BRICS

Basic Research in Computer Science

BRICS RS-02-53 O. Danvy: A Lambda-Revelation of the SECD Machine

A Lambda-Revelation of the SECD Machine

Olivier Danvy

BRICS Report Series RS-02-53

ISSN 0909-0878 December 2003

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone:	**+45 8942 3360**
Telefax:	**+45 8942 3255**
Internet:	**BRICS@brics.dk**

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/02/53/`

A lambda-revelation of the SECD machine ⋆

Olivier Danvy

BRICS, Department of Computer Science, University of Aarhus
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
Phone: (+45) 89 42 33 69. Fax: (+45) 89 42 32 55. E-mail: `danvy@brics.dk`

Abstract. We present a simple inter-derivation between λ-interpreters, i.e., functional evaluators for λ-terms, and abstract reduction machines for the λ-calculus. The two key derivation steps are the CPS transformation and Reynolds's defunctionalization. By transforming the interpreter into continuation-passing style (CPS), its flow of control is made manifest as a continuation. By defunctionalizing this continuation, the flow of control is materialized as a first-order data structure.
The derivation applies not merely to connect independently known λ-interpreters and abstract machines, it also applies to construct the abstract machine corresponding to a λ-interpreter and to construct the λ-interpreter corresponding to an abstract machine. In this article, we treat in detail the canonical example of Landin's SECD machine and we reveal its denotational content: the meaning of an expression is a partial endo-function from a stack of intermediate results and an environment to a new stack of intermediate results and an environment. The corresponding λ-interpreter is unconventional because (1) it uses a control delimiter to evaluate the body of each λ-abstraction and (2) it assumes the environment to be managed in a callee-save fashion instead of in the usual caller-save fashion.

1 Introduction

The literature abounds with elegant derivations of abstract machines for the lambda-calculus, with one remarkable exception: Landin's original SECD machine [15]. It was the first such abstract machine, it is the starting point of many university courses and textbooks, and it has been the topic of many variations and optimizations, be it for its source language (call by name, call by need, other syntactic constructs, including control operators), for its environment (de Bruijn indices, de Bruijn levels, explicit substitutions, higher-order abstract syntax), or for its control (proper tail recursion, one stack instead of two). Yet in over thirty-five years of existence, it has not been derived or reconstructed.

The goal of this article is to show how to deconstruct the SECD machine into a λ-interpreter (i.e., an evaluator for applicative expressions, i.e., lambda-terms). We use a combination of simple tools and we do so in a way that generally applies to other λ-interpreters and other abstract machines.

⋆ December 2002.

1.1 What

We show that the denotational content of the SECD machine is a semantics where the meaning of a term is a partial endo-function

$$S \times E \rightharpoonup S \times E$$

where S is a domain of stack of intermediate values and E is a domain of environments. This interpreter is unconventional in that its environment is managed in a callee-save fashion.

The methodology used to reveal the denotational content of the SECD machine applies to variants of it and also is reversible. Starting from variants of the SECD machine, it mechanically leads one to the corresponding direct-style evaluator. Conversely, starting from a canonical, direct-style evaluator for lambda-terms—one that manages its environment in the usual caller-save fashion—the methodology mechanically leads one to Felleisen's CK, CEK, etc. abstract machines [8, 10], including control operators and state operations.

1.2 How

We use a combination of fold-unfold transformation, CPS transformation, and defunctionalization.

Fold-unfold. We re-express the top-level loop of the SECD machine from one recursive function to four mutually recursive functions. The reasoning uses Burstall and Darlington's fold-unfold strategy [4] and the correctness proof is by fixed-point induction, as in Bekic's theorem [25].

The CPS transformation. A λ-term is CPS-transformed by naming each of its intermediate results, by sequentializing the computation of these results, and by introducing continuations. Equivalently, such a term can be first transformed into monadic normal form and then translated into the term model of the continuation monad [13]. Alternatively, the CPS-transformation over types corresponds to a double-negation translation.

