Sem.CLO_LAM _, rc, s, pg) =
      refocus_cont (rc, v, s, pg)
    | refocus_clo (Sem.CLO_APP (c, cs, vs, p), rc, s, pg) =
      refocus_clo (c, Sem.PUSH (cs, vs, p, rc), s, pg)
    | refocus_clo (Sem.CLO_CALL (v, vs), rc, s, pg) =
      refocus_cont (Sem.CALL (vs, rc), v, s, pg)
    | refocus_clo (Sem.CLO_COND (c0, c1, c2), rc, s, pg) =
      refocus_clo (c0, Sem.SELECT (c1, c2, rc), s, pg)
    | refocus_clo (Sem.CLO_SET (i, r, c1), rc, s, pg) =
      refocus_clo (c1, Sem.ASSIGN (i, r, rc), s, pg)
    | refocus_clo (v as Sem.CLO_UNSPECIFIED, rc, s, pg) =
      refocus_cont (rc, v, s, pg)
    | refocus_clo (v as Sem.CLO_PRIMOP (Sem.CWCC, _), rc, s, pg) =
      refocus_cont (rc, v, s, pg)
    | refocus_clo (v as Sem.CLO_CONT _, rc, s, pg) =
      refocus_cont (rc, v, s, pg)

  (*  refocus_cont : Sem.cont * Sem.value * Sem.sto * Sem.perms -> Sem.answer  *)
  and refocus_cont (Sem.HALT, v, s, pg) =
      iterate (VAL v, s, pg)
    | refocus_cont (Sem.SELECT (c1, c2, rc), c, s, pg) =
      iterate (DEC (Redexes.COND (c, c1, c2), rc), s, pg)
    | refocus_cont (Sem.ASSIGN (i, r, rc), c, s, pg) =
      iterate (DEC (Redexes.UPDATE (i, r, c), rc), s, pg)
    | refocus_cont (Sem.PUSH (nil, vs, p, rc), v, s, pg) =
      iterate (DEC (Redexes.UNPERMUTE (v, vs, p), rc), s, pg)
    | refocus_cont (Sem.PUSH (c :: cs, vs, p, rc), v, s, pg) =
      refocus_clo (c, Sem.PUSH (cs, v :: vs, p, rc), s, pg)
    | refocus_cont (Sem.CALL (vs, rc), v, s, pg) =
      iterate (DEC (Redexes.BETA (v, vs), rc), s, pg)
  and iterate ... =
      ...

  (*  evaluate : Syn.term -> Sem.answer  *)
  fun evaluate t
       = refocus_clo (Sem.CLO_GND (t, Sem.env_init),
                      Sem.HALT,
                      Sem.sto_init,
                      Perm.init)
end
```

Fig. 6. Staged eval/apply/continue abstract machine over closures (part 1/2)

