From Interpreter to Compiler and Virtual Machine: A Functional Derivation

Mads Sig Ager
Dariusz Biernacki
Olivier Danvy
Jan Midtgaard

BRICS Report Series RS-03-14

ISSN 0909-0878 March 2003

From Interpreter to Compiler and Virtual Machine: a Functional Derivation

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

March 2003

Abstract

We show how to derive a compiler and a virtual machine from a compositional interpreter. We first illustrate the derivation with two evaluation functions and two normalization functions. We obtain Krivine's machine, Felleisen et al.'s CEK machine, and a generalization of these machines performing strong normalization, which is new. We observe that several existing compilers and virtual machines—e.g., the Categorical Abstract Machine (CAM), Schmidt's VEC machine, and Leroy's Zinc abstract machine—are already in derived form and we present the corresponding interpreter for the CAM and the VEC machine. We also consider Hannan and Miller's CLS machine and Landin's SECD machine.

We derived Krivine's machine via a call-by-name CPS transformation and the CEK machine via a call-by-value CPS transformation. These two derivations hold both for an evaluation function and for a normalization function. They provide a non-trivial illustration of Reynolds's warning about the evaluation order of a meta-language.

Keywords: evaluation function, normalization function, normalization by evaluation, call by name, call by value, evaluation-order independence, Krivine's machine, CEK machine, CLS machine, SECD machine, Categorical Abstract Machine, VEC machine, Zinc abstract machine, defunctionalization, continuation-passing style.

[*]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

What is the difference between, on the one hand, the Categorical Abstract Machine [11, 14], the VEC machine [50], and the Zinc abstract machine [27, 40], and on the other hand, Krivine's machine [12, 38], the CEK machine [21, 23], the CLS machine [30], and the SECD machine [39]? Their goal is the same—implementing an evaluation function—but the former machines have an instruction set, whereas the latter ones do not; instead, they operate directly on the source λ-term (and thus are more akin to interpreters than to machines, even though they take the form of a transition system). For the purpose of our work, we need to distinguish them. We thus state that the former machines are *virtual machines* (in the sense of the Java Virtual Machine [26]) and the latter ones are *abstract machines*. The former ones require a compiler, and the latter ones do not.[1]

[1]So in that sense, and despite their names, the Categorical Abstract Machine and the Zinc abstract machine are virtual machines.

In a companion article [2], we present a functional correspondence between evaluators and abstract machines, and we exhibit the evaluators that correspond to Krivine's machine, the CEK machine, the CLS machine, and the SECD machine. The two-way derivation between an evaluator and the corresponding abstract machine consists of (1) closure conversion, (2) transformation into continuation-passing style, and (3) defunctionalization of continuations. Each of these steps is traditional but the derivation passes the reality check of relating pre-existing evaluators and pre-existing abstract machines, relating pre-existing evaluators and new abstract machines, and relating new evaluators and pre-existing abstract machines. To the best of our knowledge, passing this check is new. The derivation is also robust in the sense that variations over an evaluator are echoed into variations over an abstract machine and vice versa, which is also new. Our main result is that Krivine's machine and the CEK machine are two sides of the same coin in that they correspond to an ordinary evaluator for the λ-calculus; one uses call by name, and the other call by value. This evaluator implements a standard semantics in the sense of Milne and Strachey [42, 50]. In contrast, the CLS machine and the SECD machine correspond to evaluators that implement a stack semantics.

In the present article, we go one step further by factoring an evaluation function into a compiler and a virtual machine. The results are the virtual-machine counterparts of Krivine's abstract machine, the CEK abstract machine, the CLS abstract machine, and the SECD abstract machine. Our derivation passes the reality check of relating pre-existing interpreters and pre-existing compilers and virtual machines, relating pre-existing interpreters and new compilers and virtual machines, and relating new interpreters and pre-existing compilers and virtual machines. Indeed three of these compilers and virtual machines were known (Krivine, CLS, and SECD), and one is new (CEK). The derivation is also robust because it lets us echo variations over an interpreter into variations over a compiler and virtual machine and vice versa. In addition, it lets us identify that the Categorical Abstract Machine (CAM), the VEC machine, and the Zinc abstract machine are already in derived form; we present the interpreters that correspond to the CAM and the VEC machine. The interpreter corresponding to the CAM is not subsumed by the descriptions and motivations of the CAM available in the literature, and therefore it sheds a new denotational light on the CAM. The interpreter corresponding to the VEC machine is new. In the companion article, we present the abstract machine corresponding to the CAM interpreter.

In the present article, we also consider two normalization functions and we derive the corresponding compilers and virtual machines. These machines are extensions of Krivine's virtual machine and of the CEK virtual machine. The compilers, however, are the same. We interpret this variation as a corollary of Reynolds's observation about the evaluation order of meta-languages [49].

Related work: Methods and methodologies for deriving compilers from interpreters come in a bewildering variety. Automated methodologies include partial evaluation [9, 36], and mechanized ones include combinatory abstraction, either using a fixed set of combinators, be they categorical ones [46] or actions [44], or an ad-hoc set of (super)combinators, as in Hughes's implementation method [35] and in Wand's combinator-based compilers [53, 55]. We observe, however, that interpreters and compilers are still developed independently of each other today, indicating that the methods and methodologies for deriving compilers from interpreters have not prevailed. We also observe that derivations described in the literature only address few examples. Arrays of representative examples somehow are not considered. This is our goal here.

Our methodology for deriving a compiler and a virtual machine from an interpreter uses off-the-shelf program-transformation tools. We use closure conversion for expressible and denotable values, we transform the interpreter into continuation-passing style (CPS), we defunctionalize its continuations, and we factor it into a compositional compiler and a virtual machine. Defunctionalization is due to Reynolds [49], it has been formalized and proved correct [5, 45], and it enjoys many applications [18]; in particular, closure conversion amounts to in-place defunctionalization. The CPS transformation has a long history [48]; it comes in many forms and it also has been proved correct [47]. After closure conversion, CPS transformation, and defunctionalization, we express the evaluator as a composition of combinators and recursive calls to the evaluator. The compiler is obtained by choosing a term model where the combinators are interpreted as their names and composition as sequencing. The virtual machine is an interpreter for the sequence of combinators. This style of derivation was pioneered by Wand [24, 54, 55]. The applications presented in this article, however, are new.