For example, a term such as $\lambda f.\lambda g.\lambda x.f\ x\ (g\ x)$ is named and sequentialized into

$$\begin{array}{l}\lambda f.\lambda g.\lambda x.\mathsf{let}\ v_1 = f\ x\\ \qquad\qquad\ \mathsf{in\ let}\ v_2 = g\ x\\ \qquad\qquad\quad \mathsf{in}\ v_1\ v_2\end{array}$$

and its CPS counterpart reads as

$$\lambda k.k\ (\lambda f.\lambda k.k\ (\lambda g.\lambda k.k\ (\lambda x.f\ x\ (\lambda v_1.g\ x\ (\lambda v_2.v_1\ v_2\ k)))))$$

under call by value. In both of the sequentialized version and the CPS version, v_1 names the result of $f\ x$ and v_2 names the result of $g\ x$.

Defunctionalization. In a higher-order program, first-class functions arise as instances of function abstractions. Often, these function abstractions can be enumerated, either exhaustively or more discriminately using a control-flow analysis [21]. Defunctionalization is *a transformation where function types are replaced by an enumeration of the function abstractions in this program.*

Defunctionalization is to control-flow analysis what offline program specialization is to binding-time analysis [14]: It is a natural consumer of the results of this program analysis. Whereas an offline program specializer evaluates the static parts of a program and reconstructs its dynamic parts, a defunctionalizer replaces:

- function spaces by an enumeration, in the form of a data type, of the possible lambda-abstractions that can float there;
- function introduction by a construction into the corresponding data type; and
- function elimination by an apply function dispatching over elements of the corresponding data type.

For example, let us defunctionalize the function space `int -> int` in the following ML program:

```
fun aux f
    = (f 1) + (f 10)
fun main (x, y)
    = (aux (fn z => z)) * (aux (fn z => x + y + z))
```

The `aux` function is passed a first-class function, applies it to 1 and 10, and sums the results. The `main` function calls `aux` twice and multiplies the results. All in all, two function abstractions occur in this program, in `main`.

Defunctionalizing this program amounts to defining a data type with two constructors, one for each function abstraction, and its associated apply function. The first function abstraction contains no free variables and therefore the first data-type constructor is constant. The second function abstraction contains two free variables (`x` and `y`, of type integer), and therefore the second data-type constructor requires two integers.

In `main_def`, the first functional argument is thus introduced with the first constructor, and the second functional argument with the second constructor and the values of `x` and `y`. In `aux_def`, the functional argument is passed to a second-class function `apply` that eliminates it with a case expression dispatching over the two constructors.

```
datatype lam = LAM1
             | LAM2 of int * int
fun apply (LAM1, z)
    = z
  | apply (LAM2 (x, y), z)
     = x + y + z
```

```
fun aux_def f
    = (apply (f, 1)) + (apply (f, 10))
fun main_def (x, y)
    = (aux_def LAM1) * (aux_def (LAM2 (x, y)))
```

Defunctionalization was discovered by Reynolds thirty years ago [19]. It has been little used in practice since then, and has only been formalized over the last few years [1, 2, 18]. More detail can be found in Danvy and Nielsen's study [7].

1.3 Domain of discourse

We use ML as a meta-language. We assume a basic familiarity about Standard ML and reasoning about ML programs. In particular, given two ML expressions `e` and `e'` we write `e` $\cong$ `e'` to express that `e` and `e'` are observationally equivalent.

The source language. The source language is the λ-calculus, extended with literals (as observables). A program is a closed term.

```
structure Source
= struct
    type ide = string
    datatype term = LIT of int
                  | VAR of ide
                  | LAM of ide * term
                  | APP of term * term
    type program = term
  end
```

The environment. We make use of a structure `Env` satisfying the following signature:

```
signature ENV
= sig
    type 'a env
    exception UNBOUND
    val empty : 'a env
    val extend : Source.ide * 'a * 'a env -> 'a env
    val lookup : Source.ide * 'a env -> 'a
  end
```

The empty environment is denoted by `Env.empty`. The function extending an environment with a new binding is denoted by `Env.extend`. The function to fetch the value of an identifier in an environment is denoted by `Env.lookup`. (Unbound identifiers are handled by raising an exception, which never happens for closed terms.)

