BRICS RS-03-13 Ager et al.: A Functional Correspondence between Evaluators and Abstract Machines

BRICS

Basic Research in Computer Science

A Functional Correspondence between Evaluators and Abstract Machines

Mads Sig Ager
Dariusz Biernacki
Olivier Danvy
Jan Midtgaard

BRICS Report Series **RS-03-13**

ISSN 0909-0878 **March 2003**

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

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/03/13/`

A Functional Correspondence between Evaluators and Abstract Machines

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard
BRICS*
Department of Computer Science
University of Aarhus†

March 2003

Abstract

We bridge the gap between functional evaluators and abstract machines for the λ-calculus, using closure conversion, transformation into continuation-passing style, and defunctionalization of continuations.

We illustrate this bridge by deriving Krivine's abstract machine from an ordinary call-by-name evaluator and by deriving an ordinary call-by-value evaluator from Felleisen et al.'s CEK machine. The first derivation is strikingly simpler than what can be found in the literature. The second one is new. Together, they show that Krivine's abstract machine and the CEK machine correspond to the call-by-name and call-by-value facets of an ordinary evaluator for the λ-calculus.

We then reveal the denotational content of Hannan and Miller's CLS machine and of Landin's SECD machine. We formally compare the corresponding evaluators and we illustrate some relative degrees of freedom in the design spaces of evaluators and of abstract machines for the λ-calculus with computational effects.

For the purpose of this work, we distinguish between virtual machines, which have an instruction set, and abstract machines, which do not. The Categorical Abstract Machine, for example, has an instruction set, but Krivine's machine, the CEK machine, the CLS machine, and the SECD machine do not; they directly operate on λ-terms instead. We present the abstract machine that corresponds to the Categorical Abstract Machine.

*Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
Email: {`mads,dabi,danvy,jmi`}`@brics.dk`

Contents

1 Introduction and related work

In Hannan and Miller's words [21, Section 7], there are fundamental differences between denotational definitions and definitions of abstract machines. While a functional programmer tends to be familiar with denotational definitions [34], he typically wonders about the following issues:

- How does one design an abstract machine? How were existing abstract machines, starting with Landin's SECD machine, designed? How does one make variants of an existing abstract machine? How does one extend an existing abstract machine to a bigger source language? How does one go about designing a new abstract machine? How does one relate two abstract machines?

- How does one prove the correctness of an abstract machine? Assuming it implements a reduction strategy, should one prove that each of its transitions implements a part of this strategy? Or should one characterize it in reference to a given evaluator, or to another abstract machine?

- Why do some abstract machines operate on λ-terms directly whereas others operate on compiled λ-terms expressed with an instruction set?

A variety of answers to these questions can be found in the literature. Landin invented the SECD machine as an implementation model for functional languages [25]. Plotkin proved its correctness in connection with an evaluation function [29, Section 2]. Krivine discovered an abstract machine from a logical standpoint [24]. Crégut proved its correctness in reference to a reduction strategy and he generalized it from weak to strong normalization [6]. Curien discovered the Categorical Abstract Machine from a categorical standpoint [5, 7]. Felleisen et al. invented the CEK machine from an operational standpoint [16]. Hannan and Miller discovered the CLS machine from a proof-theoretical standpoint [21]. Many people derived, invented, or re-discovered Krivine's machine. Many others proposed modifications of existing machines. And recently, Hardin, Maranget, and Pagano introduced a method to extract the reduction strategy of a machine by extracting axioms from its transitions and structural rules from its architecture [22].

In this article, we propose one simple answer to all the questions above. We present a correspondence between functional evaluators and abstract machines based on a two-way derivation: closure conversion, transformation into continuation-passing style (CPS), and defunctionalization. This two-way derivation lets us connect each of the machines above with an evaluator, and makes it possible to echo variations over the evaluator into variations over the abstract machine, and vice versa. The evaluator puts the reduction strategy of the machine in the open. The abstract machine makes the evaluation steps explicit in a transition system. In addition, we also distinguish between abstract machines and virtual machines in the sense that virtual machines have an instruction set and abstract machines do not; instead, they directly operate on source terms and do not need a compiler.

Prerequisites: We use ML as a meta-language, and we assume a basic familiarity with Standard ML and reasoning about ML programs. In particular, given two pure ML expressions `e` and `e'` we write `e` $\cong$ `e'` to express that `e` and `e'` are observationally equivalent. Most of our implementations of the abstract machines raise compiler warnings about non-exhaustive matches. These are inherent to programming abstract machines in an ML-like language. The warnings could be avoided with an option type or with an explicit exception, at the price of readability and direct relation to the usual mathematical specifications of abstract machines.

It would be helpful to the reader to know at least one of the machines considered in the rest of this article, be it Krivine's machine, the CEK machine, the CLS machine, the SECD machine, or the Categorical Abstract Machine. It would also be helpful to have already seen a λ-interpreter written in a functional language [19, 30, 33, 37].

We make use of the CPS transformation [11, 31]: a term is CPS-transformed by naming all its intermediate results, sequentializing their computation, and introducing continuations. Plotkin was the first to establish the correctness of the CPS transformation [29].

We also make use of Reynolds's defunctionalization [30]: defunctionalizing a program amounts to replacing each of its function spaces by a data type and an apply function; the data type enumerates all the function abstractions that may give rise to inhabitants of this function space [14]. In particular, closure conversion amounts to replacing each of the function spaces in expressible and denotable values by a tuple, and inlining the corresponding apply function. Nielsen, Banerjee, Heintze, and Riecke have established the correctness of defunctionalization [2, 28].

Overview: The rest of this article is organized as follows. We first consider a call-by-name and a call-by-value evaluator, and we present the corresponding machines, which are Krivine's machine and the CEK machine. We then consider the CLS machine and the SECD machine, and we present the corresponding evaluators. We finally turn to the Categorical Abstract Machine. For simplicity, we do not cover laziness and sharing, but they come for free by CPS transformation and threading of a heap of updateable thunks.

2 Call-by-name, call-by-value, and the λ-calculus

We first go from a call-by-name evaluator to Krivine's abstract machine (Section 2.1) and then from the CEK machine to a call-by-value evaluator (Section 2.2). The derivation steps consist of closure conversion, transformation into continuation-passing style, and defunctionalization of continuations.

Krivine's abstract machine operates on de Bruijn-encoded λ-terms, and the CEK machine operates on λ-terms with names. Starting from the corresponding evaluators, it is simple to construct a version of Krivine's abstract machine that operates on λ-terms with names, and a version of the CEK machine that operates on de Bruijn-encoded λ-terms (Section 2.3).

