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

Małgorzata Biernacka
Institute of Computer Science
University of Wroclaw *
mabi@ii.uni.wroc.pl

Olivier Danvy
Department of Computer Science
University of Aarhus †
danvy@brics.dk

Abstract

We present a context-sensitive reduction semantics for a lambda-calculus with explicit substitutions and store and we show that the functional implementation of this small-step semantics mechanically corresponds to that of an abstract machine. This abstract machine is very close to the abstract machine for Core Scheme presented by Clinger at PLDI'98. This lambda-calculus with explicit substitutions and store therefore aptly accounts for Core Scheme.

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: Abstract Machines, Natural Semantics, and Denotational Semantics" [11]. 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 [15], substitution is explicitly implemented in abstract machines using environments. 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 [18], 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, 9], 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 ex-

* ul. Joliot-Curie 15, PL-50-383 Wroclaw, Poland
http://www.ii.uni.wroc.pl/~mabi
† IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
http://www.brics.dk/~danvy

2008 Workshop on Scheme and Functional Programming

plicit representation of the reduction context. As Strachey and Wadsworth originally did with continuation semantics [19], 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,[1] but also on its context, thereby providing the first small-step semantic account of control operators.

In our thesis work [2, 9], 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.

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 standalone 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 [8, 10].

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 and the environment (Section 2), and then the syntax (Section 3) and the reduction semantics (Section 4) of a Core Scheme calculus of closures. 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 5). We successively present an eval/-continue abstract machine for the Core Scheme calculus of closures that embodies the chosen reduction strategy (Section 5.1), and then an eval/continue abstract machine for terms and with environments (Section 5.2). We then analyze this machine (Section 6) before concluding (Section 7).

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

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) and the environment (Section 2.2).

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
```

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.

```
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.

3. Syntax

The following module implements the internal syntax of Core Scheme [5, 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
                | UNSPECIFIED
                | LAM_LOC of ide list * term *
                             Sto.loc
end
```

Terms include all the constructs of Core Scheme considered by Clinger: quoted values, identifiers, lambda abstractions, applications, conditional expressions and assignments. Two new syntax constructors occur for the course of reduction: one to account for the result of assignments (special term `UNSPECIFIED`) and one to account for lambda-closures being associated to a store location.[2]

We consider a language of closures built on top of terms. The following module defines the syntactic category of closures (in the data type `closure`) and store closures (in the data type `closure_store`).

As initiated by Landin and continued by Curien, 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)." We therefore introduce three more constructors in the data type of closures that represent intermediate results of one-step reductions for conditional expressions, applications and assignments. These additional constructors are used to propagate the environment in sub-terms.

```
structure Clo = struct
  datatype closure =
    CLO_GND of Syn.term * environment
  | CLO_COND of closure * closure * closure
  | CLO_APP of closure * closure list
  | CLO_SET of closure * closure
  withtype environment = Sto.loc Env.env