Expressible and denotable values. There are three kinds of values: integers, the successor function, and function closures:

```
datatype value = INT of int
               | SUCC
               | CLOSURE of value Env.env * Source.ide * Source.term
```

Function closures are canonically defined as pairing a λ-abstraction (i.e., its formal parameter and its body) and the environment of its declaration, as promoted by Landin [15] and used ever since.

The initial environment. We define the successor function in the initial environment:

```
val e_init = Env.extend ("succ", SUCC, Env.empty)
```

1.4 Overview

Section 2 presents the SECD machine as classically specified in the literature, i.e., as one tail-recursive function `run`. Section 3 presents an alternative specification where `run` is disentangled into four mutually (tail) recursive functions `run_c`, `run_d`, `run_t`, and `run_s`, each of which has one and only one induction variable. Section 4 identifies that the disentangled definition is in defunctionalized form, and presents its higher-order counterpart. This higher-order counterpart is in continuation-passing style, and Section 5 presents its direct-style equivalent. This direct-style equivalent is again in continuation-passing style would it be for the fact that not all of its calls are tail calls, which is characteristic of delimited control. Section 6 presents the corresponding direct-style evaluator, which uses a control delimiter. Section 7 revisits the issue of compositionality and closures in functional evaluators and presents the denotational content of the SECD machine. Section 8 assesses the methodology of going back and forth between lambda-interpreters and abstract machines, and Section 9 concludes.

2 The SECD machine as specified in the literature

The SECD machine is defined as a state-transition system with four components:

- A *stack* register holding a list of intermediate results. This component has type `value list`.
- An *environment* register holding the current environment. This component has type `value Env.env`.
- A *control* register holding a list of control directives. This component has type `directive`, where `directive` is defined as follows:

```
datatype directive = TERM of term
                   | APPLY
```

– A *dump* register holding a list of triples. Each triple contains the previous contents of the stack, environment, and control registers. This component has type `(value list * value Env.env * control) list`.

The SECD machine is canonically defined as the following transitive closure of a set of transitions between its four components [15].

```
(*  run : S * E * C * D -> value  *)
(*  where S = value list          *)
(*        E = value Env.env       *)
(*        C = directive list      *)
(*        D = (S * E * C) list    *)
fun run (v :: nil, e', nil, nil)                                (* 1 *)
    = v
  | run (v :: nil, e', nil, (s, e, c) :: d)                     (* 2 *)
    = run (v :: s, e, c, d)
  | run (s, e, (TERM (LIT n)) :: c, d)                          (* 3 *)
    = run ((INT n) :: s, e, c, d)
  | run (s, e, (TERM (VAR x)) :: c, d)                          (* 4 *)
    = run ((Env.lookup (x, e)) :: s, e, c, d)
  | run (s, e, (TERM (LAM (x, t))) :: c, d)                     (* 5 *)
    = run ((CLOSURE (e, x, t)) :: s, e, c, d)
  | run (s, e, (TERM (APP (t0, t1))) :: c, d)                   (* 6 *)
    = run (s, e, (TERM t1) :: (TERM t0) :: APPLY :: c, d)
  | run (SUCC :: (INT n) :: s, e, APPLY :: c, d)                (* 7 *)
    = run ((INT (n+1)) :: s, e, c, d)
  | run ((CLOSURE (e', x, t)) :: v :: s, e, APPLY :: c, d)      (* 8 *)
    = run (nil, Env.extend (x, v, e'), (TERM t) :: nil, (s, e, c) :: d)

(*  evaluate0 : Source.program -> value  *)
fun evaluate0 t                                                 (* 9 *)
    = run (nil, e_init, (TERM t) :: nil, nil)
```

Essentially:

1. The first clause specifies that the machine terminates when both the current list of control directives and the current dump are empty.
2. The second clause specifies what to do if the current list of control directives is empty but the current dump is not empty, which corresponds to a function return: the computation should continue with the stack, environment, and control stored in the top-most component of the dump, transferring the top-most value of the current stack onto the new stack.
3. The third clause specifies what to do if the top current control directive is a literal, which corresponds to evaluating a literal: the corresponding value should be pushed on the current stack.
4. The fourth clause specifies what to do if the top current control directive is an identifier, which corresponds to evaluating an identifier: the corresponding value should be fetched in the current environment and pushed on the current stack.