2.1 From a call-by-name evaluator to Krivine's machine

Krivine's abstract machine [6] operates on de Bruijn-encoded λ-terms. In this representation, identifiers are represented by their lexical offset, as traditional since Algol 60 [38].

```
datatype term = IND of int    (* de Bruijn index *)
              | ABS of term
              | APP of term * term
```

Programs are closed terms.

2.1.1 A higher-order and compositional call-by-name evaluator

Our starting point is the canonical call-by-name evaluator for the λ-calculus [33, 35]. This evaluator is compositional in the sense of denotational semantics [32, 35, 39] and higher order. It is compositional because it solely defines the meaning of each term as a composition of the meaning of its parts. It is higher order because the data types `denval` and `expval` contain functions. Denotable values are thunks and expressible values are functions [36]. Environments are represented as lists of denotable values. A program is evaluated in an empty environment.

```
structure Eval0
= struct
    datatype denval = THUNK of unit -> expval
         and expval = FUNCT of denval -> expval

    (*  eval : term * denval list -> expval  *)
    fun eval (IND n, e)
        = let val (THUNK thunk) = List.nth (e, n)
          in thunk ()
          end
      | eval (ABS t, e)
        = FUNCT (fn v => eval (t, v :: e))
      | eval (APP (t0, t1), e)
        = let val (FUNCT f) = eval (t0, e)
          in f (THUNK (fn () => eval (t1, e)))
          end

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil)
  end
```

2.1.2 From higher-order functions to closures

In `Eval0`, the function spaces in the data types of denotable and expressible values are only inhabited by instances of the λ-abstractions `fn v => eval (t,` `v :: e)` in the meaning of abstractions, and `fn () => eval (t1, e)` in the meaning of applications. Each of these λ-abstractions has two free variables: a term and an environment. We defunctionalize these function spaces into closures [14, 25, 30], and we inline the corresponding apply functions.

```
structure Eval1
= struct
    datatype denval = THUNK of term * denval list
         and expval = FUNCT of term * denval list

    (*  eval : term * denval list -> expval  *)
    fun eval (IND n, e)
        = let val (THUNK (t, e')) = List.nth (e, n)
          in eval (t, e')
          end
      | eval (ABS t, e)
        = FUNCT (t, e)
      | eval (APP (t0, t1), e)
        = let val (FUNCT (t, e')) = eval (t0, e)
          in eval (t, (THUNK (t1, e)) :: e')
          end

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil)
  end
```

The definition of an abstraction is now `Eval1.FUNCT (t, e)` instead of `fn v => Eval0.eval (t, v :: e)`, and its use is now `Eval1.eval (t, (Eval1.THUNK (t1, e)) :: e')` instead of `f (Eval0.THUNK (fn () => Eval0.eval (t1, e)))`. Similarly, the definition of a thunk is now `Eval1.THUNK (t1, e)` instead of `Eval0.THUNK (fn () => Eval0.eval (t1, e))` and its use is `Eval1.eval (t, e')` instead of `thunk ()`.

The following proposition is a corollary of the correctness of defunctionalization.

Proposition 1 (full correctness) *For any ML value* `p : term` *denoting a program, evaluating* `Eval0.main p` *yields a value* `FUNCT f` *and evaluating* `Eval1.main p` *yields a value* `FUNCT (t, e)` *such that*

$$\texttt{f} \;\cong\; \texttt{fn v => Eval1.eval (t, v :: e)}$$

2.1.3 CPS transformation

We transform `eval` into continuation-passing style.[1] Doing so makes it tail recursive.

[1] Since programs are closed, applying `List.nth` cannot fail and therefore it denotes a total function. We thus keep it in direct style [13].

```
structure Eval2
= struct
    datatype denval = THUNK of term * denval list
         and expval = FUNCT of term * denval list

    (*  eval : term * denval list * (expval -> 'a) -> 'a  *)
    fun eval (IND n, e, k)
        = let val (THUNK (t, e')) = List.nth (e, n)
          in eval (t, e', k)
          end
      | eval (ABS t, e, k)
        = k (FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, fn (FUNCT (t, e'))
                       => eval (t, (THUNK (t1, e)) :: e', k))

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil, fn v => v)
  end
```

The following proposition is a corollary of the correctness of the CPS transformation. (Here observational equivalence reduces to structural equality over ML values of type `expval`.)

Proposition 2 (full correctness) *For any ML value* `p : term` *denoting a program,*

$$\texttt{Eval1.main p} \;\cong\; \texttt{Eval2.main p}$$

2.1.4 Defunctionalizing the continuations

The function space of the continuation is inhabited by instances of two λ-abstractions: the initial one in the definition of `Eval2.main`, with no free variables, and one in the meaning of an application, with three free variables. To defunctionalize the continuation, we thus define a data type `cont` with two summands and the corresponding `apply_cont` function to interpret these summands.

```
structure Eval3
= struct
    datatype denval = THUNK of term * denval list
         and expval = FUNCT of term * denval list
         and   cont = CONT0 | CONT1 of term * denval list * cont

    (*  eval : term * denval list * cont -> expval  *)
    fun eval (IND n, e, k)
        = let val (THUNK (t, e')) = List.nth (e, n)
          in eval (t, e', k)
          end
      | eval (ABS t, e, k)
        = apply_cont (k, FUNCT (t, e))
```

```
    | eval (APP (t0, t1), e, k)
      = eval (t0, e, CONT1 (t1, e, k))
  and apply_cont (CONT0, v)
      = v
    | apply_cont (CONT1 (t1, e, k), FUNCT (t, e'))
      = eval (t, (THUNK (t1, e)) :: e', k)

  (*  main : term -> expval  *)
   fun main t
       = eval (t, nil, CONT0)
end
```

The following proposition is a corollary of the correctness of defunctionalization. (Again, observational equivalence reduces here to structural equality over ML values of type `expval`.)

Proposition 3 (full correctness) *For any ML value* `p : term` *denoting a program,*

$$\texttt{Eval2.main p} \cong \texttt{Eval3.main p}$$

We identify that `cont` is a stack of thunks, and that the transitions are those of Krivine's abstract machine.

2.1.5 Krivine's abstract machine

To obtain the canonical definition of Krivine's abstract machine, we abandon the distinction between denotable and expressible values and we use thunks instead, we represent the defunctionalized continuation as a list of thunks instead of a data type, and we inline `apply_cont`.

```
structure Eval4
= struct
    datatype thunk = THUNK of term * thunk list

    (*  eval : term * thunk list * thunk list -> term * thunk list  *)
    fun eval (IND n, e, s)
        = let val (THUNK (t, e')) = List.nth (e, n)
          in eval (t, e', s)
          end
      | eval (ABS t, e, nil)
        = (ABS t, e)
      | eval (ABS t, e, (t', e') :: s)
        = eval (t, (THUNK (t', e')) :: e, s)
      | eval (APP (t0, t1), e, s)
        = eval (t0, e, (t1, e) :: s)

     (*  main : term -> term * thunk list  *)
     fun main t
         = eval (t, nil, nil)
  end
```

The following proposition is straightforward to prove.