Prerequisites: We use ML as a meta-language, and we assume a basic familiarity with Standard ML, including its module language. (Incidentally, most of our implementations below raise compiler warnings about non-exhaustive matches. These warnings could be avoided with an option type or with an explicit exception, at the price of readability.) We also assume a passing acquaintance with defunctionalization and the CPS transformation as can be gathered in Reynolds's "Definitional Interpreters for Higher-Order Programming Languages" [49]. It would be helpful to the reader to know at least one of the machines considered in the rest of this article, e.g., Krivine's machine or the CEK machine.

Overview: We first derive compilers and virtual machines from evaluation functions (Section 2) and from normalization functions (Section 3). We then turn to interpreters that correspond to existing compilers and virtual machines (Section 4). We also present compilers and virtual machines that correspond to existing abstract machines (Section 5).

2 Virtual machines for evaluation functions

We consider evaluation functions that encode the standard call-by-name and call-by-value semantics of the λ-calculus. In the companion article [2], we show that Krivine's abstract machine corresponds to the evaluation function for call by name, and that the CEK abstract machine corresponds to the evaluation function for call by value. Each of them is our starting point below.

2.1 Source terms

Krivine's abstract machine operates on de Bruijn-encoded λ-terms. For simplicity, we label each λ-abstraction with a unique integer, and we make the machine yield a result pairing a label and the environment. This way, both abstract and virtual machines return the same result. (Otherwise, the abstract machine would yield a result pairing a source λ-abstraction and an environment, and the virtual machine would yield a result pairing a compiled λ-abstraction and an environment.)

```
type label = int

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

Programs are closed terms.

The CEK machine operates on λ-terms with names.

2.2 Call by name

Krivine's abstract machine reads as follows:

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

    (*  eval : term * thunk list * thunk list -> label * 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 (l, t), e, nil)
        = (l, e)
      | eval (ABS (l, t), e, (THUNK (t', e')) :: s)
        = eval (t, (THUNK (t', e')) :: e, s)
      | eval (APP (t0, t1), e, s)
        = eval (t0, e, (THUNK (t1, e)) :: s)

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

This specification is not compositional because it does not solely define the meaning of each term as a composition of the meaning of its parts [50, 51, 56]. Indeed not all calls to `eval`, on the right-hand side, are over proper sub-parts of the terms in the left-hand side.

Let us make `eval` compositional and curry it. We make it compositional using a recursive data type `recur`. We curry it to exemplify its denotational nature of mapping a source term into a (functional) meaning.

```
structure Eval_Krivine_compositional_and_curried
= struct
    datatype thunk = THUNK of recur * thunk list
         and recur = RECUR of thunk list * thunk list -> label * thunk list

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

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

We now factor `eval` into a composition of combinators and recursive calls. The clause for applications, for example, is factored as

```
      | eval (APP (t0, t1))
        = (eval t0) o (push (eval t1))
```

where `push c` denotes

```
     (fn (e, s) => (e, (THUNK (RECUR c, e)) :: s))
```

The clause for abstractions, however, forces us to introduce a data type of intermediate results (see `I_eval` below).

The factored version can be expressed as a functor parameterized with an interpretation:

```
     signature INTERPRETATION
     = sig
         type computation
         type result
```

```
    val access : int -> computation
    val grab : label -> computation
    val push : computation -> computation
    val combine : computation * computation -> computation
    val compute : computation -> result
  end
```

In the following functor, `oo` is an infix operator:

```
functor mkProcessor (structure I : INTERPRETATION)
= struct
    val (op oo) = I.combine

    fun eval (IND n)
        = I.access n
      | eval (ABS (l, t))
        = (I.grab l) oo (eval t)
      | eval (APP (t0, t1))
        = (I.push (eval t1)) oo (eval t0)

    fun main t
        = I.compute (eval t)
  end
```

The following structure implements a standard interpretation (`oo` is defined as reversed function composition). Instantiating `mkProcessor` with this interpretation yields the evaluator above (modulo the intermediate data type).

```
structure I_eval
= struct
    datatype   thunk = THUNK of recur * thunk list
    and        recur = RECUR of intermediate -> intermediate
    and intermediate = GOING of thunk list * thunk list
                     | DONE of label * thunk list

    type computation = intermediate -> intermediate
    type result = label * thunk list

    fun lift f
        = (fn (GOING v)
              => f v
            | d
              => d)

    fun grab l
        = lift (fn (e, nil)
                   => DONE (l, e)
                 | (e, THUNK (c, e') :: s)
                   => GOING ((THUNK (c, e')) :: e, s))
    fun push c
        = lift (fn (e, s) => GOING (e, (THUNK (RECUR c, e)) :: s))

    fun access n
        = lift (fn (e, s) => let val (THUNK (RECUR c, e'))
                                     = List.nth (e, n)
                             in c (GOING (e', s))
                             end)

    fun combine (f, g)
        = g o f

    fun compute c
        = let val (DONE l) = c (GOING (nil, nil))
          in l
          end
  end

structure Eval_Krivine = mkProcessor (structure I = I_eval)
```

Conversely, we can consider a term model where each of `access`, `close`, and `push` is a virtual-machine instruction, `combine` is a sequencing operation, and the types `computation` and `result` are lists of instructions. The corresponding structure `I_compile` implements a non-standard interpretation. Instantiating `mkProcessor` with it yields a compiler. It is then a simple exercise to define the virtual machine as interpreting a list of instructions according to the definition of `access`, `close`, and `push` in the standard interpretation so that the following diagram commutes:

$$\begin{tikzcd} \texttt{mkProcessor} \arrow[r, "\texttt{I\_compile}"] \arrow[dr, "\texttt{I\_eval}"'] & {} \arrow[d, dotted, "\text{virtual machine}"] \\ & {} \end{tikzcd}$$