```
structure EAC_AM = struct
  (* ... *)
  and iterate (VAL v, s, pg) =
      Sem.VALUE (v, s, pg)
    | iterate (DEC (Redexes.LOOKUP (i, r), rc), s, pg) =
      (case Env.lookup (i, r)
         of (SOME l)
            => (case Sto.fetch (l, s)
                  of (SOME sv)
                     => (case sv
                           of (SOME v)
                              => refocus_cont (rc, v, s, pg)
                            | NONE
                           => Sem.STUCK "attempt to reference an undefined variable")
                   | NONE
                     => Sem.STUCK "attempt to read an invalid location")
          | NONE
            => Sem.STUCK "attempt to reference an undeclared variable")
    | iterate (DEC (Redexes.UNPERMUTE (v, vs, pi), rc), s, pg) =
      refocus_clo (Sem.CLO_CALL (pi (v, vs)), rc, s, pg)
    | iterate (DEC (Redexes.BETA (Sem.CLO_LAM (is, t, r, l), vs), rc), s, pg) =
      if List.length is = List.length vs
      then let val (ls, s') = Sto.news (s, vs)
           in refocus_clo (Sem.CLO_GND (t, Env.extends (is, ls, r)), rc, s', pg)
           end
      else Sem.STUCK "arity mismatch"
    | iterate (DEC (Redexes.BETA (Sem.CLO_PRIMOP (Sem.CWCC, _), vs), rc), s, pg) =
      if 1 = List.length vs
      then let val (l, s') = Sto.new (s, Sem.CLO_UNSPECIFIED)
         in refocus_clo (Sem.CLO_CALL (hd vs, [Sem.CLO_CONT (rc, l)]), rc, s', pg)
           end
      else Sem.STUCK "arity mismatch"
    | iterate (DEC (Redexes.BETA (Sem.CLO_CONT (rc', l), vs), rc), s, pg) =
      if 1 = List.length vs
      then refocus_cont (rc', hd vs, s, pg)
      else Sem.STUCK "arity mismatch"
    | iterate (DEC (Redexes.BETA (_, vs), rc), s, pg) =
      Sem.STUCK "attempt to apply a non-procedure"
     | iterate (DEC (Redexes.UPDATE (i, r, v), rc), s, pg)
       = (case Env.lookup (i, r)
           of (SOME l)
              => (case Sto.update (l, v, s)
                    of (SOME s')
                       => iterate (refocus_cont (rc, Sem.CLO_UNSPECIFIED), s', pg)
                     | NONE
                       => Sem.STUCK "attempt to write an invalid location")
            | NONE
              => Sem.STUCK "attempt to assign an undeclared variable")
    | iterate (DEC (Redexes.COND (Sem.CLO_QUOTE (Syn.QBOOL false), c1, c2), rc),
             s, pg) =
      refocus_clo (c2, rc, s, pg)
    | iterate (DEC (Redexes.COND (_, c1, c2), rc), s, pg) =
      refocus_clo (c1, rc, s, pg)
    | iterate (DEC (Redexes.PROC (is, t, r), rc), s, pg) =
      let val (l, s') = Sto.new (s, Sem.CLO_UNSPECIFIED)
      in refocus_cont (rc, Sem.CLO_LAM (is, t, r, l), s', pg) end
    | iterate (DEC (Redexes.PROP_APP (t, ts, r), rc), s, pg) =
      let val ((pi, rev_pi_inv), pg') = Perm.new pg
          val (c, cs) = rev_pi_inv (Sem.CLO_GND (t, r),
                                    map (fn t => Sem.CLO_GND (t, r)) ts)
      in refocus_clo (Sem.CLO_APP (c, cs, nil, pi), rc, s, pg') end
    | iterate (DEC (Redexes.PROP_COND (t0, t1, t2, r), rc), s, pg) =
      refocus_clo (Sem.CLO_COND (Sem.CLO_GND (t0, r),
                                 Sem.CLO_GND (t1, r),
                                 Sem.CLO_GND (t2, r)), rc, s, pg)
    | iterate (DEC (Redexes.PROP_SET (i, t, r), rc), s, pg) =
      refocus_clo (Sem.CLO_SET (i, r, Sem.CLO_GND (t, r)), rc, s, pg)
  (* ... *)
end
```

Fig. 7. Staged eval/apply/continue abstract machine over closures (part 2/2)

6.2 An abstract machine over terms and environments

The result of applying refocusing to the calculus of closures is a machine operating on closures, as shown in Section 6.1. Since we are not interested in modeling the execution of programs in the closure calculus, but in Core Scheme, i.e., with explicit terms and environments, we go the rest of the way and bypass closure manipulation using the method developed in our previous work [3, 4, 9].

To this end, we first short-circuit the 'corridor' transitions corresponding to building intermediate closures – these are the transitions corresponding to the propagation of environments in closures: specifically, we observe that each of the closures built with `CLO_COND`, `CLO_APP` and `CLO_SET` is immediately consumed in exactly one of the clauses of `refocus` after being constructed. Since these

```
structure EC_AM = struct
  type env = Sto.loc Env.env

  datatype primop = CWCC

  datatype clo = CLO_GND of Syn.term * env

  datatype value = VAL_QUOTE of Syn.quotation
                 | VAL_UNSPECIFIED
                 | VAL_LAM of (ide list * Syn.term) * env * Sto.loc
                 | VAL_PRIMOP of primop * Sto.loc
                 | VAL_CONT of cont * Sto.loc
  and       cont = HALT
                 | SELECT of clo * clo * cont
                 | ASSIGN of ide * env * cont
                 | PUSH of clo list * value list * value Perm.perm * cont
                 | CALL of value list * cont

  local val (l_primop, s1) = Sto.new (Sto.empty, VAL_UNSPECIFIED)
        val (l_cwcc, s2) = Sto.new (s1, VAL_PRIMOP (CWCC, l_primop))
  in val env_init = Env.extend ("call/cc", l_cwcc, Env.empty)
     val sto_init = s2
  end

  type sto = value option Sto.sto

  type perms = (value, clo) Perm.permgen

  datatype answer = VALUE of value * sto * perms
                  | STUCK of string

  (*  eval : Syn.term * env * cont * sto * perms -> answer  *)
  fun eval ... =
      ...
  (*  continue : cont * value * sto * perms -> answer  *)
  and continue ... =
      ...