Proposition 4 (full correctness) *For any ML value* `p : term` *denoting a program,*

$$\texttt{Eval3.main p} \;\cong\; \texttt{Eval4.main p}$$

For comparison, the canonical definition of Krivine's abstract machine is as follows [6, 20, 24], where t denotes terms, v denotes expressible values, e denotes environments, and s denotes stacks of expressible values:

- Source syntax:
$$t \;::=\; n \mid \lambda t \mid t_0\, t_1$$

- Expressible values (closures):
$$v \;::=\; [t,\, e]$$

- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t,\, nil,\, nil\rangle$
$\langle n,\, e,\, s\rangle$	$\Rightarrow$	$\langle t,\, e',\, s\rangle,\ where\ [t,\, e'] = nth(e,\, n)$
$\langle \lambda t,\, e,\, [t',\, e'] :: s\rangle$	$\Rightarrow$	$\langle t,\, [t',\, e'] :: e,\, s\rangle$
$\langle t_0\, t_1,\, e,\, s\rangle$	$\Rightarrow$	$\langle t_0,\, e,\, [t_1,\, e] :: s\rangle$
$\langle \lambda t,\, e,\, nil\rangle$	$\Rightarrow$	$[t,\, e]$

Variables n are represented by their de Bruijn index, and the abstract machine operates on triples consisting of a term, an environment, and a stack of expressible values.

Each line in the canonical definition matches a clause in `Eval4`. We conclude that Krivine's abstract machine can be seen as a defunctionalized, CPS-transformed, and closure-converted version of the standard call-by-name evaluator for the λ-calculus. This evaluator evidently implements Hardin, Maranget, and Pagano's K strategy [22, Section 3].

2.2 From the CEK machine to a call-by-value evaluator

The CEK machine [15, 16] operates on λ-terms with names and distinguishes between values and computations in their syntax (i.e., it distinguishes trivial and serious terms, in Reynolds's words [30]).

```
datatype term = VALUE of value
              | COMP of comp
    and value = VAR of string    (* name *)
              | LAM of string * term
    and  comp = APP of term * term
```

Programs are closed terms.

2.2.1 The CEK abstract machine

Our starting point reads as follows [18, Figure 2, page 239], where t denotes terms, w denotes values, v denotes expressible values, k denotes evaluation contexts, e denotes environments, and s denotes stacks of expressible values:

- Source syntax:
$$\begin{array}{rcl} t & ::= & w \mid t_0\, t_1 \\ w & ::= & x \mid \lambda x.t \end{array}$$

- Expressible values (closures) and evaluation contexts:
$$\begin{array}{rcl} v & ::= & [x,\, t,\, e] \\ k & ::= & \mathtt{stop} \mid \mathtt{fun}(v, k) \mid \mathtt{arg}(t, e, k) \end{array}$$

- Initial transition, transition rules (two kinds), and final transition:

t	$\Rightarrow_{eval}$	$\langle t,\, mt,\, \mathtt{stop}\rangle$
$\langle w,\, e,\, k\rangle$	$\Rightarrow_{eval}$	$\langle k,\, \gamma(w, e)\rangle$
$\langle t_0\, t_1,\, e,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0,\, e,\, \mathtt{arg}(t_1, e, k)\rangle$
$\langle \mathtt{arg}(t_1, e, k),\, v\rangle$	$\Rightarrow_{cont}$	$\langle t_1,\, e,\, \mathtt{fun}(v, k)\rangle$
$\langle \mathtt{fun}([x,\, t,\, e], k),\, v\rangle$	$\Rightarrow_{cont}$	$\langle t,\, e[x \mapsto v],\, k\rangle$
$\langle \mathtt{stop},\, v\rangle$	$\Rightarrow_{cont}$	v

where
$$\begin{array}{rcl} \gamma(x, e) & = & e(x) \\ \gamma(\lambda x.t, e) & = & [x,\, t,\, e] \end{array}$$

Variables x are represented by their name, and the abstract machine consists of two mutually recursive transition functions. The first transition function operates on triples consisting of a term, an environment, and an evaluation context. The second operates on pairs consisting of an evaluation context and an expressible value. Environments are extended in the `fun`-transition, and consulted in γ. The empty environment is denoted by mt.

This specification is straightforward to program in ML:

```
signature ENV
= sig
    type 'a env
    val mt : 'a env
    val lookup : 'a env * string -> 'a
    val extend : string * 'a * 'a env -> 'a env
  end
```

Environments are represented as a structure `Env : ENV` containing a representation of the empty environment `mt`, an operation `lookup` to retrieve the value bound to a name in an environment, and an operation `Env.extend` to extend an environment with a binding.

```
structure Eval0
= struct
    datatype expval = CLOSURE of string * term * expval Env.env
    datatype evaluation_context = STOP
             | ARG of term * expval Env.env * evaluation_context
             | FUN of expval * evaluation_context

    (*  eval : term * expval Env.env * evaluation_context -> expval  *)
    fun eval (VALUE v, e, k)
        = continue (k, eval_value (v, e))
      | eval (COMP (APP (t0, t1)), e, k)
        = eval (t0, e, ARG (t1, e, k))
    and eval_value (VAR x, e)
        = Env.lookup (e, x)
      | eval_value (LAM (x, t), e)
        = CLOSURE (x, t, e)
    and continue (STOP, w)
        = w
      | continue (ARG (t1, e, k), w)
        = eval (t1, e, FUN (w, k))
      | continue (FUN (CLOSURE (x, t, e), k), w)
        = eval (t, Env.extend (x, w, e), k)

    (*  main : term -> expval  *)
    fun main t
        = eval (t, Env.mt, STOP)
  end
```

2.2.2 Refunctionalizing the evaluation contexts into continuations

We identify that the data type `evaluation_context` and the function `continue` are a defunctionalized representation. The corresponding higher-order evaluator reads as follows. As can be observed, it is in continuation-passing style.

```
structure Eval1
= struct
    datatype expval = CLOSURE of string * term * expval Env.env

    (*  eval : term * expval Env.env * (expval -> 'a) -> 'a  *)
    fun eval (VALUE v, e, k)
        = k (eval_value (v, e))
      | eval (COMP (APP (t0, t1)), e, k)
        = eval (t0,
                e,
                fn (CLOSURE (x, t, e'))
                => eval (t1,
                         e,
                         fn w
                         => eval (t,
                                  Env.extend (x, w, e),
                                  k)))
```

```
  and eval_value (VAR x, e)
      = Env.lookup (e, x)
    | eval_value (LAM (x, t), e)
      = CLOSURE (x, t, e)

  (*  main : term -> expval  *)
  fun main t
      = eval (t, Env.mt, fn w => w)
end
```

The following proposition is a corollary of the correctness of defunctionalization. (Observational equivalence reduces here to structural equality over ML values of type `expval`.)