The resulting compiler and virtual machine read as follows, where t denotes terms, ℓ denotes labels, i denotes instructions, c denotes lists of instructions, and e denotes environments:

- Source and target syntax:

$$\begin{array}{rcl} t & ::= & n \mid \lambda^{\ell} t \mid t_0\, t_1 \\ i & ::= & \texttt{access}\, n \mid \texttt{grab}\, \ell \mid \texttt{push}\, c \end{array}$$

- Compiler:

$$\begin{array}{rcl} [\![n]\!] & = & \texttt{access}\, n \\ [\![\lambda^{\ell} t]\!] & = & \texttt{grab}\, \ell; [\![t]\!] \\ [\![t_0\, t_1]\!] & = & \texttt{push}\, [\![t_1]\!]; [\![t_0]\!] \end{array}$$

- Expressible values (closures) and results:

$$\begin{array}{rcl} v & ::= & [c,\, e] \\ r & ::= & [\ell,\, e] \end{array}$$

- Initial transition, transition rules, and final transition:

c	$\Rightarrow$	$\langle c,\, nil,\, nil\rangle$
$\langle \mathtt{access}\, n; c,\, e,\, s\rangle$	$\Rightarrow$	$\langle c',\, e',\, s\rangle$ *where* $[c',\, e'] = e(n)$
$\langle \mathtt{grab}\, \ell; c,\, e,\, [c',\, e'] :: s\rangle$	$\Rightarrow$	$\langle c,\, [c',\, e'] :: e,\, s\rangle$
$\langle \mathtt{push}\, c'; c,\, e,\, s\rangle$	$\Rightarrow$	$\langle c,\, e,\, [c',\, e] :: s\rangle$
$\langle \mathtt{grab}\, \ell; c,\, e,\, nil\rangle$	$\Rightarrow$	$[\ell,\, e]$

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

Except for the labels, this version of Krivine's machine coincides with the definition of Krivine's machine in Leroy's economical implementation of ML [40]. (The names `access`, `grab`, and `push` originate in this presentation.)

2.3 Call by value

Starting from the CEK abstract machine, the same sequence of steps as in Section 2.2 leads us to the following compiler and virtual machine, where t denotes terms, i denotes instructions, c denotes lists of instructions, e denotes environments, v denotes expressible values, and k denotes evaluation contexts:

- Source and target syntax:

$$\begin{array}{rcl} t & ::= & x \mid \lambda x.t \mid t_0\, t_1 \\ i & ::= & \mathtt{access}\, x \mid \mathtt{close}(x, c) \mid \mathtt{push}\, c \end{array}$$

- Compiler:

$$\begin{array}{rcl} [\![x]\!] & = & \mathtt{access}\, x \\ [\![\lambda x.t]\!] & = & \mathtt{close}(x, [\![t]\!]) \\ [\![t_0\, t_1]\!] & = & \mathtt{push}\, [\![t_1]\!]; [\![t_0]\!] \end{array}$$

- Expressible values (closures) and evaluation contexts:

$$\begin{array}{rcl} v & ::= & [x,\, c,\, e] \\ k & ::= & \mathtt{ECONT0} \mid \mathtt{ECONT1}(c,\, e,\, k) \mid \mathtt{ECONT2}(v,\, k) \end{array}$$

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

c	$\Rightarrow$	$\langle c,\, mt,\, \mathtt{ECONT0}\rangle$
$\langle \mathtt{access}\, x; c,\, e,\, k\rangle$	$\Rightarrow$	$\langle k,\, e(x)\rangle$
$\langle \mathtt{close}(x, c'); c,\, e,\, k\rangle$	$\Rightarrow$	$\langle k,\, [x,\, c',\, e]\rangle$
$\langle \mathtt{push}\, c'; c,\, e,\, k\rangle$	$\Rightarrow$	$\langle c,\, e,\, \mathtt{ECONT1}(c',\, e,\, k)\rangle$
$\langle \mathtt{ECONT1}(c,\, e,\, k),\, v\rangle$	$\Rightarrow$	$\langle c,\, e,\, \mathtt{ECONT2}(v,\, k)\rangle$
$\langle \mathtt{ECONT2}([x,\, c,\, e],\, k),\, v\rangle$	$\Rightarrow$	$\langle c,\, e[x \mapsto v],\, k\rangle$
$\langle \mathtt{ECONT0},\, v\rangle$	$\Rightarrow$	v

Variables x are represented by their name, and the virtual machine consists of two mutually recursive transition functions. The first transition function operates on triples consisting of a list of instructions, an environment, and an evaluation context. The second operates on pairs consisting of an evaluation context and a value.

To the best of our knowledge, this version of the CEK machine is new.

2.4 Call by need

Implementing the thunks of Section 2.2 as memo-thunks, i.e., as functions with a local state to memoize their result, yields a call-by-need version of the evaluation function. To make it purely functional, one can thread a state of updateable thunks in `eval`. CPS transformation and defunctionalization then yield a call-by-need version of Krivine's machine, and factorization yields a compiler and a virtual machine with a state component. We do not go further into detail.

2.5 Summary, related work, and conclusions

We have derived a compiler and a virtual machine from Krivine's abstract machine and from the CEK abstract machine, each of which corresponds to an evaluation function for the λ-calculus. Krivine's compiler and virtual machine were known, and they have been used in Hardin, Maranget, and Pagano's study of functional runtime systems within the $\lambda\sigma$-calculus [31]. To the bext of our knowledge, the CEK compiler and virtual machine are new. We have also outlined how to construct a compiler and virtual machine for call-by-need.

In the companion article, we point out that variants of the CEK machines can easily be constructed by considering variants of the original evaluator (e.g., in state-passing style or again in monadic style together with any monad). The corresponding compilers and virtual machines are equally simple to construct. In fact, we believe that they provide a more fitting model than the corresponding abstract machines for characterizing the essence of compiling with continuations [23]—a future work.