  type store = closure Sto.sto
  type closure_store = closure * store

  datatype answer = VALUE of closure_store
                  | STUCK of string
end
```

Moreover, the notion of contraction for Core Scheme depends not only on closures built out of terms and environments, but also on the store. Therefore we introduce the category of store closures, the entity on which reductions operate (rather than just on terms or just ground closures).

Any terminating reduction sequence starting from a store closure either leads to a store closure (a value store closure) or becomes stuck. The result of a non-diverging evaluation is reflected in the above data type of answers.

4. Reduction semantics

A reduction semantics is a small-step operational semantics with an explicit representation of the reduction context. It consists in a grammar of terms (here, the grammar of store closures from Section 3), a notion of contraction specifying the basic computation steps, and a reduction strategy embodied by a grammar of reduction contexts. In this section, we present a reduction semantics for the calculus of closures introduced in Section 3.

4.1 Potential redexes and contraction

The notion of contraction is displayed in Figure 1 with a data type for potential redexes and a contraction function. 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 the reduction is stuck
- `BETA` – performing the usual β-reduction for n-ary functions; it can only take place if the function argument is already represented by the `LAM_LOC` constructor, i.e., when a location in the store has been allocated for the function
- `UPDATE` – updating the value of an identifier in the store and returning the "unspecified" closure, i.e., a closure built from the term `UNSPECIFIED`
- `CONDITIONAL` – selecting one of the branches of a conditional expression, based on the value of its test
- `ABSTRACTION` – allocating a fresh location in the store (with an unspecified value) for a source lambda abstraction and converting this abstraction to `LAM_LOC`
- `PROP_APP`, `PROP_COND`, and `PROP_SET` – propagating the environment into all subterms of an application, a conditional expression, and an assignment, respectively

While most of the contractions account directly for the reductions in the language, the last three – the propagation

[2] In an extended version of this article, we address Clinger's permute/unpermute functions. There, we need a third syntax constructor for the course of reduction, to account for applications whose sub-terms have been permuted.

```
structure Redexes = struct
  datatype potential_redex =
    LOOKUP of ide * Sto.loc Env.env * Clo.closure Sto.sto
  | BETA of Clo.closure * Clo.closure list * Clo.closure Sto.sto
  | UPDATE of Clo.closure * Clo.closure * Clo.closure Sto.sto
  | CONDITIONAL of Clo.closure * Clo.closure * Clo.closure * Clo.closure Sto.sto
  | ABSTRACTION of ide list * Syn.term * Sto.loc Env.env * Clo.closure Sto.sto
  | PROP_APP of Syn.term * Syn.term list * Sto.loc Env.env * Clo.closure Sto.sto
  | PROP_COND of Syn.term * Syn.term * Syn.term * Sto.loc Env.env * Clo.closure Sto.sto
  | PROP_SET of ide * Syn.term * Sto.loc Env.env * Clo.closure Sto.sto

  datatype contractum = CLO of Clo.closure_store
                      | STUCK of string

  fun contract (LOOKUP (i, r, s))                              (* potential_redex -> contractum *)
    = (case Env.lookup (i, r)
        of SOME l
           => (case Sto.fetch (l, s)
                 of SOME v
                    => CLO (v, s)
                  | NONE
                    => STUCK "attempt to read an invalid location")
         | NONE
           => STUCK "attempt to reference an undeclared variable")
    | contract (BETA (Clo.CLO_GND (Syn.LAM_LOC (is, t, _), r), vs, s))
      = let val (ls, s') = Sto.news (s, vs)
        in CLO (Clo.CLO_GND (t, Env.extends (is, ls, r)), s') end
    | contract (UPDATE (c as Clo.CLO_GND (Syn.VAR i, r), v, s))
      = (case Env.lookup (i, r)
          of SOME l
             => CLO (Clo.CLO_GND (Syn.UNSPECIFIED, r), Sto.update (l, v, s))
           | NONE
             => STUCK "attempt to assign an undeclared variable")
    | contract (CONDITIONAL (Clo.CLO_GND (Syn.QUOTATION (Syn.QBOOL false), r), c1, c2, s))
      = CLO (c2, s)
    | contract (CONDITIONAL (_, c1, c2, s))
      = CLO (c1, s)
    | contract (ABSTRACTION (is, t, r, s))
      = let val (l, s') = Sto.new (s, Clo.CLO_GND (Syn.UNSPECIFIED, Env.empty))
        in CLO (Clo.CLO_GND (Syn.LAM_LOC (is, t, l), r), s') end
    | contract (PROP_APP (t, ts, r, s))
      = let val (c :: cs) = rev (map (fn t => Clo.CLO_GND (t, r)) (t :: ts))
        in CLO (Clo.CLO_APP (c, cs), s) end
    | contract (PROP_COND (t0, t1, t2, r, s))
      = CLO (Clo.CLO_COND (Clo.CLO_GND (t0, r), Clo.CLO_GND (t1, r), Clo.CLO_GND (t2, r)), s)
    | contract (PROP_SET (i, t, r, s))
      = CLO (Clo.CLO_SET (Clo.CLO_GND (Syn.VAR i, r), Clo.CLO_GND (t, r)), s)
end
```

Figure 1. Notion of contraction for the Core Scheme calculus of closures

contractions – are "administrative" reductions necessary to maintain the proper syntactic structure of closures after each reduction step.

4.2 Reduction strategy

The reduction strategy is embodied in the grammar of reduction contexts (defined in the data type `context` in the module below) and the associated function `plug` that reconstructs the closure given a (sub)closure and a context. Similarly, the function `plug_sto` reconstructs a store closure from its decomposition.

The only deviation from Clinger's reduction strategy is that we consider fixed, right-to-left order of reducing function arguments. (We address Clinger's permute/unpermute functions in an extended version of this article.)