Proposition 5 (full correctness) *For any ML value* `p : term` *denoting a program,*

$$\texttt{Eval0.main p} \cong \texttt{Eval1.main p}$$

2.2.3 Back to direct style

CPS-transforming the following direct-style evaluator yields the evaluator of Section 2.2.2 [9].

```
structure Eval2
= struct
    datatype expval = CLOSURE of string * term * expval Env.env

    (*  eval : term * expval Env.env -> expval  *)
    fun eval (VALUE v, e)
        = eval_value (v, e)
      | eval (COMP (APP (t0, t1)), e)
        = let val (CLOSURE (x, t, e')) = eval (t0, e)
              val w = eval (t1, e)
          in eval (t, Env.extend (x, w, e))
          end
    and eval_value (VAR x, e)
        = Env.lookup (e, x)
      | eval_value (LAM (x, t), e)
        = CLOSURE (x, t, e)

    (*  main : term -> expval  *)
    fun main t
        = eval (t, Env.mt)
  end
```

The following proposition is a corollary of the correctness of the direct-style transformation. (Again, observational equivalence reduces here to structural equality over ML values of type `expval`.)

Proposition 6 (full correctness) *For any ML value* `p : term` *denoting a program,*

```
Eval1.main p  ≅  Eval2.main p
```

2.2.4 From closures to higher-order functions

We observe that the closures, in `Eval2`, are defunctionalized representations with an apply function inlined. The corresponding higher-order evaluator reads as follows.

```
structure Eval3
= struct
    datatype expval = CLOSURE of expval -> expval

    (*  eval : term * expval Env.env -> expval  *)
    fun eval (VALUE v, e)
        = eval_value (v, e)
      | eval (COMP (APP (t0, t1)), e)
        = let val (CLOSURE f) = eval (t0, e)
              val w = eval (t1, e)
          in f w
          end
    and eval_value (VAR x, e)
        = Env.lookup (e, x)
      | eval_value (LAM (x, t), e)
        = CLOSURE (fn w => eval (t, Env.extend (x, w, e)))

    (*  main : term -> expval  *)
    fun main t
        = eval (t, Env.mt)
  end
```

The following proposition is a corollary of the correctness of defunctionalization.

Proposition 7 (full correctness) *For any ML value* `p : term` *denoting a program, evaluating* `Eval2.main p` *yields a value* `CLOSURE (x, t, e)` *and evaluating* `Eval3.main p` *yields a value* `CLOSURE f` *such that*

```
fn w => Eval2.eval (t, Env.extend (x, w, e))  ≅  f
```

2.2.5 A higher-order and compositional call-by-value evaluator

The result in `Eval3` is a call-by-value evaluator that is compositional and higher-order. This call-by-value evaluator is the canonical one for the λ-calculus [33, 35]. We conclude that the CEK machine can be seen as a defunctionalized, CPS-transformed, and closure-converted version of the standard call-by-value evaluator for λ-terms.

2.3 Variants of Krivine's machine and of the CEK machine

It is easy to construct a variant of Krivine's abstract machine for λ-terms with names, by starting from a call-by-name evaluator for λ-terms with names. Similarly, it is easy to construct a variant of the CEK machine for λ-terms with de Bruijn indices, by starting from a call-by-value evaluator for λ-terms with indices. It is equally easy to start from a call-by-value evaluator for λ-terms with de Bruijn indices and no distinction between values and computations; the resulting abstract machine coincides with Hankin's eager machine [20, Section 8.1.2].

Abstract machines processing λ-terms with de Bruijn indices often resolve indices with transitions:

$$\begin{array}{rcl} \langle 0,\, v :: e,\, s\rangle & \Rightarrow & v :: s \\ \langle n+1,\, v :: e,\, s\rangle & \Rightarrow & \langle n,\, e,\, s\rangle \end{array}$$

Compared to the evaluator of Section 2.1.1, page 5, the evaluator corresponding to this machine has `List.nth` inlined and is not compositional:

```
fun eval (IND 0, denval :: e, s)
    = ... denval ...
  | eval (IND n, denval :: e, s)
    = eval (IND (n - 1), e, s)
  | ...
```

2.4 Conclusion

We have shown that Krivine's abstract machine and the CEK abstract machine are counterparts of canonical evaluators for call-by-name and for call-by-value λ-terms, respectively. The derivation of Krivine's machine is strikingly simpler than what can be found in the literature. That the CEK machine can be derived is, to the best of our knowledge, new. That these two machines are two sides of the same coin is also new. We have not explored any other aspect of this call-by-name/call-by-value duality [8].

Using substitutions instead of environments or inlining one of the standard computational monads (state, continuations, etc. [37]) in the call-by-value evaluator yields variants of the CEK machine that have been documented in the literature [15, Chapter 8]. For example, inlining the state monad in a monadic evaluator yields a state-passing evaluator. The corresponding abstract machine has one more component to represent the state. In general, inlining monads provides a generic recipe to construct arbitrarily many new abstract machines. It does not seem as straightforward, however, to construct a "monadic abstract machine" and then to inline a monad; we are currently studying the question.

On another note, one can consider an evaluator for strictness-annotated λ-terms—represented either with names or with indices, and with or without distinction between values and computations. One is then led to an abstract machine that generalizes Krivine's machine and the CEK machine [12].

Finally, it is straightforward to extend Krivine's machine and the CEK machine to bigger source languages (with literals, primitive operations, conditional expressions, block structure, recursion, etc.), by starting from evaluators for these bigger languages. For example, all the abstract machines in "The essence of compiling with continuations" [18] are defunctionalized continuation-passing evaluators, i.e., interpreters.

In the rest of this article, we illustrate further the correspondence between evaluators and abstract machines.

3 The CLS abstract machine

The CLS abstract machine is due to Hannan and Miller [21]. In the following, t denotes terms, v denotes expressible values, c denotes lists of directives (a term or the special tag `ap`), e denotes environments, l denotes stacks of environments, and s denotes stacks of expressible values.