5. The fifth clause specifies what to do if the top current control directive is a λ-abstraction, which corresponds to evaluating a λ-abstraction: the corresponding function closure should be pushed on the current stack. This closure groups the current environment, and the two components of the λ-abstraction, i.e., its formal parameter and its body.
6. The sixth clause specifies what to do if the top current control directive is an application, which corresponds to evaluating an application: an apply directive, the operator, and the operand should be pushed on the list of control directives.
7. The seventh clause specifies what to do if the top current control directive is an apply directive, the top of the current stack is the successor function, and the next element in the current stack is an integer, which corresponds to the application of the successor function: the current stack should be popped twice and the integer should be incremented and pushed on the stack.
8. The eighth clause specifies what to do if the top current control directive is an apply directive, the top of the current stack is a closure, and there is a next element in the current stack, which corresponds to a function call: the stack should be popped twice and, together with the current environment and the rest of the list of control directives, pushed on the dump (thereby saving the current state of the machine). The current stack should be initialized with the empty list, the current environment should be initialized with the closure environment, suitably extended, and the current list of directives should be initialized with the body of the closure.
9. Evaluation is initialized with an empty current stack, the initial environment, the expression to evaluate as a single control directive, and an empty dump.

The SECD machine does not terminate for divergent terms. If it becomes stuck, the result is an ML pattern-matching error (alternatively, the co-domain of `run` could be made `value option` and an else clause could be added). Otherwise, the result of the evaluation is `v` for some ML value `v : value`.

3 A more structured specification of the SECD machine

In the definition of Section 2, all the possible transitions are meshed together in one recursive function, `run`. Instead, let us factor `run` into several mutually recursive functions, each of them with one induction variable.

In this disentangled definition,

– `run_c` interprets the list of control directives, i.e., it specifies which transition to take if the list is empty, starts with a term, or starts with an apply directive. If the list is empty, it calls `run_d`. If the list starts with a term, it calls `run_t`, caching the term in a fifth component (the first parameter of `run_t`). If the list starts with an apply directive, it calls `run_s`.
– `run_d` interprets the dump, i.e., it specifies which transition to take if the dump is empty or non-empty, given a valid stack.
– `run_t` interprets the top term in the list of control directives.
– `run_s` interprets the top value in the current stack.

```
(*  run_c : S * E * C * D -> value               *)
(*  run_d : S * D -> value                       *)
(*  run_t : Source.term * S * E * C * D -> value  *)
(*  run_s : S * E * C * D -> value               *)
(*  where S = value list                         *)
(*        E = value Env.env                      *)
(*        C = directive list                     *)
(*        D = (S * E * C) list                   *)
fun run_c (s, e, nil, d)
    = run_d (s, d)
  | run_c (s, e, (TERM t) :: c, d)
    = run_t (t, s, e, c, d)
  | run_c (s, e, APPLY :: c, d)
    = run_s (s, e, c, d)
and run_d (v :: nil, nil)
    = v
  | run_d (v :: nil, (s, e, c) :: d)
    = run_c (v :: s, e, c, d)
and run_t (LIT n, s, e, c, d)
    = run_c ((INT n) :: s, e, c, d)
  | run_t (VAR x, s, e, c, d)
    = run_c ((Env.lookup (x, e)) :: s, e, c, d)
  | run_t (LAM (x, t), s, e, c, d)
    = run_c ((CLOSURE (e, x, t)) :: s, e, c, d)
  | run_t (APP (t0, t1), s, e, c, d)
    = run_t (t1, s, e, (TERM t0) :: APPLY :: c, d)
and run_s (SUCC :: (INT n) :: s, e, c, d)
    = run_c ((INT (n+1)) :: s, e, c, d)
  | run_s ((CLOSURE (e', x, t)) :: v :: s, e, c, d)
    = run_t (t, nil, Env.extend (x, v, e'), nil, (s, e, c) :: d)

(*  evaluate1 : Source.program -> value  *)
fun evaluate1 t
    = run_t (t, nil, e_init, nil, nil)
```