A similar kind of factorization of an evaluator into a core semantics and interpretations can be found in Jones and Nielson's handbook chapter on abstract interpretation [37]. By that book, other interpretations could yield program analyses.

All in all, we have presented two virtual machines performing evaluation. The starting evaluation function itself, however, does not change—what changes is the evaluation order of its meta-language, as captured in the CPS transformation [32]. In that, we capitalize on the interplay between defining language and defined language identified by Reynolds thirty years ago in his seminal article about definitional interpreters [49]. We come back to this point in Section 6.

Our closest related work is Streicher and Reus's derivation of abstract machines from two continuation semantics [52]. Indeed they derive Krivine's machine from a call-by-name semantics and the CEK machine from a call-by-value semantics.

3 Virtual machines for normalization functions

Normalization by evaluation is a reduction-free approach to normalization. Rather than normalizing a term based on reduction rules, one defines a *normalization function* that maps a term into its normal form. The idea is due to Martin Löf, who suggested intuitionistic type theory as a good meta-language for expressing normalization functions [41], and it has been formally investigated by Altenkirch, Coquand, Dybjer, Scott, Streicher, and others [3, 4, 10, 13]. In the early 1990's [8], Berger and Schwichtenberg have struck upon a type-directed specification of normalization functions for the simply typed λ-calculus that corresponds to a normalization proof due to Tait [6]. This type-directed specification naturally arises in offline partial evaluation for functional languages with sums and computational effects [15, 17, 20, 22]. Other normalization functions have also been developed [1, 25, 43].

We observe that normalization functions lend themselves to constructing compilers and virtual machines in the same way as evaluation functions do. The rest of this section illustrates this construction, first with call by name, then with call by value, and finally with call by need.

3.1 Source and residual terms

We first specify source λ-terms with de Bruijn indices and then target terms in β-normal form with names.

```
structure Source
= struct
    datatype term = IND of int
                  | ABS of term
                  | APP of term * term
  end

structure Residual
= struct
    datatype nf = LAM of string * nf
                | VAL of at
         and at = APP of at * nf
                | VAR of string
  end
```

These two specifications enable us to reflect in the type of the normalization function that it maps a source term to a residual term in normal form [19].

As is often done in practice [1, 6, 12, 22, 27], we could have specified residual terms with de Bruijn levels. Doing so would only complicate the normalization functions below. Therefore, for simplicity, we consider residual terms with names and as a result the normalization functions use a generator of fresh names `Gensym.new`. We could also use monads to the same effect [22].

3.2 Call by name

The following ML structure implements a call-by-name version of an untyped normalization function. This version uses thunks to implement call by name [33], as can be seen in the definition of expressible values. This untyped normalization function can be found, for example, in Aehlig and Joachimski's recent work [1], where it is formalized and programmed in Haskell. We program it in ML instead; in addition, we use the specialized data type of λ-terms in normal form rather than a general data type of λ-terms for residual terms.

```
structure NbE_cbn
= struct
    datatype expval = FUN of (unit -> expval) -> expval
                    | RES of Residual.at

    (*  reify : expval -> Residual.nf  *)
    fun reify (FUN f)
        = let val x = Gensym.new "x"
          in Residual.LAM (x, reify (f (fn () => RES (Residual.VAR x))))
          end
      | reify (RES r)
          = Residual.VAL r

    (*  eval : Source.term * (unit -> expval) list -> expval  *)
    fun eval (Source.IND n, vs)
        = List.nth (vs, n) ()
      | eval (Source.ABS t, vs)
        = FUN (fn v => eval (t, v :: vs))
      | eval (Source.APP (t0, t1), vs)
        = (case eval (t0, vs)
             of (FUN f)
                => f (fn () => eval (t1, vs))
              | (RES r)
                => RES (Residual.APP (r, reify (eval (t1, vs)))))

    (*  normalize : Source.term -> Residual.nf  *)
    fun normalize t
        = reify (eval (t, nil))
  end
```

We subject this normalization function to the following sequence of steps: (1) closure conversion of the two function spaces of `expval`; (2) CPS transformation of `eval` and `reify`; (3) defunctionalization of the continuations, inlining the apply function of the continuation of `eval`, and (4) factorization of `eval` into a composition of functions and of recursive calls to `eval`. The resulting compiler and virtual machine read as follow, where t denotes terms, i denotes instructions, c denotes lists of instructions, and e denotes environments:

- Source and target syntax:

$$\begin{array}{rcl} t & ::= & n \mid \lambda t \mid t_0\, t_1 \\ i & ::= & \mathtt{access}\, n \mid \mathtt{grab} \mid \mathtt{push}\, c \end{array}$$

- Compiler:

$$\begin{array}{rcl} [\![n]\!] & = & \mathtt{access}\, n \\ [\![\lambda t]\!] & = & \mathtt{grab};\, [\![t]\!] \\ [\![t_0\, t_1]\!] & = & \mathtt{push}\, [\![t_1]\!];\, [\![t_0]\!] \end{array}$$

- Expressible values, residual terms (using infix $\underline{@}$ to denote application), evaluation contexts, and reification contexts:

$$\begin{array}{rcl} v & ::= & \mathtt{FUN}[c,\ e] \mid \mathtt{RES}[r] \\ r & ::= & x \mid \underline{\lambda} x.r \mid r_0 \,\underline{@}\, r_1 \\ ke & ::= & \mathtt{ECONT0} \mid \mathtt{ECONT1}(x, kr) \mid \mathtt{ECONT2}(r, ke) \mid \mathtt{ECONT3}(c, e, ke) \\ kr & ::= & \mathtt{RCONT0} \mid \mathtt{RCONT1}(x, kr) \mid \mathtt{RCONT2}(r, ke) \end{array}$$