- Source syntax:

$$t \quad ::= \quad n \;\mid\; \lambda t \;\mid\; t_0\, t_1$$

- Expressible values (closures):

$$v \quad ::= \quad [t,\, e]$$

- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t :: nil,\, nil :: nil,\, nil\rangle$
$\langle \lambda t :: c,\, e :: l,\, s\rangle$	$\Rightarrow$	$\langle c,\, l,\, [t,\, e] :: s\rangle$
$\langle (t_0\, t_1) :: c,\, e :: l,\, s\rangle$	$\Rightarrow$	$\langle t_0 :: t_1 :: \mathtt{ap} :: c,\, e :: e :: l,\, s\rangle$
$\langle 0 :: c,\, (v :: e) :: l,\, s\rangle$	$\Rightarrow$	$\langle c,\, l,\, v :: s\rangle$
$\langle n+1 :: c,\, (v :: e) :: l,\, s\rangle$	$\Rightarrow$	$\langle n :: c,\, e :: l,\, s\rangle$
$\langle \mathtt{ap} :: c,\, l,\, v :: [t,\, e] :: s\rangle$	$\Rightarrow$	$\langle t :: c,\, (v :: e) :: l,\, s\rangle$
$\langle nil,\, nil,\, v :: s\rangle$	$\Rightarrow$	v

Variables n are represented by their de Bruijn index, and the abstract machine operates on triples consisting of a list of directives, a stack of environments, and a stack of expressible values.

3.1 The CLS machine

Hannan and Miller's specification is straightforward to program in ML:

```
datatype term = IND of int    (* de Bruijn index *)
              | ABS of term
              | APP of term * term
```

Programs are closed terms.

```
structure Eval0
= struct
    datatype directive = TERM of term
                       | AP
    datatype env = ENV of expval list
      and expval = CLOSURE of term * env

    (*  run : directive list * env list * expval list -> expval  *)
    fun run (nil, nil, v :: s)
        = v
      | run ((TERM (IND 0)) :: c, (ENV (v :: e)) :: l, s)
        = run (c, l, v :: s)
      | run ((TERM (IND n)) :: c, (ENV (v :: e)) :: l, s)
        = run ((TERM (IND (n - 1))) :: c, (ENV e) :: l, s)
      | run ((TERM (ABS t)) :: c, e :: l, s)
        = run (c, l, (CLOSURE (t, e)) :: s)
      | run ((TERM (APP (t0, t1))) :: c, e :: l, s)
        = run ((TERM t0) :: (TERM t1) :: AP :: c, e :: e :: l, s)
      | run (AP :: c, l, v :: (CLOSURE (t, ENV e)) :: s)
        = run ((TERM t) :: c, (ENV (v :: e)) :: l, s)

    (*  main : term -> expval  *)
    fun main t
        = run ((TERM t) :: nil, (ENV nil) :: nil, nil)
  end
```

3.2 A disentangled definition of the CLS machine

In the definition of Section 3.1, 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, the computation terminates. If the list starts with a term, `run_t` is called, caching the term in the first parameter. If the list starts with an apply directive, `run_a` is called.

- `run_t` interprets the top term in the list of control directives.

- `run_a` interprets the top value in the current stack.

The disentangled definition reads as follows:

```
structure Eval1
= struct
    datatype directive = TERM of term
                       | AP
```

```
  datatype env = ENV of expval list
    and expval = CLOSURE of term * env

  (*  run_c : directive list * env list * expval list -> expval  *)
  fun run_c (nil, nil, v :: s)
      = v
    | run_c ((TERM t) :: c, l, s)
      = run_t (t, c, l, s)
    | run_c (AP :: c, l, s)
      = run_a (c, l, s)
  and run_t (IND 0, c, (ENV (v :: e)) :: l, s)
      = run_c (c, l, v :: s)
    | run_t (IND n, c, (ENV (v :: e)) :: l, s)
      = run_t (IND (n - 1), c, (ENV e) :: l, s)
    | run_t (ABS t, c, e :: l, s)
      = run_c (c, l, (CLOSURE (t, e)) :: s)
    | run_t (APP (t0, t1), c, e :: l, s)
      = run_t (t0, (TERM t1) :: AP :: c, e :: e :: l, s)
  and run_a (c, l, v :: (CLOSURE (t, ENV e)) :: s)
      = run_t (t, c, (ENV (v :: e)) :: l, s)

  (*  main : term -> expval  *)
  fun main t
      = run_t (t, nil, (ENV nil) :: nil, nil)
end
```

Proposition 8 (full correctness) *For any ML value* `p : term` *denoting a program,*

$$\texttt{Eval0.main p} \;\cong\; \texttt{Eval1.main p}$$

Proof: By fold-unfold [4]. The invariants are as follows. For any ML values `t : term`, `e : expval list`, and `s : expval list`,

$$\left\{\begin{array}{rcl}
\texttt{Eval1.run_c (c, l, s)} & \cong & \texttt{Eval0.run (c, l, s)} \\
\texttt{Eval1.run_t (t, c, l, s)} & \cong & \texttt{Eval0.run ((TERM t) :: c, l, s)} \\
\texttt{Eval1.run_a (c, l, s)} & \cong & \texttt{Eval0.run (AP :: c, l, s)}
\end{array}\right.$$

□

3.3 The evaluator corresponding to the CLS machine

In the disentangled definition of Section 3.2, there are three possible ways to construct a list of control directives (nil, cons'ing a term, and cons'ing an apply directive). We could specify these constructions as a data type rather than as a list. Such a data type, together with `run_c`, is in the image of defunctionalization (`run_c` is the apply functions of the data type). The corresponding higher-order evaluator is in continuation-passing style. Transforming it back to direct style yields the following evaluator:

```
structure Eval3
= struct
    datatype env = ENV of expval list
      and expval = CLOSURE of term * env

    (*  run_t : term * env list * expval list -> env list * expval list  *)
    fun run_t (IND 0, (ENV (v :: e)) :: l, s)
        = (l, v :: s)
      | run_t (IND n, (ENV (v :: e)) :: l, s)
        = run_t (IND (n - 1), (ENV e) :: l, s)
      | run_t (ABS t, e :: l, s)
        = (l, (CLOSURE (t, e)) :: s)
      | run_t (APP (t0, t1), e :: l, s)
        = let val (l, s) = run_t (t0, e :: e :: l, s)
              val (l, s) = run_t (t1, l, s)
          in run_a (l, s)
          end
    and run_a (l, v :: (CLOSURE (t, ENV e)) :: s)
        = run_t (t, (ENV (v :: e)) :: l, s)

    (*  main : term -> expval  *)
    fun main t
        = let val (nil, v :: s) = run_t (t, (ENV nil) :: nil, nil)
          in v
          end
  end
```

The following proposition is a corollary of the correctness of defunctionalization and of the CPS transformation. (Here observational equivalence reduces to structural equality over ML values of type `expval`.)

Proposition 9 (full correctness) *For any ML value* `p : term` *denoting a program,*

$$\texttt{Eval1.main p} \cong \texttt{Eval3.main p}$$

As in Section 2, this evaluator can be made compositional by refunctionalizing the closures into higher-order functions and by factoring the resolution of de Bruijn indices into an auxiliary lookup function.