Proposition 1 (full correctness). *For any ML value* `t : Source.program`*,*

$$\texttt{evaluate1 t} \;\cong\; \texttt{evaluate0 t}$$

Proof. By fold-unfold [4]. The invariants are as follows. For any ML values `s : stack`, `e : environment`, `c : control`, `d : dump`, and `t : Source.term`,

$$\begin{cases} \texttt{run\_c (s, e, c, d)} \cong \texttt{run (s, e, c, d)} \\ \texttt{run\_d (s, d)} \cong \texttt{run (s, e, nil, d)} \\ \texttt{run\_t (t, s, e, c, d)} \cong \texttt{run (s, e, (TERM t) :: c, d)} \\ \texttt{run\_s (s, e, c, d)} \cong \texttt{run (s, e, APPLY :: c, d)} \end{cases}$$

□

4 A higher-order counterpart of the SECD machine

In the disentangled definition, there are two possible ways to construct a dump (nil and cons) and three possible ways to construct a list of control directives (nil, cons'ing a term, and cons'ing an apply directive). (We could phrase these constructions as two data types rather than as two lists.)

These data types, together with run_d and run_c, are in the image of defunctionalization (run_d and run_c are the apply functions of these two data types). The corresponding higher-order evaluator reads as follows.

```
(*  run_t : Source.term * S * E * C * D -> value  *)
(*  run_s : S * E * C * D -> value                *)
(*  where S = value list                          *)
(*        E = value Env.env                       *)
(*        C = (S * E * D) -> value                *)
(*        D = S -> value                          *)
fun run_t (LIT n, s, e, c, d)
    = c ((INT n) :: s, e, d)
  | run_t (VAR x, s, e, c, d)
    = c ((Env.lookup (x, e)) :: s, e, d)
  | run_t (LAM (x, t), s, e, c, d)
    = c ((CLOSURE (e, x, t)) :: s, e, d)
  | run_t (APP (t0, t1), s, e, c, d)
    = run_t (t1, s, e,
             fn (s, e, d) => run_t (t0, s, e,
                                    fn (s, e, d) => run_s (s, e, c, d),
                                    d),
             d)
and run_s (SUCC :: (INT n) :: s, e, c, d)
    = c ((INT (n+1)) :: s, e, d)
  | run_s ((CLOSURE (e', x, t)) :: v :: s, e, c, d)
    = run_t (t, nil, Env.extend (x, v, e'),
             fn (s, _, d) => d s,
             fn (v :: nil) => c (v :: s, e, d))

(*  evaluate2 : Source.program -> value  *)
fun evaluate2 t
    = run_t (t, nil, e_init,
             fn (s, _, d) => d s,
             fn (v :: nil) => v)
```

The resulting evaluator is in CPS, with two nested continuations, one for the dump, and one for evaluating a root expression (i.e., the top-level expression or the body of a λ-abstraction). It inherits the characteristics of the SECD machine, i.e., it threads a stack of intermediate results, an environment, a control continuation, and a dump continuation. As a lambda-interpreter, it is a bit unusual in that:

1. it has two continuations,

2. it threads a stack of intermediate results, and
3. the environment is saved by the recursive callees, not by the callers. (Usually, the environment is not threaded but saved across recursive calls.)

Otherwise the interpreter follows the traditional eval/apply schema identified by McCarthy in his definition of Lisp in Lisp [16], by Reynolds in his definitional interpreters [19], and by Steele and Sussman in their lambda-papers [22]: `run_t` is eval and `run_s` is apply.

Proposition 2 (full correctness). *For any ML value* `p : Source.program`*,*

$$\texttt{evaluate2 p} \cong \texttt{evaluate1 p}.$$

Proof. Defunctionalizing `evaluate2` yields `evaluate1`, and defunctionalization has been proved correct [1, 18].