```
structure Contexts = struct
  datatype context =
    HALT
  | SELECT of Clo.closure * Clo.closure * context
  | ASSIGN of Clo.closure * context
  | PUSH of Clo.closure list * Clo.closure list *
            context
```

```
structure Decomposition : DECOMPOSITION
= struct
    datatype decomposition = VAL of Clo.closure_store
                           | DEC of Redexes.potential_redex * Contexts.context

                              (* decompose' : (closure * store) * Contexts.context -> decomposition *)
    fun decompose' ((Clo.CLO_GND (Syn.QUOTATION q, r), s), rc)
      = decompose'_aux (rc, s, (Clo.CLO_GND (Syn.QUOTATION q, r)))
      | decompose' ((Clo.CLO_GND (Syn.VAR i, r), s), rc)
        = DEC (Redexes.LOOKUP (i, r, s), rc)
      | decompose' ((c as Clo.CLO_GND (Syn.LAM (is, t), r), s), rc)
        = DEC (Redexes.ABSTRACTION (is, t, r, s), rc)
      | decompose' ((Clo.CLO_GND (Syn.APP (t,ts), r), s), rc)
        = DEC (Redexes.PROP_APP (t, ts, r, s), rc)
      | decompose' ((Clo.CLO_GND (Syn.CND (t0, t1, t2), r), s), rc)
        = DEC (Redexes.PROP_COND (t0, t1, t2, r, s), rc)
      | decompose' ((Clo.CLO_GND (Syn.SET (i, t), r), s), rc)
        = DEC (Redexes.PROP_SET (i, t, r, s), rc)
      | decompose' ((Clo.CLO_APP (c0, cs), s), rc)
        = decompose' ((c0, s), Contexts.PUSH (cs, nil, rc))
      | decompose' ((Clo.CLO_SET (c0, c1), s), rc)
        = decompose' ((c1, s), Contexts.ASSIGN (c0, rc))
      | decompose' ((Clo.CLO_COND (c0, c1, c2), s), rc)
        = decompose' ((c0, s), Contexts.SELECT (c1, c2, rc))
      | decompose' ((Clo.CLO_GND (Syn.UNSPECIFIED, r), s), rc)
        = decompose'_aux (rc, s, Clo.CLO_GND (Syn.UNSPECIFIED, r))
      | decompose' ((c as Clo.CLO_GND (Syn.LAM_LOC (is, t, l), r), s), rc)
        = decompose'_aux (rc, s, c)
                             (* decompose'_aux : Contexts.context * store * closure -> decomposition *)
    and decompose'_aux (Contexts.HALT, s, c)
        = VAL (c, s)
      | decompose'_aux (Contexts.PUSH (cnext::cs, vcs, rc), s, c)
        = decompose' ((cnext, s), Contexts.PUSH (cs, c::vcs, rc))
      | decompose'_aux (Contexts.PUSH (nil, vcs, rc), s, c)
        = DEC (Redexes.BETA (c, vcs, s), rc)
      | decompose'_aux (Contexts.SELECT (c1, c2, rc), s, c)
        = DEC (Redexes.CONDITIONAL (c, c1, c2, s), rc)
      | decompose'_aux (Contexts.ASSIGN (c0, rc), s, c)
        = DEC (Redexes.UPDATE (c0, c, s), rc)

    fun decompose (c, s)                                  (* decompose : closure_store -> decomposition *)
        = decompose' ((c, s), Contexts.HALT)
  end
```

Figure 2. The decomposition structure for the calculus of closures