We conclude that the evaluation model embodied in the CLS machine is a call-by-value interpreter threading a stack of environments and a stack of intermediate results with a caller-save strategy (witness the duplication of environments on the stack in the meaning of applications) and with a left-to-right evaluation of sub-terms. In particular, the meaning of a term is a partial endofunction over a stack of environments and a stack of intermediate results.

4 The SECD abstract machine

The SECD abstract machine is due to Landin [25]. In the following, t denotes terms, v denotes expressible values, c denotes lists of directives (a term or the special tag ap), e denotes environments, s denotes stacks of expressible values, and d denotes dumps (list of triples consisting of a stack, an environment and a list of directives).

- Source syntax:
$$t \quad ::= \quad x \;\mid\; \lambda x.t \;\mid\; t_0\, t_1$$
- Expressible values (closures):
$$v \quad ::= \quad [x,\, t,\, e]$$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle nil,\, mt,\, t :: nil,\, nil\rangle$
$\langle s,\, e,\, x :: c,\, d\rangle$	$\Rightarrow$	$\langle e(x) :: s,\, e,\, c,\, d\rangle$
$\langle s,\, e,\, (\lambda x.t) :: c,\, d\rangle$	$\Rightarrow$	$\langle [x,\, t,\, e] :: s,\, e,\, c,\, d\rangle$
$\langle s,\, e,\, (t_0\, t_1) :: c,\, d\rangle$	$\Rightarrow$	$\langle s,\, e,\, t_1 :: t_0 :: \texttt{ap} :: c,\, d\rangle$
$\langle [x,\, t,\, e'] :: v :: s,\, e,\, \texttt{ap} :: c,\, d\rangle$	$\Rightarrow$	$\langle nil,\, e'[x \mapsto v],\, t :: nil,\, (s, e, c) :: d\rangle$
$\langle v :: s,\, e,\, nil,\, (s', e', d') :: d\rangle$	$\Rightarrow$	$\langle v :: s',\, e',\, c',\, d\rangle$
$\langle v :: s,\, e,\, nil,\, nil\rangle$	$\Rightarrow$	v

Variables x are represented by their name, and the abstract machine operates on quadruples consisting of a stack of expressible values, an environment, a list of directives, and a dump. Environments are consulted in the first transition rule, and extended in the fourth. The empty environment is denoted by mt.

4.1 The SECD machine

Landin's specification is straightforward to program in ML. Programs are closed terms. Environments are as in Section 2.2.

```
datatype term = VAR of string    (* name *)
              | LAM of string * term
              | APP of term * term

structure Eval0
= struct
    datatype directive = TERM of term
                       | AP

    datatype value = CLOSURE of string * term * value Env.env

    fun run (v :: nil, e', nil, nil)
        = v
      | run (s, e, (TERM (VAR x)) :: c, d)
        = run ((Env.lookup (e, x)) :: s, e, c, d)
      | run (s, e, (TERM (LAM (x, t))) :: c, d)
        = run ((CLOSURE (x, t, e)) :: s, e, c, d)
      | run (s, e, (TERM (APP (t0, t1))) :: c, d)
        = run (s, e, (TERM t1) :: (TERM t0) :: AP :: c, d)
```

```
    | run ((CLOSURE (x, t, e')) :: v :: s, e, AP :: c, d)
      = run (nil, Env.extend (x, v, e'), (TERM t) :: nil, (s, e, c) :: d)
    | run (v :: nil, e', nil, (s, e, c) :: d)
      = run (v :: s, e, c, d)

  (*  main : term -> value  *)
  fun main t
      = run (nil, Env.mt, (TERM t) :: nil, nil)
end
```

4.2 A disentangled definition of the SECD machine

As in the CLS machine, in the definition of Section 4.1, all the possible transitions are meshed together in one recursive function, `run`. Instead, we can factor `run` into several mutually recursive functions, each of them with one induction variable. These mutually recursive functions are in defunctionalized form: the one processing the dump is an apply function for the data type representing the dump (a list of stacks, environments, and lists of directives), and the one processing the control is an apply function for the data type representing the control (a list of directives). The corresponding higher-order evaluator is in continuation-passing style with two nested continuations and one control delimiter [11, 17]. The delimiter resets the control continuation when evaluating the body of a λ-abstraction. (More detail is available in a technical report [10].)

4.3 The evaluator corresponding to the SECD machine

The direct-style version of the evaluator from Section 4.2 reads as follows:

```
structure Eval4
= struct
    datatype value = CLOSURE of string * term * value Env.env

    (*  eval : term * value list * value Env.env  *)
    (*             -> value list * value Env.env  *)
    fun eval (VAR x, s, e)
        = ((Env.lookup (x, e)) :: s, e)
      | eval (LAM (x, t), s, e)
        = ((CLOSURE (x, t, e)) :: 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 ((CLOSURE (x, t, e')) :: v :: s, e)
        = let val (v :: nil, _)
                  = reset (fn ()
                               => eval (t, nil, Env.extend (x, v, e')))
          in (v :: s, e)
          end
```

```
(*  main : term -> value  *)
fun main t
    = let val (v :: nil, _)
              = reset (fn ()
                       => eval (t, nil, Env.mt))
      in v
      end
end
```

The following proposition is a corollary of the correctness of defunctionalization and of the CPS transformation. (Here observational equivalence reduces to structural equality over ML values of type `value`.)

Proposition 10 (full correctness) *For any ML value* `p : term` *denoting a program,*

$$\texttt{Eval0.main p} \cong \texttt{Eval4.main p}$$

As in Sections 2 and 3, this evaluator can be made compositional by refunctionalizing the closures into higher-order functions.

We conclude that the evaluation model embodied in the SECD machine is a call-by-value interpreter threading a stack of intermediate results and an environment with a callee-save strategy (witness the dynamic passage of environments in the meaning of applications), a right-to-left evaluation of sub-terms, and a control delimiter. In particular, the meaning of a term is a partial endofunction over a stack of intermediate results and an environment. Furthermore, this evaluator evidently implements Hardin, Maranget, and Pagano's L strategy, i.e., right-to-left call by value, without us having to "guess" its inference rules [22, Section 4].

The denotational content of the SECD machine puts a new light on it. For example, its separation between a control register and a dump register is explained by the control delimiter in the evaluator. Removing this control delimiter gives rise to an abstract machine with a single stack component for control—not by a clever change in the machine itself, but by a straightforward simplification in the corresponding evaluator.

5 Variants of the CLS machine and of the SECD machine

It is straightforward to construct a variant of the CLS machine for λ-terms with names, by starting from an evaluator for λ-term with names. Similarly, it is straightforward to construct a variant of the SECD machine for λ-terms with de Bruijn indices, by starting from an evaluator for λ-term with indices. In the same vein, it is simple to construct call-by-name versions of the CLS machine and of the SECD machine, by starting from call-by-name evaluators. It is also simple to construct a properly tail recursive version of the SECD machine, and to extend the CLS machine and the SECD machine to bigger source languages, by extending the corresponding evaluator.