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

c	$\Rightarrow$	$\langle c,\ nil,\ \mathtt{ECONT0}\rangle$
$\langle \mathtt{access}\, n; c,\ e,\ ke\rangle$	$\Rightarrow_{eval}$	$\langle c',\ e',\ ke\rangle, \textit{if } \mathtt{FUN}[c',\ e'] = e(n)$
$\langle \mathtt{access}\, n; c,\ e,\ ke\rangle$	$\Rightarrow_{eval}$	$\langle ke,\ r\rangle,\ \textit{if } \mathtt{RES}[r] = e(n)$
$\langle \mathtt{push}\, c'; c,\ e,\ ke\rangle$	$\Rightarrow_{eval}$	$\langle c,\ e,\ \mathtt{ECONT3}(c', e, ke)\rangle$
$\langle \mathtt{grab}; c,\ e,\ \mathtt{ECONT0}\rangle$	$\Rightarrow_{eval}$	$\langle c,\ \mathtt{RES}[x] :: e,\ \mathtt{ECONT1}(x, \mathtt{RCONT0})\rangle$, *where x is fresh*
$\langle \mathtt{grab}; c,\ e,\ \mathtt{ECONT1}(x, kr)\rangle$	$\Rightarrow_{eval}$	$\langle c,\ \mathtt{RES}[x'] :: e,\ \mathtt{ECONT1}(x', \mathtt{RCONT1}(x', kr))\rangle$, *where x' is fresh*
$\langle \mathtt{grab}; c,\ e,\ \mathtt{ECONT2}(r, ke)\rangle$	$\Rightarrow_{eval}$	$\langle c,\ \mathtt{RES}[x] :: e,\ \mathtt{ECONT1}(x, \mathtt{RCONT2}(r, ke))\rangle$, *where x is fresh*
$\langle \mathtt{grab}; c,\ e,\ \mathtt{ECONT3}(c', e', ke)\rangle$	$\Rightarrow_{eval}$	$\langle c,\ \mathtt{FUN}[c',\ e'] :: e,\ ke\rangle$
$\langle \mathtt{ECONT0},\ r\rangle$	$\Rightarrow_{apply\_ke}$	$\langle \mathtt{RCONT0},\ r\rangle$
$\langle \mathtt{ECONT1}(x, kr),\ r\rangle$	$\Rightarrow_{apply\_ke}$	$\langle \mathtt{RCONT1}(x, kr),\ r\rangle$
$\langle \mathtt{ECONT2}(r', ke),\ r\rangle$	$\Rightarrow_{apply\_ke}$	$\langle \mathtt{RCONT2}(r', ke),\ r\rangle$
$\langle \mathtt{ECONT3}(c', e', ke),\ r\rangle$	$\Rightarrow_{apply\_ke}$	$\langle c',\ e',\ \mathtt{ECONT2}(r, ke)\rangle$
$\langle \mathtt{RCONT1}(x, kr),\ r\rangle$	$\Rightarrow_{apply\_kr}$	$\langle kr,\ \underline{\lambda} x.r\rangle$
$\langle \mathtt{RCONT2}(r', ke),\ r\rangle$	$\Rightarrow_{apply\_kr}$	$\langle ke,\ r' \,\underline{@}\, r\rangle$
$\langle \mathtt{RCONT0},\ r\rangle$	$\Rightarrow_{apply\_kr}$	r

Variables in a source term are represented by their de Bruijn index. Variables in a residual term are represented by their name. The virtual machine consists of three mutually recursive transition functions. The first transition function operates on triples consisting of a list of instructions, an environment, and an evaluation context. The second operates on pairs consisting of an evaluation context and a residual term. The third operates on pairs consisting of a reification context and a residual term.

The instruction set in the target language and the compiler is the same as in Krivine's virtual machine, in Section 2.2, modulo the labels. We conclude that this new virtual machine is a strong-normalization counterpart of Krivine's virtual machine (which performs weak normalization).

3.3 Call by value

The following ML structure implements the call-by-value counterpart of the normalization function of Section 3.2. As can be seen in the definition of expressible values, it does not use thunks.

```
structure NbE_cbv
= struct
    datatype expval = FUN of expval -> expval
                    | RES of Residual.at

    (*  reify : expval -> Residual.nf  *)
    fun reify (FUN f)
        = let val x = Gensym.new "x"
          in Residual.LAM (x, reify (f (RES (Residual.VAR x))))
          end
      | reify (RES r)
        = Residual.VAL r

    (*  eval : Source.term * expval list -> expval  *)
    fun eval (Source.IND n, vs)
        = List.nth (vs, n)
      | eval (Source.ABS t, vs)
        = FUN (fn v => eval (t, v :: vs))
      | eval (Source.APP (t0, t1), vs)
        = (case eval (t0, vs)
             of (FUN f)
                => f (eval (t1, vs))
              | (RES r)
                => RES (Residual.APP (r, reify (eval (t1, vs)))))

    (*  normalize : Source.term -> Residual.nf  *)
    fun normalize t
        = reify (eval (t, nil))
  end
```

We subject this normalization function to the same sequence of steps as in Section 3.2: (1) closure conversion of the function space of `expval`; (2) CPS transformation of `eval` and `reify`; (3) defunctionalization of the continuations, inlining the apply function of the continuation of `eval`, and (4) factorization of `eval` into a composition of functions and of recursive calls to `eval`. The resulting compiler and virtual machine read as follow, where t denotes terms, i denotes instructions, c denotes lists of instructions, and e denotes environments:

- Source and target syntax:

$$\begin{array}{rcl} t & ::= & n \mid \lambda t \mid t_0\, t_1 \\ i & ::= & \mathtt{access}\, n \mid \mathtt{close}\, c \mid \mathtt{push}\, c \end{array}$$

- Compiler:

$$\begin{array}{rcl} [\![n]\!] & = & \mathtt{access}\, n \\ [\![\lambda t]\!] & = & \mathtt{close}\, [\![t]\!] \\ [\![t_0\, t_1]\!] & = & \mathtt{push}\, [\![t_1]\!]; [\![t_0]\!] \end{array}$$