```
  (* Clo.closure * context -> Clo.closure *)
    fun plug (c, HALT)
        = c
      | plug (c, SELECT (c1, c2, rc))
        = plug (Clo.CLO_COND (c, c1, c2), rc)
      | plug (c, ASSIGN (c0, rc))
        = plug (Clo.CLO_SET (c0, c), rc)
      | plug (c, PUSH (tcs, nil, rc))
        = plug (Clo.CLO_APP (c, tcs), rc)
      | plug (c, PUSH (tcs, v::vcs, rc))
        = plug (Clo.CLO_APP (v, vcs@(c::tcs)), rc)

  (* Clo.closure_store * context -> context_store *)
    fun plug_sto ((c, sto), rc)
        = (plug (c, rc), sto)
  end
```

The constructors in the definition of contexts define respectively: the empty context, the context of a condition in a conditional expression, the context of a value to be assigned to a variable, and the context of an immediate subterm in an application.

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. The signature of the decomposition function is shown below, and its implementation is displayed in Figure 2. In particular, `decompose` is called at the top level and its role is to call

an auxiliary function, `decompose'`, with a store closure to decompose and the empty context. In turn, `decompose'` traverses a closure and accumulates the current context until a potential redex or a value closure is found; in the latter case, `decompose'_aux` is called in order to dispatch on the accumulated context for this given value.

```
signature DECOMPOSITION = sig
  datatype decomposition =
    VAL of Clo.closure_store
  | DEC of Redexes.potential_redex *
           Contexts.context

  val decompose : closure_store -> decomposition
end
```

The decomposition function can be expressed in a variety of ways. In Figure 2, we have conveniently specified it as a big-step abstract machine with two transition functions, `decompose'` and `decompose'_aux`.

4.3 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) plugging the contractum into the context.

```
(*  reduce : Clo.closure * Clo.store
             -> Clo.closure option    *)

fun reduce (c, s)
= (case Decomposition.decompose (c, s)
     of Decomposition.VAL cs
        => SOME cs
      | Decomposition.DEC (pr, rc)
        => (case Redexes.contract pr
              of Redexes.CLO c
                 => SOME
                      (Contexts.plug_sto
                        (c, rc))
               | Redexes.STUCK s
                 => NONE))
```

4.4 Reduction-based evaluation

Finally, we can define evaluation as the iteration of one-step reduction. It is implemented by the following function `evaluate`.

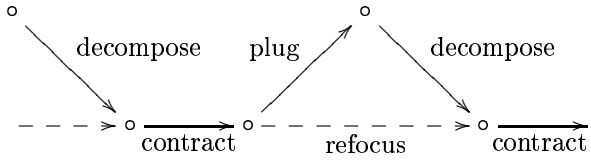
```
fun iterate (Decomposition.VAL c)
    = Clo.VALUE c
  | iterate (Decomposition.DEC (r, rc))
    = (case Redexes.contract r
         of Redexes.CLO c
            => iterate
                 (Decomposition.decompose
                    (Contexts.plug_sto
                       (c, rc)))
          | Redexes.STUCK s
            => Clo.STUCK s)