6 The Categorical Abstract Machine

What is the difference between an abstract machine and a virtual machine? In a companion article [1], we propose to distinguish them based on the notion of instruction set: A virtual machine has an instruction set whereas an abstract machine does not. An abstract machine directly operates on a λ-term, but a virtual machine operates on a compiled representation of a λ-term, expressed using an instruction set.

The Categorical Abstract Machine [5], for example, has an instruction set—categorical combinators—and therefore (despite its name) it is a virtual machine, not an abstract machine. In contrast, Krivine's machine, the CEK machine, the CLS machine, and the SECD machine are all abstract machines, not virtual machines, since they directly operate on λ-terms. In this section, we present the abstract machine corresponding to the Categorical Abstract Machine (CAM). We start from the evaluation model embodied in the CAM, as obtained in the companion article.

6.1 The evaluator corresponding to the CAM

The evaluation model embodied in the CAM is an interpreter threading a stack with its top element cached in a register, representing environments as expressible values (namely nested pairs linked as lists), with a caller-save strategy (witness the duplication of the register on the stack in the meaning of applications below), and with a left-to-right evaluation of sub-terms. In particular, the meaning of a term is a partial endofunction over the register and the stack. This evaluator reads as follows:

```
datatype term = IND of int    (* de Bruijn index *)
              | ABS of term
              | APP of term * term
              | NIL
              | CONS of term * term
              | CAR of term
              | CDR of term
```

Programs are closed terms.

```
structure Eval0
= struct
    datatype expval = NULL
                    | PAIR of expval * expval
                    | CLOSURE of expval * (expval * expval list ->
                                           expval * expval list)

    (*  access : int * expval * 'a -> expval * 'a  *)
    fun access (0, PAIR (v1, v2), s)
        = (v2, s)
      | access (n, PAIR (v1, v2), s)
        = access (n - 1, v1, s)
```

```
    (*  eval : term * expval * expval list -> expval * expval list  *)
    fun eval (IND n, v, s)
        = access (n, v, s)
      | eval (ABS t, v, s)
        = (CLOSURE (v, fn (v, s) => eval (t, v, s)), s)
      | eval (APP (t0, t1), v, s)
        = let val (v, v' :: s) = eval (t0, v, v :: s)
              val (v', (CLOSURE (v, f)) :: s) = eval (t1, v', v :: s)
          in f (PAIR (v, v'), s)
          end
      | eval (NIL, v, s)
        = (NULL, s)
      | eval (CONS (t1, t2), v, s)
        = let val (v, v' :: s) = eval (t1, v, v :: s)
              val (v, v' :: s) = eval (t2, v', v :: s)
          in (PAIR (v', v), s)
          end
      | eval (CAR t, v, s)
        = let val (PAIR (v1, v2), s) = eval (t, v, s)
          in (v1, s)
          end
      | eval (CDR t, v, s)
        = let val (PAIR (v1, v2), s) = eval (t, v, s)
          in (v2, s)
          end

    (*  main : term -> expval  *)
    fun main t
        = let val (v, nil) = eval (t, NULL, nil)
          in v
          end
  end
```

This evaluator evidently implements Hardin, Maranget, and Pagano's X strategy [22, Section 6].

6.2 The abstract machine corresponding to the CAM

As in Sections 2, 3, and 4, we can closure-convert the evaluator of Section 6.1 by defunctionalizing its expressible values, transform it into continuation-passing style, and defunctionalize its continuations. The resulting abstract machine reads as follows, where t denotes terms, v denotes expressible values, k denotes evaluation contexts, and s denotes stacks of expressible values.

- Source syntax:

$$t \quad ::= \quad n \;|\; \lambda t \;|\; t_0\, t_1 \;|\; \texttt{nil} \;|\; (\texttt{cons}\, t_1\, t_2) \;|\; (\texttt{car}\, t) \;|\; (\texttt{cdr}\, t)$$

- Expressible values (unit value, pairs, and closures) and evaluation contexts:

$$\begin{array}{rcl} v & ::= & \mathtt{null} \mid (v_1,\, v_2) \mid [v,\, t] \\ k & ::= & \mathtt{CONT0} \mid \mathtt{CONT1}(t,\, k) \mid \mathtt{CONT2}(k) \mid \mathtt{CONT3}(t,\, k) \mid \\ & & \mathtt{CONT4}(k) \mid \mathtt{CONT5}(k) \mid \mathtt{CONT6}(k) \end{array}$$

- Initial transition, transition rules (two kinds), and final transition:

t	$\Rightarrow_{eval}$	$\langle t,\, \mathtt{null},\, nil,\, \mathtt{CONT0}\rangle$
$\langle n,\, v,\, s,\, k\rangle$	$\Rightarrow_{eval}$	$\langle k,\, \gamma(n,v),\, s\rangle$
$\langle \lambda t,\, v,\, s,\, k\rangle$	$\Rightarrow_{eval}$	$\langle k,\, [v,\, t],\, s\rangle$
$\langle \mathtt{nil},\, v,\, s,\, k\rangle$	$\Rightarrow_{eval}$	$\langle k,\, \mathtt{null},\, s\rangle$
$\langle t_0\, t_1,\, v,\, s,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0,\, v,\, v :: s,\, \mathtt{CONT1}(t_1,\, k)\rangle$
$\langle (\mathtt{cons}\, t_1\, t_2),\, v,\, s,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t_1,\, v,\, v :: s,\, \mathtt{CONT3}(t_2,\, k)\rangle$
$\langle (\mathtt{car}\, t),\, v,\, s,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t,\, v,\, s,\, \mathtt{CONT5}(k)\rangle$
$\langle (\mathtt{cdr}\, t),\, v,\, s,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t,\, v,\, s,\, \mathtt{CONT6}(k)\rangle$
$\langle \mathtt{CONT1}(t,\, k),\, v,\, v' :: s\rangle$	$\Rightarrow_{cont}$	$\langle t,\, v',\, v :: s,\, \mathtt{CONT2}(k)\rangle$
$\langle \mathtt{CONT2}(k),\, v',\, [v,\, t] :: s\rangle$	$\Rightarrow_{cont}$	$\langle t,\, (v,\, v'),\, s,\, k\rangle$
$\langle \mathtt{CONT3}(t_1,\, k),\, v,\, v' :: s\rangle$	$\Rightarrow_{cont}$	$\langle t_1,\, v',\, v :: s,\, \mathtt{CONT4}(k)\rangle$
$\langle \mathtt{CONT4}(k),\, v,\, v' :: s\rangle$	$\Rightarrow_{cont}$	$\langle k,\, (v',\, v),\, s\rangle$
$\langle \mathtt{CONT5}(k),\, (v_1,\, v_2),\, s\rangle$	$\Rightarrow_{cont}$	$\langle k,\, v_1,\, s\rangle$
$\langle \mathtt{CONT6}(k),\, (v_1,\, v_2),\, s\rangle$	$\Rightarrow_{cont}$	$\langle k,\, v_2,\, s\rangle$
$\langle \mathtt{CONT0},\, v,\, nil\rangle$	$\Rightarrow_{cont}$	v

where $$\begin{array}{rcl} \gamma(0,(v_1,\, v_2)) & = & v_2 \\ \gamma(n,(v_1,\, v_2)) & = & \gamma(n-1,v_1) \end{array}$$

Variables n are represented by their de Bruijn index, and the abstract machine consists of two mutually recursive transition functions. The first transition function operates on quadruples consisting of a term, an expressible value, a stack of expressible values, and an evaluation context. The second transition function operates on triples consisting of an evaluation context, an expressible value, and a stack of expressible values.