- Expressible values, residual terms (using infix $\underline{@}$ to denote application), evaluation contexts, and reification contexts:

$$\begin{array}{rcl} v & ::= & \mathtt{FUN}[c,\ e] \mid \mathtt{RES}[r] \\ r & ::= & x \mid \underline{\lambda} x.r \mid r_0 \,\underline{@}\, r_1 \\ ke & ::= & \mathtt{ECONT0} \mid \mathtt{ECONT1}(x, kr) \mid \mathtt{ECONT2}(r, ke) \mid \\ & & \mathtt{ECONT3}(c, e, ke) \mid \mathtt{ECONT4}(c, e, ke) \\ kr & ::= & \mathtt{RCONT0} \mid \mathtt{RCONT1}(x, kr) \mid \mathtt{RCONT2}(r, ke) \end{array}$$

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

c	$\Rightarrow$	$\langle c,\ nil,\ \mathtt{ECONT0}\rangle$
$\langle \mathtt{access}\, n; c,\ e,\ ke\rangle$	$\Rightarrow_{eval}$	$\langle ke,\ e(n)\rangle$
$\langle \mathtt{push}\, c'; c,\ e,\ ke\rangle$	$\Rightarrow_{eval}$	$\langle c,\ e,\ \mathtt{ECONT4}(c', e, ke)\rangle$
$\langle \mathtt{close}\, c'; c,\ e,\ ke\rangle$	$\Rightarrow_{eval}$	$\langle ke,\ \mathtt{FUN}[c',\ e]\rangle$
$\langle \mathtt{ECONT0},\ v\rangle$	$\Rightarrow_{apply\_ke}$	$\langle v,\ \mathtt{RCONT0}\rangle$
$\langle \mathtt{ECONT1}(x, kr),\ v\rangle$	$\Rightarrow_{apply\_ke}$	$\langle v,\ \mathtt{RCONT1}(x, kr)\rangle$
$\langle \mathtt{ECONT2}(r, ke),\ v\rangle$	$\Rightarrow_{apply\_ke}$	$\langle v,\ \mathtt{RCONT2}(r, ke)\rangle$
$\langle \mathtt{ECONT3}(c, e, ke),\ v\rangle$	$\Rightarrow_{apply\_ke}$	$\langle c,\ v :: e,\ ke\rangle$
$\langle \mathtt{ECONT4}(c, e, ke),\ \mathtt{FUN}[c',\ e']\rangle$	$\Rightarrow_{apply\_ke}$	$\langle c,\ e,\ \mathtt{ECONT3}(c', e', ke)\rangle$
$\langle \mathtt{ECONT4}(c, e, ke),\ \mathtt{RES}[r]\rangle$	$\Rightarrow_{apply\_ke}$	$\langle c,\ e,\ \mathtt{ECONT2}(r, ke)\rangle$
$\langle \mathtt{FUN}[c,\ e],\ kr\rangle$	$\Rightarrow_{apply\_kr}$	$\langle c,\ \mathtt{RES}[x] :: e,\ \mathtt{ECONT1}(x, kr)\rangle$ *where x is fresh*
$\langle \mathtt{RES}[r],\ kr\rangle$	$\Rightarrow_{apply\_kr}$	$\langle kr,\ r\rangle$
$\langle \mathtt{RCONT1}(x, kr),\ r\rangle$	$\Rightarrow_{reify}$	$\langle kr,\ \underline{\lambda} x.r\rangle$
$\langle \mathtt{RCONT2}(r', ke),\ r\rangle$	$\Rightarrow_{reify}$	$\langle ke,\ \mathtt{RES}[r' \,\underline{@}\, r]\rangle$
$\langle \mathtt{RCONT0},\ r\rangle$	$\Rightarrow_{reify}$	r

Variables in a source term are represented by their de Bruijn index. Variables in a residual term are represented by their name. The virtual machine consists of four mutually recursive transition functions. The first transition function operates on triples consisting of a list of instructions, an environment, and an evaluation context. The second operates on pairs consisting of an evaluation context and a value. The third operates on pairs consisting of a value and a reification context. The fourth operates on pairs consisting of a reification context and a residual term.

The instruction set in the target language and the compiler in the same as in the CEK virtual machine, in Section 2.3. We conclude that this new virtual machine is a strong-normalization counterpart of the CEK virtual machine (which performs weak normalization).

3.4 Call by need

As in Section 2.4, implementing the thunks of Section 3.2 as memo-thunks yields a call-by-need version of the normalization function. It is then a simple exercise to derive the corresponding abstract machine, compiler, and virtual machine.

3.5 Summary, related work, and conclusions

In Section 2, we have shown how to derive a compiler and a virtual machine from an evaluation function. In this section, we have shown that a similar derivation can be carried out for a normalization function. The result was then a compiler and a virtual machine for weak normalization; it is now a compiler and a virtual machine for strong normalization.

Evaluation functions and normalization functions depend on the evaluation order of their meta-language. In Sections 2.2 and 2.3, we have capitalized on this dependence by deriving two distinct virtual machines—Krivine's virtual machine and the CEK virtual machine—from a call-by-name evaluation function and from a call-by-value evaluation function implementing the standard semantics of the λ-calculus. In this section, we have capitalized again on this dependence by deriving two virtual machines—a generalization of Krivine's virtual machine and a generalization of the CEK virtual machine—from a call-by-name normalization function and from a call-by-value normalization function corresponding to the standard semantics of the λ-calculus.

We are aware of two other machines for strong normalization: one is due to Crégut and is an abstract machine that generalizes Krivine's abstract machine [12] and the other is due to Laulhere, Grégoire, and Leroy and is a virtual machine that generalizes Leroy's Zinc virtual machine [27]:

- Crégut's is an abstract machine in the sense that it operates directly over the source term; however, we observe that some of its transition steps are not elementary (for example, one step performs a non-atomic operation over all the elements of a stack), which makes it closer to being an algorithm than an abstract machine.
- Laulhere, Grégoire, and Leroy's is a virtual machine in the sense that it first compiles the source term into a series of elementary instructions.

Both machines were invented and each needed to be proven correct. In contrast, we started from a proved normalization function and we derived a machine using meaning-preserving transformations. In an independent work, we have also constructed an abstract machine that generalizes Krivine's abstract machine (and whose transition steps all are elementary). We believe that we can construct a normalization function out of Grégoire and Leroy's virtual machine. Assuming that it exists, this normalization function would correspond to a stack semantics of the λ-calculus. We would then be in position to construct normalization functions and abstract machines for other stack semantics of the λ-calculus, e.g., the CAM, the VEC machine, the CLS machine, and the SECD machine.