5 Back to direct style

The evaluator of Section 4 is in continuation-passing style and therefore it is in the image of the CPS transformation [5]. Its direct-style counterpart reads as follows, renaming `run_t` as `eval` and `run_s` as `apply`.

```
(*   eval : Source.term * S * E * C -> stack  *)
(*  apply : S * E * C -> S                    *)
(*  where S = value list                      *)
(*        E = value Env.env                   *)
(*        C = S * E -> S                      *)
fun eval (LIT n, s, e, c)
    = c ((INT n) :: s, e)
  | eval (VAR x, s, e, c)
    = c ((Env.lookup (x, e)) :: s, e)
  | eval (LAM (x, t), s, e, c)
    = c ((CLOSURE (e, x, t)) :: s, e)
  | eval (APP (t0, t1), s, e, c)
    = eval (t1, s, e, fn (s, e) =>
      eval (t0, s, e, fn (s, e) =>
      apply (s, e, c)))
and apply (SUCC :: (INT n) :: s, e, c)
    = c ((INT (n+1)) :: s, e)
  | apply ((CLOSURE (e', x, t)) :: v :: s, e, c)
    = let val (v :: nil) = eval (t, nil, Env.extend (x, v, e'),
                                 fn (s, _) => s)
      in c (v :: s, e)
      end

(*  evaluate3 : Source.program -> value  *)
fun evaluate3 t
    = let val (v :: nil) = eval (t, nil, e_init, fn (s, _) => s)
      in v
      end
```

Proposition 3 (full correctness). *For any ML value* `p : Source.program`*,*

`evaluate3 p` $\cong$ `evaluate2 p`.

Proof. CPS-transforming `evaluate2` yields `evaluate3`, and the CPS transformation is meaning-preserving.

6 Back to direct style (continued)

Except for two, all the calls to `eval` are tail calls, in the evaluator of Section 5. Except for these two calls, the evaluator is in CPS. These two calls are characteristic of control delimiters [6, 9].

Using the syntactic sugar `<<e>>` as a control delimiter for `e`,[1] the direct-style counterpart of the evaluator reads as follows:

```
(*   eval : Source.term * S * E -> S * E  *)
(*  apply : S * E -> S * E                *)
(*  where S = value list                  *)
(*        E = value Env.env               *)
fun eval (LIT n, s, e)
    = ((INT n) :: s, e)
  | eval (VAR x, s, e)
    = ((Env.lookup (x, e)) :: s, e)
  | eval (LAM (x, t), s, e)
    = ((CLOSURE (e, x, t)) :: s, e)
  | eval (APP (t0, t1), s, e)
    = let val (s, e) = eval (t1, s, e)
          val (s, e) = eval (t0, s, e)
      in apply (s, e)
      end
and apply (SUCC :: (INT n) :: s, e)
    = ((INT (n+1)) :: s, e)
  | apply ((CLOSURE (e', x, t)) :: v :: s, e)
    = let val (v :: nil, _) = <<eval (t, nil, Env.extend (x, v, e'))>>
      in (v :: s, e)
      end

(*  evaluate4 : Source.program -> value  *)
fun evaluate4 t
    = let val (v :: nil, _) = <<eval (t, nil, e_init)>>
      in v
      end
```

Proposition 4 (full correctness). *For any ML value* `p : Source.program`*,*

`evaluate4 p` $\cong$ `evaluate3 p`.

Proof. CPS-transforming `evaluate3` yields `evaluate4`, and the CPS transformation is meaning-preserving.

[1] `<<e>>` is actually written `reset (fn () => e)` where `reset` is an ML function [11].

7 Compositionality, defunctionalization, and closures

The lambda-interpreters of Sections 4, 5, and 6 represent function closures as introduced by Landin [15]. In Section 1.3, this representation was epitomized by the definition of values:

```
datatype value = INT of int
               | SUCC
               | CLOSURE of value Env.env * Source.ide * Source.term
```

A function closure pairs a source λ-abstraction and the environment of its declaration.