This abstract machine embodies the evaluation model of the CAM. Naturally, more intuitive names could be chosen instead of `CONT0`, `CONT1`, etc.

7 Conclusion and issues

We have presented a constructive, mechanical, and generic correspondence between functional evaluators and abstract machines. This correspondence builds on off-the-shelf program transformations: closure conversion, CPS transformation, defunctionalization, and inlining.[2] We have shown how to reconstruct known machines (Krivine's machine, the CEK machine, the CLS machine, and the SECD machine) and how to construct new ones. Conversely, we have revealed the denotational content of known abstract machines. We have shown that Krivine's abstract machine and the CEK machine correspond to canonical evaluators for the λ-calculus. We have also shown that they are dual of each other since they correspond to call-by-name and call-by-value evaluators in the same direct style.[3] In terms of denotational semantics [26, 32], Krivine's machine and the CEK machine correspond to a standard semantics, whereas the CLS machine and the SECD machine correspond to a stack semantics of the λ-calculus. Finally, we have exhibited the abstract machine corresponding to the CAM, which puts the reader in a new position to answer the recurrent question as to whether the CLS machine is closer to the CAM or to the SECD machine.

It seems to us that this correspondence between functional evaluators and abstract machines builds a reliable bridge between denotational definitions and definitions of abstract machines. On the one hand, it allows one to identify the denotational content of an abstract machine in the form of a functional interpreter. On the other hand, it gives one a precise and generic recipe to construct arbitrarily many new variants of abstract machines (e.g., with substitutions or environments, or with stacks) or of arbitrarily many new abstract machines, starting from an evaluator with any given computational monad [27].

Acknowledgments: We are grateful to Małgorzata Biernacka and Henning Korsholm Rohde for timely comments.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2003.

[2] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki

[2] Indeed the push-enter twist of Krivine's machine is obtained by inlining `apply_cont` in Section 2.1.5.

[3] In fact, one could say that they correspond to the *same* evaluator in the sense that after closure conversion, Krivine's machine is obtained by call-by-name CPS transformation and the CEK machine is obtained by call-by-value CPS transformation [23]. We elaborate on this point in the companion article [1].

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.

[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] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[6] Pierre Crégut. An abstract machine for lambda-terms normalization. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 333–340, Nice, France, June 1990. ACM Press.

[7] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Progress in Theoretical Computer Science. Birkhaüser, 1993.

[8] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In Philip Wadler, editor, *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 35, No. 9, pages 233–243, Montréal, Canada, September 2000. ACM Press.

[9] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[10] Olivier Danvy. A lambda-revelation of the SECD machine. Technical Report BRICS RS-02-53, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2002.

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

[12] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[13] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 627–648, New Orleans, Louisiana, April 1993. Springer-Verlag.

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

[15] 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.

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

[17] Andrzej Filinski. Representing monads. In Boehm [3], pages 446–457.

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[19] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition*. The MIT Press, 2001.

[20] Chris Hankin. *Lambda Calculi, a guide for computer scientists*, volume 1 of *Graduate Texts in Computer Science*. Oxford University Press, 1994.

[21] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[22] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[23] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [3], pages 458–471.

[24] Jean-Louis Krivine. Un interprète du λ-calcul. Brouillon. Available online at `http://www.logique.jussieu.fr/~krivine`, 1985.

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

[26] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, and John Wiley, New York, 1976.

[27] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[28] 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.

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

[30] 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).

[31] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[32] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[33] 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.

[34] Joseph Stoy. Some mathematical aspects of functional programming. In John Darlington, Peter Henderson, and David A. Turner, editors, *Functional Programming and its Applications*. Cambridge University Press, 1982.

[35] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[36] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):1–49, 2000.

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

[38] Mitchell Wand. A short proof of the lexical addressing algorithm. *Information Processing Letters*, 35:1–5, 1990.

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

Recent BRICS Report Series Publications

RS-03-13 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. ***A Functional Correspondence between Evaluators and Abstract Machines*****. March 2003. 28 pp.**

RS-03-12 Mircea-Dan Hernest and Ulrich Kohlenbach. ***A Complexity Analysis of Functional Interpretations*****. February 2003. 70 pp.**

RS-03-11 Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. ***Fast Partial Evaluation of Pattern Matching in Strings*****. February 2003. 14 pp.**

RS-03-10 Federico Crazzolara and Giuseppe Milicia. ***Wireless Authentication in χ-Spaces*****. February 2003. 20 pp.**

RS-03-9 Ivan B. Damgård and Gudmund Skovbjerg Frandsen. ***An Extended Quadratic Frobenius Primality Test with Average and Worst Case Error Estimates*****. February 2003. 53 pp.**

RS-03-8 Ivan B. Damgård and Gudmund Skovbjerg Frandsen. ***Efficient Algorithms for gcd and Cubic Residuosity in the Ring of Eisenstein Integers*****. February 2003. 11 pp.**

RS-03-7 Claus Brabrand, Michael I. Schwartzbach, and Mads Vanggaard. ***The METAFRONT System: Extensible Parsing and Transformation*****. February 2003. 24 pp.**

RS-03-6 Giuseppe Milicia and Vladimiro Sassone. ***Jeeg: Temporal Constraints for the Synchronization of Concurrent Objects*****. February 2003. 41 pp. Short version appears in Fox and Getov, editors,** ***Joint ACM-ISCOPE Conference on Java Grande*****, JGI '02 Proceedings, 2002, pages 212–221.**

RS-03-5 Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. ***Precise Analysis of String Expressions*****. February 2003. 15 pp.**

RS-03-4 Marco Carbone and Mogens Nielsen. ***Towards a Formal Model for Trust*****. January 2003.**

RS-03-3 Claude Crépeau, Paul Dumais, Dominic Mayers, and Louis Salvail. ***On the Computational Collapse of Quantum Information*****. January 2003. 31 pp.**