We have presented two virtual machines performing normalization. The starting normalization function itself, however, does not change—what changes is the evaluation order of its meta-language, as captured in the CPS transformation. In that, and as in Section 2, we capitalize on the interplay between defining language and defined language identified by Reynolds. That normalization functions are sensitive to the evaluation order of the meta-language is now folklore in the normalization-by-evaluation community [7]. That they can give rise to compilers and virtual machines, however, is new.

4 Virtual machines and compilers from interpreters

We observe that existing compilers and virtual machines are presented in the literature as if they were in derived form, i.e., as a compositional compiling function into byte-code instructions and as a transition system over a sequence of byte-code instructions. We consider three such examples: the Categorical Abstract Machine (Section 4.1), the VEC machine (Section 4.2), and the Zinc abstract machine (Section 4.3).

4.1 The Categorical Abstract Machine

Our starting point is the CAM and its compiler, restricted to the λ-calculus with pairs and one distinguished literal, `nil` [11, 14]. The compiler is defined compositionally, and the CAM is defined as a transition system operating on triples consisting of an instruction list, an environment, and a stack of expressible values. The environment is represented as a list, encoded with pairs. In the following, t denotes terms, i denotes instructions, c denotes lists of instructions, v denotes expressible values, and s denotes stacks of expressible values. Variables n are represented by their de Bruijn index.

- Source and target syntax:

$$\begin{array}{lcl} t & ::= & n \mid \lambda t \mid t_0\, t_1 \mid \texttt{nil} \mid \texttt{mkpair}(t_1, t_2) \mid \texttt{car}\, t \mid \texttt{cdr}\, t \\ i & ::= & \texttt{fst} \mid \texttt{snd} \mid \texttt{push} \mid \texttt{swap} \mid \texttt{cons} \mid \texttt{call} \mid \texttt{cur}\, c \mid \texttt{quote}\, v \end{array}$$

- Compiler:

$$\begin{array}{rcl}
[\![0]\!] & = & \texttt{snd} \\
[\![n]\!] & = & \texttt{fst};\, [\![n-1]\!] \\
[\![\lambda t]\!] & = & \texttt{cur}\, [\![t]\!] \\
[\![t_0\, t_1]\!] & = & \texttt{push};\, [\![t_0]\!];\, \texttt{swap};\, [\![t_1]\!];\, \texttt{cons};\, \texttt{call} \\
[\![\texttt{nil}]\!] & = & \texttt{quote}\, \mathit{null} \\
[\![\texttt{mkpair}(t_1,\, t_2)]\!] & = & \texttt{push};\, [\![t_1]\!];\, \texttt{swap};\, [\![t_2]\!];\, \texttt{cons} \\
[\![\texttt{car}\, t]\!] & = & [\![t]\!];\, \texttt{fst} \\
[\![\texttt{cdr}\, t]\!] & = & [\![t]\!];\, \texttt{snd}
\end{array}$$

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

$$v \quad ::= \quad \mathit{null} \ \mid\ (v_1,\, v_2) \ \mid\ [v,\, c]$$

- Initial transition, transition rules, and final transition:

$$\begin{array}{|rcl|}
\hline
c & \Rightarrow & \langle c,\, \mathit{null},\, \mathit{nil} \rangle \\
\hline
\langle \texttt{fst};c,\, (v_1,\, v_2),\, s \rangle & \Rightarrow & \langle c,\, v_1,\, s \rangle \\
\langle \texttt{snd};c,\, (v_1,\, v_2),\, s \rangle & \Rightarrow & \langle c,\, v_2,\, s \rangle \\
\langle \texttt{quote}\, v';c,\, v,\, s \rangle & \Rightarrow & \langle c,\, v',\, s \rangle \\
\langle \texttt{cur}\, c';c,\, v,\, s \rangle & \Rightarrow & \langle c,\, [v,\, c'],\, s \rangle \\
\langle \texttt{push};c,\, v,\, s \rangle & \Rightarrow & \langle c,\, v,\, v :: s \rangle \\
\langle \texttt{swap};c,\, v,\, v' :: s \rangle & \Rightarrow & \langle c,\, v',\, v :: s \rangle \\
\langle \texttt{cons};c,\, v,\, v' :: s \rangle & \Rightarrow & \langle c,\, (v',\, v),\, s \rangle \\
\langle \texttt{call};c,\, ([v,\, c'],\, v'),\, s \rangle & \Rightarrow & \langle c';c,\, (v,\, v'),\, s \rangle \\
\hline
\langle \mathit{nil},\, v,\, \mathit{nil} \rangle & \Rightarrow & v \\
\hline
\end{array}$$

This specification lends itself to the following factorization, where `oo` is an infix operator. We successively define the syntax of source terms, the signature of an interpretation, and a generic functor over source terms.

```
structure Source
= struct
    datatype term = IND of int
                  | ABS of term
                  | APP of term * term
                  | NIL
                  | MKPAIR of term * term
                  | CAR of term
                  | CDR of term
  end
```

Programs are closed terms.

```
     signature INTERPRETATION
     = sig
         type computation
         type result

         val fst : computation
         val snd : computation
         val push : computation
         val swap : computation
         val cons : computation
         val call : computation
         val cur : computation -> computation
         val quote_null : computation
         val combine : computation * computation -> computation
         val compute : computation -> result
       end

functor mkProcessor (structure I : INTERPRETATION)
: sig val main : Source.program -> I.result end
= struct
    val (op oo) = I.combine

    (*  access : int -> computation  *)
    fun access 0 = I.snd
      | access n = I.fst oo (access (n - 1))

    (*  walk : Source.term -> computation  *)
    fun walk (Source.IND n)
        = access n
      | walk (Source.ABS t)
        = I.cur (walk t)
      | walk (Source.APP (t0, t1))
        = I.push oo (walk t0) oo I.swap oo (walk t1) oo I.cons oo I.call
      | walk (Source.NIL)
        = I.quote_null
      | walk (Source.MKPAIR (t1, t2))
        = I.push oo (walk t1) oo I.swap oo (walk t2) oo I.cons
      | walk (Source.CAR t)
        = (walk t) oo I.fst
      | walk (Source.CDR t)
        = (walk t) oo I.snd

    (*  main : term -> result  *)
    fun main t
        = I.compute (walk t)
  end
```

It is straightforward to specify a compiling interpretation such that instantiating the functor with this interpretation yields the CAM compiler. In this interpretation, the type `result` is defined as a list of target instructions. If the type `computation` is also defined as a list of target instructions, then `combine` is defined as list concatenation; if it is defined with an accumulator, then `combine` is defined as function composition.