  (*  evaluate : Syn.term -> answer  *)
  fun evaluate t =
      eval (t, env_init, HALT, sto_init, Perm.init)
end
```

Fig. 8. The eval/continue abstract machine over terms and environments (part 1/3)

```
structure EC_AM = struct
  (* ... *)
  fun eval (Syn.QUOTE q, r, rc, s, pg) =
      continue (rc, VAL_QUOTE q, s, pg)
    | eval (Syn.VAR i, r, rc, s, pg) =
      (case Env.lookup (i, r)
         of (SOME l)
            => (case Sto.fetch (l, s)
                  of (SOME sv)
                     => (case sv
                           of (SOME v)
                              => continue (rc, v, s, pg)
                            | NONE
                              => STUCK "attempt to reference an undefined variable")
                   | NONE
                     => STUCK "attempt to read an invalid location")
          | NONE
            => STUCK "attempt to reference an undeclared variable")
    | eval (Syn.LAM (is, t), r, rc, s, pg) =
      let val (l, s') = Sto.new (s, VAL_UNSPECIFIED)
      in continue (rc, VAL_LAM ((is, t), r, l), s', pg) end
    | eval (Syn.APP (t, ts), r, rc, s, pg) =
      let val ((pi, rev_pi_inv), pg') = Perm.new pg
          val (CLO_GND (t', r'), cs) = rev_pi_inv (CLO_GND (t, r),
                                                    map (fn t => CLO_GND (t, r)) ts)
      in eval (t', r', PUSH (cs, nil, pi, rc), s, pg) end
    | eval (Syn.COND (t0, t1, t2), r, rc, s, pg) =
      eval (t0, r, SELECT (CLO_GND (t1, r), CLO_GND (t2, r), rc), s, pg)
    | eval (Syn.SET (i, t), r, rc, s, pg) =
      eval (t, r, ASSIGN (i, r, rc), s, pg)
  (* ... *)
end
```

Fig. 9. The eval/continue abstract machine over terms and environments (part 2/3)

closures were only needed to express intermediate results of one-step reduction (and they do not arise from the Core Scheme term language), we can merge the two clauses of `refocus` for each such closure. We then obtain a machine that operates only on `CLO_GND` closures, which are pairs of terms and environments. Hence, we can unfold a closure `CLO_GND (t, s)` into a term and an environment. (The reader is directed to our previous work for numerous other examples of this derivation [3, 4, 9].) This final machine is displayed in Figs. 8, 9, and 10. It is an eval/continue abstract machine for Core Scheme terms, in which an eval configuration consists of a term, an environment, a context, a store, and a permutation generator, and a continue configuration consists of a context, a value, a store, and a permutation generator.[5] The eval transition function dispatches on the term and the continue transition function on the context.

[5] This machine is the same one as in the companion paper [10]. As pointed out there, it is in defunctionalized form: refunctionalizing it yields the continuation-passing evaluation function of a natural semantics, and closure-unconverting this evaluation function yields the compositional valuation function of a denotational semantics.