fun evaluate t
    = iterate
        (Decomposition.decompose
           (Clo.CLO_GND (t, nil), Sto.empty))
```

5. 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 4 into an abstract machine. In this section we show the main steps of this procedure and their effect on the Core Scheme calculus of closures.

The reduction sequence as described in Section 4 consists in repeating the following steps: decomposing a term into a redex and a context, contracting the redex, and plugging the contractum in the context. The plugging operation creates an intermediate term which is then immediately decomposed in the next iteration. Using refocusing, we can bypass the creation of intermediate terms and proceed directly from one redex to the next. The method is based on the observation that the composition of functions `plug` and `decompose` can be replaced by a more efficient function, called `refocus`, which is extensionally equal to (and optimally implemented by) `decompose'`. The situation is depicted in the following diagram:



5.1 An eval/continue abstract machine over closures

First, we fuse the functions `plug` 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 [12]. 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 Figure 3. (The definition of `refocus` and `refocus_aux` is a clone of the definition of `decompose'` and `decompose'_aux` in Figure 2.)

This resulting transition system is *staged* in that the call to the contraction function is localized in `iterate` whereas `refocus` and `refocus_aux` implement the congruence rules, i.e., the navigation in the 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` that dispatches on closures, and `refocus_aux` that dispatches on contexts. This machine operates on store closures and is presented in Figure 4.

```
structure BS_TS = struct
  datatype decomposition = VAL of Clo.closure_store
                         | DEC of Redexes.contractum * Contexts.context

  fun refocus ((Clo.CLO_GND (Syn.QUOTATION q, r), s), rc)
    = refocus_aux (rc, s, (Clo.CLO_GND (Syn.QUOTATION q, r)))
    | refocus ((Clo.CLO_GND (Syn.VAR i, r), s), rc)
      = iterate (DEC
            (case Env.lookup (i, r)
              of SOME l => (case Sto.fetch (l, s)
                              of SOME v => Redexes.CLO (v, s)
                               | NONE   => Redexes.STUCK "attempt to read an invalid location")
               | NONE     => Redexes.STUCK "attempt to reference an undeclared variable", rc))
    | refocus ((c as Clo.CLO_GND (Syn.LAM (is, t), r), s), rc)
      = iterate (DEC (let val (l, s') = Sto.new (s, Clo.CLO_GND (Syn.UNSPECIFIED, Env.empty))
                      in Redexes.CLO (Clo.CLO_GND (Syn.LAM_LOC (is, t, l), r), s') end, rc))
    | refocus ((Clo.CLO_GND (Syn.APP (t,ts), r), s), rc)
      = let val (c :: cs) = rev (map (fn t => Clo.CLO_GND (t, r)) (t :: ts))
        in iterate (DEC (Redexes.CLO (Clo.CLO_APP (c, cs), s), rc)) end
    | refocus ((Clo.CLO_GND (Syn.CND (t0, t1, t2), r), s), rc)
      = iterate (DEC (Redexes.CLO (Clo.CLO_COND (Clo.CLO_GND (t0, r),
                                                 Clo.CLO_GND (t1, r),
                                                 Clo.CLO_GND (t2, r)), s), rc))
    | refocus ((Clo.CLO_GND (Syn.SET (i, t), r), s), rc)
      = iterate (DEC (Redexes.CLO (Clo.CLO_SET (Clo.CLO_GND (Syn.VAR i, r), Clo.CLO_GND (t, r)), s),
                      rc))
    | refocus ((Clo.CLO_APP (c0, cs), s), rc)
      = refocus ((c0, s), Contexts.PUSH (cs, nil, rc))
    | refocus ((Clo.CLO_SET (c0, c1), s), rc)
      = refocus ((c1, s), Contexts.ASSIGN (c0, rc))
    | refocus ((Clo.CLO_COND (c0, c1, c2), s), rc)
      = refocus ((c0, s), Contexts.SELECT (c1, c2, rc))
    | refocus ((Clo.CLO_GND (Syn.UNSPECIFIED, r), s), rc)
      = refocus_aux (rc, s, Clo.CLO_GND (Syn.UNSPECIFIED, r))
    | refocus ((c as Clo.CLO_GND (Syn.LAM_LOC (is, t, l), r), s), rc)
      = refocus_aux (rc, s, c)
  and refocus_aux (Contexts.HALT, s, c)
      = iterate (VAL (c, s))
    | refocus_aux (Contexts.PUSH (cnext :: cs, vcs, rc), s, c)
      = refocus ((cnext, s), Contexts.PUSH (cs, c :: vcs, rc))
    | refocus_aux (Contexts.PUSH (nil, vcs, rc), s, c0 as Clo.CLO_GND (Syn.LAM_LOC (is, t, _), r))
      = let val (ls, s') = Sto.news (s, vcs)
        in iterate (DEC (Redexes.CLO (Clo.CLO_GND (t, Env.extends (is, ls, r)), s'), rc)) end
    | refocus_aux (Contexts.SELECT (c1, c2, rc), s, Clo.CLO_GND (Syn.QUOTATION (Syn.QBOOL false), r))
      = iterate (DEC (Redexes.CLO (c2, s), rc))
    | refocus_aux (Contexts.SELECT (c1, c2, rc), s, _)
      = iterate (DEC (Redexes.CLO (c1, s), rc))
    | refocus_aux (Contexts.ASSIGN (c0 as Clo.CLO_GND (Syn.VAR i, r), rc), s, c)
      = iterate (DEC (case Env.lookup (i, r)
          of SOME l => Redexes.CLO (Clo.CLO_GND (Syn.UNSPECIFIED, r), Sto.update (l, c, s))
           | NONE   => Redexes.STUCK "attempt to assign an undeclared variable", rc))
  and iterate (VAL c)                    = Clo.VALUE c
    | iterate (DEC (Redexes.CLO c, rc))  = refocus (c, rc)
    | iterate (DEC (Redexes.STUCK s, rc)) = Clo.STUCK s