Because of this representation, none of the lambda-interpreters above are compositional in the sense of denotational semantics [20, 23, 25]. To be compositional, they should solely define the meaning of each term as a composition of the meaning of its parts. Yet if we view Landin's closures as the result of defunctionalization (with one data constructor and the apply function inlined in `apply`, as in Reynolds's original article on definitional interpreters [19]), we obtain a lambda-interpreter that *is* compositional. This interpreter embodies the denotational content of the SECD machine, be it in direct style, in continuation-passing style, or with two levels of continuations.

Starting from the lambda-interpreter of Section 6, currying it to emphasize that it maps a term to its meaning, and ignoring the control delimiter since no delimited continuation is ever captured, the denotational counterpart of the SECD machine therefore reads as follows:

```
datatype value = INT of int
               | SUCC
               | FUN of value -> value list * value Env.env

(*   eval : Source.term -> S * E -> S * E  *)
(*  apply : S * E -> S * E                 *)
(*  where S = value list                   *)
(*        E = value Env.env                *)
fun eval (LIT n)
    = (fn (s, e) => ((INT n) :: s, e))
  | eval (VAR x)
    = (fn (s, e) => ((Env.lookup (x, e)) :: s, e))
  | eval (LAM (x, t))
    = (fn (s, e) =>
       ((FUN (fn v => eval t (nil, Env.extend (x, v, e)))) :: s, e))
  | eval (APP (t0, t1))
    = apply o (eval t0) o (eval t1)
and apply (SUCC :: (INT n) :: s, e)
    = ((INT (n+1)) :: s, e)
  | apply ((CLOSURE f) :: v :: s, e)
    = let val (v :: nil, _) = f v
      in (v :: s, e)
      end
```

```
(*  evaluate5 : Source.program -> value  *)
fun evaluate5 t
    = let val (v :: nil, _) = eval t (nil, e_init)
      in v
      end
```

The denotational content of the SECD machine is therefore that the meaning of an expression is the mapping $S \times E \rightharpoonup S \times E$ where S is the domain of stacks of intermediate results and E is the domain of environments. Evaluating an ill-typed term is undefined (i.e., in ML, raises a pattern-matching error). Evaluating a well-typed but divergent term diverges. Evaluating a well-typed and convergent term converges to a value.

One can also observe that defunctionalizing the function space of a lambda-interpreter leads one to deep closures (i.e., closures pointing to the current branch of the environment tree), whereas defunctionalizing the function space of a normal program leads one to flat closures (i.e., closures pointing to a minimal copy of the values of the variables occurring free in a lambda-abstraction).

8 Assessment

8.1 From abstract machines to lambda-interpreters

None of the steps from the SECD machine to either of the corresponding lambda-interpreters is tied to the particular architecture of the SECD machine. Therefore, these derivation steps can be applied to any variant of the SECD machine, e.g., properly tail-recursive abstract machines, abstract machines with a single control stack (i.e., without dump), and abstract machines that thread environments in the conventional caller-save fashion.

For example, in his famous 700 follow-up work [17], Morris presents a "shorter equivalent" of the SECD machine as an interpreter written in an applicative language. We note, though, that while Morris's interpreter is definitely shorter, it is not strictly equivalent to the SECD machine. (For example, its environment is saved by the callers, not by the callees.) Indeed, defunctionalizing the CPS counterpart of Morris's interpreter yields a different abstract machine. For example, this abstract machine has one control stack and no dump. (In fact, it coincides with Felleisen's CEK abstract machine [8, 10].)

8.2 From lambda-interpreters to abstract machines

All of the steps from the SECD machine to the corresponding lambda-interpreter are reversible. Therefore these derivation steps can be applied to any lambda-interpreter. For example, since there is no need for first-class continuations to evaluate a λ-term, the control delimiter of Sections 6 and 7 can be dispensed with and the resulting abstract machine has one control stack and no dump. As has been discussed in the literature [24], the dual existence of the control and dump components in the SECD machine has led Landin to a slightly complicated control operator (the J-operator). Unifying these two components leads one to the traditional escape and call/cc control operators.