```
structure EC_AM = struct
  (* ... *)
  and continue (HALT, v, s, pg) =
      VALUE (v, s, pg)
    | continue (SELECT (CLO_GND (t1, r1), CLO_GND (t2, r2), rc),
                VAL_QUOTE (Syn.QBOOL false), s, pg) =
      eval (t2, r2, rc, s, pg)
    | continue (SELECT (CLO_GND (t1, r1), CLO_GND (t2, r2), rc),
                _, s, pg) =
      eval (t1, r1, rc, s, pg)
    | continue (ASSIGN (i, r, rc), v, s, pg) =
      (case Env.lookup (i, r)
         of (SOME l)
            => (case Sto.update (l, v, s)
                  of (SOME s')
                     => continue (rc, VAL_UNSPECIFIED, s', pg)
                   | NONE
                     => STUCK "attempt to write an invalid location")
          | NONE
            => STUCK "attempt to assign an undeclared variable")
    | continue (PUSH (nil, vs, pi, rc), v, s, pg) =
      let val (v', vs') = pi (v, vs)
      in continue (CALL (vs', rc), v', s, pg) end
    | continue (PUSH ((CLO_GND (t, r)) :: cs, vs, p, rc), v, s, pg) =
      eval (t, r, PUSH (cs, v :: vs, p, rc), s, pg)
    | continue (CALL (vs, rc), VAL_LAM ((is, t), r, l), s, pg) =
      if List.length is = List.length vs
      then let val (ls, s') = Sto.news (s, vs)
           in eval (t, Env.extends (is, ls, r), rc, s', pg) end
      else STUCK "arity mismatch"
    | continue (CALL (vs, rc), VAL_PRIMOP (VAL_CWCC, _), s, pg) =
      if 1 = List.length vs
      then let val (l, s') = Sto.new (s, VAL_UNSPECIFIED)
           in continue (CALL ([VAL_CONT (rc, l)], rc), hd vs, s', pg) end
      else STUCK "arity mismatch"
    | continue (CALL (vs, rc), VAL_CONT (rc', l), s, pg) =
      if 1 = List.length vs
      then continue (rc', hd vs, s, pg)
      else STUCK "arity mismatch"
    | continue (CALL (vs, rc), _, s, pg) =
      STUCK "attempt to apply a non-procedure"
  (* ... *)
end
```

Fig. 10. The eval/continue abstract machine over terms and environments (part 3/3)

7 Analysis

Compared to Figs. 8, 9, and 10, Clinger's machine [5, Fig. 5] has one configuration and two transition functions. This single configuration is a tuple and it is, so to speak, the superposition of our two configurations.

The single real difference between Clinger's machine and the one of Figs. 8, 9, and 10 is that it dissociates subterms and the current environment. In contrast, the propagation rules of our calculus of closures ensure that terms and environments stick together at all times.

Ergo, the variant of the $\lambda\widehat{\rho}$-calculus presented here aptly accounts for Core Scheme. An obvious next step is to scale this calculus to full Scheme and to compare it with the reduction semantics in the R^6RS. One could also refocus the reduction semantics of the R^6RS to obtain the corresponding abstract machine.

This abstract machine would then provide a sound alternative semantics for the R^{6}RS.

8 Conclusion and perspectives

We have presented a version of the $\lambda\widehat{\rho}$-calculus and its reduction semantics, and we have transformed a functional implementation of this reduction semantics into the functional implementation of an abstract machine. This abstract machine is essentially the same as the abstract machine for Core Scheme presented by Clinger at PLDI'98. The transformations are the ones we have already used in the past to derive other abstract machines from other reduction semantics, or to posit a reduction semantics and verify whether transforming it yields a given abstract machine.

This work is part of a larger effort to inter-derive semantic specifications soundly and consistently.

Acknowledgments. The authors are grateful to the anonymous reviewers of the 2008 Workshop on Scheme and Functional Programming for their comments, to Will Clinger for extra precisions, and to Ian Zerny for some timely feedback. This work is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545. We are happy to dedicate it to Peter Mosses for his 60th birthday, and we wish him many happy returns.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. A preliminary version was presented at the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL 1990).
2. Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages.* PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, January 2006.
3. Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007. Article #6. Extended version available as the research report BRICS RS-06-3.
4. 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. 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.
6. Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

7. Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Progress in Theoretical Computer Science. Birkhaüser, 1993.
8. Olivier Danvy. *An Analytical Approach to Program as Data Objects.* DSc thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, October 2006.
9. Olivier Danvy. From reduction-based to reduction-free normalization. In *Advanced Functional Programming, Sixth International School*, Lecture Notes in Computer Science, Nijmegen, The Netherlands, May 2008. Springer-Verlag. Lecture notes of 100+ pages including 70+ exercises. To appear.
10. Olivier Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, part I: Denotational semantics, natural semantics, and abstract machines. Companion article submitted to the Peter Mosses Festschrift, April 2009.
11. 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.
12. Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's SECD machine with the J operator. *Logical Methods in Computer Science*, 4(4:12):1–67, November 2008.
13. Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, Aarhus University, 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.
14. 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.
15. Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
16. Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.
17. Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference (RTA 2004)*, number 3091 in Lecture Notes in Computer Science, pages 301–311, Aachen, Germany, June 2001. Springer-Verlag.
18. Peter D. Mosses. A foreword to 'Fundamental concepts in programming languages'. *Higher-Order and Symbolic Computation*, 13(1/2):7–9, 2000.
19. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
20. Christopher Strachey. Fundamental concepts in programming languages. International Summer School in Computer Programming, Copenhagen, Denmark, August 1967. Reprinted in Higher-Order and Symbolic Computation 13(1/2):11–49, 2000, with a foreword [18].
21. Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford

University Computing Laboratory, Programming Research Group, Oxford, England, 1974. Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000, with a foreword [22].
22. Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.