  fun evaluate t
      = refocus ((Clo.CLO_GND (t, nil), Sto.empty), Contexts.HALT)
end
```

Figure 3. The staged big-step state-transition system over closures

```
structure EC_AM = struct
  fun refocus ((Clo.CLO_GND (Syn.QUOTATION q, r), s), rc)
    = refocus_aux (rc, s, (Clo.CLO_GND (Syn.QUOTATION q, r)))
    | refocus ((Clo.CLO_GND (Syn.VAR i, r), s), rc)
      = (case Env.lookup (i, r)
          of SOME l
             => (case Sto.fetch (l, s)
                  of SOME v
                     => refocus ((v, s), rc)
                   | NONE
                     => Clo.STUCK "attempt to read an invalid location")
           | NONE
             => Clo.STUCK "attempt to reference an undeclared variable")
    | refocus ((c as Clo.CLO_GND (Syn.LAM (is, t), r), s), rc)
      = refocus (let val (l, s') = Sto.new (s, Clo.CLO_GND (Syn.UNSPECIFIED, Env.empty))
                 in (Clo.CLO_GND (Syn.LAM_LOC (is, t, l), r), s')
                 end, rc)
    | refocus ((Clo.CLO_GND (Syn.APP (t,ts), r), s), rc)
      = let val (c::cs) = rev (map (fn t => Clo.CLO_GND (t, r)) (t::ts))
        in refocus ((Clo.CLO_APP (c, cs), s), rc) end
    | refocus ((Clo.CLO_GND (Syn.CND (t0, t1, t2), r), s), rc)
      = refocus ((Clo.CLO_COND (Clo.CLO_GND (t0, r), Clo.CLO_GND (t1, r), Clo.CLO_GND (t2, r)), s), rc)
    | refocus ((Clo.CLO_GND (Syn.SET (i, t), r), s), rc)
      = refocus ((Clo.CLO_SET (Clo.CLO_GND (Syn.VAR i, r), Clo.CLO_GND (t, r)), s), rc)
    | refocus ((Clo.CLO_APP (c0, cs), s), rc)
      = refocus ((c0, s), Contexts.PUSH (cs, nil, rc))
    | refocus ((Clo.CLO_SET (c0, c1), s), rc)
      = refocus ((c1, s), Contexts.ASSIGN (c0, rc))
    | refocus ((Clo.CLO_COND (c0, c1, c2), s), rc)
      = refocus ((c0, s), Contexts.SELECT (c1, c2, rc))
    | refocus ((Clo.CLO_GND (Syn.UNSPECIFIED, r), s), rc)
      = refocus_aux (rc, s, Clo.CLO_GND (Syn.UNSPECIFIED, r))
    | refocus ((c as Clo.CLO_GND (Syn.LAM_LOC (is, t, l), r), s), rc)
      = refocus_aux (rc, s, c)
  and refocus_aux (Contexts.HALT, s, c)
      = Clo.VALUE (c, s)
    | refocus_aux (Contexts.PUSH (cnext::cs, vcs, rc), s, c)
      = refocus ((cnext, s), Contexts.PUSH (cs, c::vcs, rc))
    | refocus_aux (Contexts.PUSH (nil, vcs, rc), s, Clo.CLO_GND (Syn.LAM_LOC (is, t, _), r))
      = let val (ls, s') = Sto.news (s, vcs)
        in refocus ((Clo.CLO_GND (t, Env.extends (is, ls, r)), s'), rc) end
    | refocus_aux (Contexts.SELECT (c1, c2, rc), s, Clo.CLO_GND (Syn.QUOTATION (Syn.QBOOL false), r))
      = refocus ((c2, s), rc)
    | refocus_aux (Contexts.SELECT (c1, c2, rc), s, _)
      = refocus ((c1, s), rc)
    | refocus_aux (Contexts.ASSIGN (c0 as Clo.CLO_GND (Syn.VAR i, r), rc), s, c)