In retrospect, the overall methodology of going from a lambda-interpreter to an abstract machine is obvious: one CPS-transforms the interpreter to make it tail-recursive, i.e., iterative, and one defunctionalizes the result to make it first order, thereby obtaining a finite-state, iterative abstract machine. The overall derivation is something that could come straight out of a textbook such as *Essentials of Programming Languages* [12], except that this textbook does not use off-the-shelf defunctionalization.

9 Conclusion

We have presented a methodology to relate lambda-interpreters and abstract machines. The contributions of this article can be summarized as follows:

- a more structured specification of the SECD machine;
- a new application of defunctionalization;
- a lambda-revelation of the SECD machine, i.e., a revelation of its denotational content;
- a methodology for inter-converting lambda-interpreters and abstract machines;
- informal applications of this methodology (e.g., for variants of the SECD machine and for Felleisen's CEK machine); and finally,
- a new example of control delimiters in programming practice.

The methodology directly applies to λ-calculi extended with computational effects à la Moggi, e.g., control and state.

References

1. Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, number 2215 in Lecture Notes in Computer Science, Sendai, Japan, October 2001. Springer-Verlag.
2. Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, Amsterdam, The Netherlands, June 1997. ACM Press.
3. Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.
4. Rod M. Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.
5. Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.
6. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
7. Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press.

8. Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.
9. Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.
10. Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`, 1989-2003.
11. Andrzej Filinski. Representing monads. In Boehm [3], pages 446–457.
12. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition.* The MIT Press, 2001.
13. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [3], pages 458–471.
14. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.
15. Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
16. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* The MIT Press, Cambridge, Massachusetts, 1962.
17. Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993.
18. Lasse R. Nielsen. A denotational investigation of defunctionalization. Technical Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2000.
19. John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.
20. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.
21. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.
22. Guy L. Steele Jr. and Gerald J. Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). AI Memo 453, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.
23. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.
24. Hayo Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.
25. Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundation of Computing Series. The MIT Press, 1993.

Recent BRICS Report Series Publications

RS-02-53 Olivier Danvy. *A Lambda-Revelation of the SECD Machine*. December 2003. 15 pp.

RS-02-52 Olivier Danvy. *A New One-Pass Transformation into Monadic Normal Form*. December 2002. 16 pp. Appears in Hedin, editor, *Compiler Construction, 12th International Conference*, CC '03 Proceedings, LNCS 2622, 2003, pages 77–89.

RS-02-51 Gerth Stølting Brodal, Rolf Fagerberg, Anna Östlin, Christian N. S. Pedersen, and S. Srinivasa Rao. *Computing Refined Buneman Trees in Cubic Time*. December 2002. 14 pp.

RS-02-50 Kristoffer Arnsfelt Hansen, Peter Bro Miltersen, and V. Vinay. *Circuits on Cylinders*. December 2002. 16 pp.

RS-02-49 Mikkel Nygaard and Glynn Winskel. *HOPLA—A Higher-Order Process Language*. December 2002. 18 pp. Appears in Brim, Jančar, Křetínský and Antonín, editors, *Concurrency Theory: 13th International Conference*, CONCUR '02 Proceedings, LNCS 2421, 2002, pages 434–448.

RS-02-48 Mikkel Nygaard and Glynn Winskel. *Linearity in Process Languages*. December 2002. 27 pp. Appears in Plotkin, editor, *Seventeenth Annual IEEE Symposium on Logic in Computer Science*, Lics '02 Proceedings, 2002, pages 433–446.

RS-02-47 Zoltán Ésik. *Extended Temporal Logic on Finite Words and Wreath Product of Monoids with Distinguished Generators*. December 2002. 16 pp. To appear in *6th International Conference, Developments in Language Theory*, DLT '02 Revised Papers, LNCS, 2002.

RS-02-46 Zoltán Ésik and Hans Leiß. *Greibach Normal Form in Algebraically Complete Semirings*. December 2002. 43 pp. An extended abstract appears in Bradfield, editor, *European Association for Computer Science Logic: 16th International Workshop*, CSL '02 Proceedings, LNCS 2471, 2002, pages 135–150.

RS-02-45 Jesper Makholm Byskov. *Chromatic Number in Time $O(2.4023^n)$ Using Maximal Independent Sets*. December 2002. 6 pp.