      = (case Env.lookup (i, r)
          of SOME l
             => refocus ((Clo.CLO_GND (Syn.UNSPECIFIED, r), Sto.update (l, c, s)), rc)
           | NONE
             => Clo.STUCK "attempt to assign an undeclared variable")

  fun evaluate t
      = refocus ((Clo.CLO_GND (t, nil), Sto.empty), Contexts.HALT)
end
```

Figure 4. The eval/continue abstract machine over closures

```
structure Env_EC_AM = struct
  type environment = Sto.loc Env.env

  type closure = Syn.term * environment

  type store = closure Sto.sto

  datatype context = HALT
                   | SELECT of closure * closure * context
                   | ASSIGN of closure * context
                   | PUSH of closure list * closure list * context

  datatype answer = VALUE of closure * store
                  | STUCK of string

  fun refocus (Syn.QUOTATION q, r, s, rc)
    = refocus_aux (rc, s, (Syn.QUOTATION q, r))
    | refocus (Syn.VAR i, r, s, rc)
      = (case Env.lookup (i, r)
          of SOME l => (case Sto.fetch (l, s)
                          of SOME (t, r) => refocus (t, r, s, rc)
                           | NONE        => STUCK "attempt to read an invalid location")
           | NONE     => STUCK "attempt to reference an undeclared variable")
    | refocus (Syn.LAM (is, t), r, s, rc)
      = let val (l, s') = Sto.new (s, (Syn.UNSPECIFIED, Env.empty))
        in refocus_aux (rc, s', (Syn.LAM_LOC (is, t, l), r)) end
    | refocus (Syn.APP (t, ts), r, s, rc)
      = let val ((t, r) :: cs) = rev (map (fn t => (t, r)) (t :: ts))
        in refocus (t, r, s, PUSH (cs, nil, rc)) end
    | refocus (Syn.CND (t0, t1, t2), r, s, rc)
      = refocus (t0, r, s, SELECT ((t1, r), (t2, r), rc))
    | refocus (Syn.SET (i, t), r, s, rc)
      = refocus (t, r, s, ASSIGN ((Syn.VAR i, r), rc))
    | refocus (Syn.UNSPECIFIED, r, s, rc)
      = refocus_aux (rc, s, (Syn.UNSPECIFIED, r))
  and refocus_aux (HALT, s, c)
      = VALUE (c, s)
    | refocus_aux (PUSH ((t,r) :: cs, vcs, rc), s, c)
      = refocus (t, r, s, PUSH (cs, c :: vcs, rc))
    | refocus_aux (PUSH (nil, vcs, rc), s, c0 as (Syn.LAM_LOC (is, t, _), r))
      = let val (ls, s') = Sto.news (s, vcs)
        in refocus (t, Env.extends (is, ls, r), s', rc) end
    | refocus_aux (SELECT (c1, (t2, r2), rc), s, (Syn.QUOTATION (Syn.QBOOL false), r))
      = refocus (t2, r2, s, rc)
    | refocus_aux (SELECT ((t1, r1), c2, rc), s, _)
      = refocus (t1, r1, s, rc)
    | refocus_aux (ASSIGN (c0 as (Syn.VAR i, r), rc), s, c)
      = (case Env.lookup (i, r)
          of SOME l => refocus (Syn.UNSPECIFIED, r, Sto.update (l, c, s), rc)
           | NONE   => STUCK "attempt to assign an undeclared variable")

  fun evaluate t
      = refocus (t, nil, Sto.empty, HALT)
end
```

Figure 5. The eval/continue abstract machine over terms and environments

5.2 An abstract machine with environments

The result of applying refocusing to the calculus of closures is a machine operating on store closures, as shown in Section 5.1. 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].

To this end, we first short-circuit the transitions corresponding to building intermediate closures – these are the transition corresponding to the propagation of environments in closures: specifically, we observe that each of the clo-

sures 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 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, 8].) This final machine is shown in Figure 5. It is an eval/continue abstract machine for Core Scheme terms, in which a configuration consists of a term, an environment, a store (all three arising by unfolding a store closure) and a reduction context.

6. Analysis

Overall, the abstract machine of Figure 5 is an 'eval/continue' one. The 'eval' transition function operates on a quadruple—a term, an environment, a store, and a context—and dispatches on the term. The 'continue' transition function operates on a triple—a value, a store, and a context—and dispatches on the context.[3] In comparison, Clinger's machine has one configuration and one transition function. This single configuration is a quadruple, and this single transition function is, so to speak, the union of our two transition functions.

The single real difference between Clinger's machine and the one of Figure 5 is that it dissociates sub-terms and the current environment. In contrast, the propagation rules of the $\lambda\widehat{\rho}$-calculus ensures that terms and environments stick together at all times.

Finally, Clinger's machine also features permutations/unpermutations functions to account for the unspecified sequencing order to evaluate the sub-terms of an application. In an extended version of this article, we modify the reduction semantics to syntactically account for these permutations/unpermutations functions as we did for associating locations to lambda-closures in Section 4.1. The resulting abstract machine then essentially coincides with Clinger's.

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^{6}RS. One could also refocus the reduction semantics of the R^{6}RS to obtain the corresponding abstract machine. This abstract machine would then provide a sound alternative semantics for the R^{6}RS.

[3] This machine is the same one as in the companion paper [11]. As pointed out there, this abstract machine 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.

7. Conclusion and perspectives

We have presented a version of the $\lambda\widehat{\rho}$-calculus with a store 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 very close to 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 for their comments, and to Will Clinger for extra precisions.

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

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, University of Aarhus, 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. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, volume 124(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

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

[10] 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. To appear.

[11] Olivier Danvy. Towards compatible and interderivable semantic specifi cations for the Scheme programming language, Part I: Denotational semantics, natural 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.

[12] 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.

[13] 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.

[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. In Kathleen Fisher, editor, *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, SIGPLAN Notices, Vol. 39, No. 9, pages 4–15, Snowbird, Utah, September 2004. ACM Press.

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

[19] 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 [20].

[20] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.