We can also specify an evaluating interpretation by threading a register, a stack, and a continuation and by cut-and-pasting the actions of the CAM on the stack for each of the target instructions. In this interpretation, `combine` is defined as reverse function composition in CPS:[2]

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

    type computation = expval * expval list *
                       (expval * expval list -> expval) -> expval
    type result = expval

    val fst = (fn (PAIR (v1, v2), s, k) => k (v1, s))
    val snd = (fn (PAIR (v1, v2), s, k) => k (v2, s))
    val quote_null = (fn (v, s, k) => k (NULL, s))
    fun cur f = (fn (v, s, k) => k (CLOSURE (v, f), s))
    val call = (fn (PAIR (CLOSURE (v, f), v'), s, k)
                 => f (PAIR (v, v'), s, k))
    val push = (fn (v, s, k) => k (v, v :: s))
    val swap = (fn (v, v' :: s, k) => k (v', v :: s))
    val cons = (fn (v, v' :: s, k) => k (PAIR (v', v), s))
    fun combine (f, g)
        = (fn (v, s, k) => f (v, s, fn (v', s') => g (v', s', k)))

    fun compute c
        = c (NULL, nil, fn (v, nil) => v)
  end

structure Eval_CAM = mkProcessor (structure I = I_eval_CAM)
```

Inlining the functor instantiation, simplifying, uncurrying `access` and `walk`, and renaming `walk` into `eval` yield a compositional evaluator in CPS. Its direct-style counterpart reads as follows:

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

[2]Type-wise, it would actually be more true to CPS to define `expval` as a polymorphic data type parameterized with the type of answers.

```
    (*  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 (MKPAIR (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 compositional evaluator crystallizes the denotational content of the CAM as a stack semantics in the sense of Milne and Strachey [42, 50]. In particular, the meaning of a term is a partial endofunction over a stack of expressible values with the top element of the stack cached in a register. To the best of our knowledge, this denotational characterization of the CAM is new. It falls out of the scope of this article to cast this evaluator in category-theoretic terms, but we do wish to stress the two following points:

1. it is very simple to adapt the evaluator to an alternative language (e.g., with call-by-name); and
2. it is also simple to extend the evaluator to handle a richer source language.

In both cases, the factorization provides precise guidelines to extend the CAM, based on the adapted or extended evaluator. This parallel development of the evaluator and of the CAM contrasts with the original development of the CAM, where source language extensions are handled by CAM extensions with no reference to the original categorical semantics [11].

4.2 The VEC machine

In Chapter 10 of his book on denotational semantics [50], Schmidt presents the VEC machine and its compiler. The compiler is compositional and the virtual machine is a transition system operating on triples consisting of a stack of values, a stack of environments, and a list of instructions. The compiler accepts both call-by-name ($\lambda^{\mathrm{n}}x.t$) and call-by-value ($\lambda^{\mathrm{v}}x.t$) parameter passing, as in Algol.

In the following we illustrate primitive operations in the VEC machine with one source predefined function, `succ`, and the corresponding target instruction, `incr`. We let t denote terms, l denote literals, b denote boolean literals, n denote numerals, V denote a stack of values, e denote environments, E denote a stack of environments, i denote target instructions, C denote a list of instructions, and m denote integer values. Variables x are represented by their name.

- Source and target syntax:

$$\begin{array}{rcl}
t & ::= & t_0\,t_1 \mid \lambda^{\mathrm{n}}x.t \mid \lambda^{\mathrm{v}}x.t \mid x \mid l \mid \mathtt{succ}\,t \mid \mathtt{if}\,t_0\,\mathtt{then}\,t_1\,\mathtt{else}\,t_2 \\
l & ::= & b \mid n \\
b & ::= & \mathtt{true} \mid \mathtt{false} \\
i & ::= & \mathtt{pushclosure}\,(C) \mid \mathtt{pushconst}\,l \mid \mathtt{call} \mid \mathtt{return} \mid \mathtt{push}\,x \mid \\
 & & \mathtt{bind}\,x \mid \mathtt{incr} \mid \mathtt{test}\,(C_1,\,C_2)
\end{array}$$

- Compiler:

$$\begin{array}{rcl}
[\![t_0\,t_1]\!] & = & \mathtt{pushclosure}\,([\![t_1]\!];\mathtt{return});[\![t_0]\!];\mathtt{call} \\
[\![\lambda^{\mathrm{n}}x.t]\!] & = & \mathtt{pushclosure}\,(\mathtt{bind}\,x;[\![t]\!];\mathtt{return}) \\
[\![\lambda^{\mathrm{v}}x.t]\!] & = & \mathtt{pushclosure}\,(\mathtt{call};\mathtt{bind}\,x;[\![t]\!];\mathtt{return}) \\
[\![x]\!] & = & \mathtt{push}\,x \\
[\![l]\!] & = & \mathtt{pushconst}\,l \\
[\![\mathtt{succ}\,t]\!] & = & [\![t]\!];\mathtt{incr} \\
[\![\mathtt{if}\,t_0\,\mathtt{then}\,t_1\,\mathtt{else}\,t_2]\!] & = & [\![t_0]\!];\mathtt{test}\,([\![t_1]\!],\,[\![t_2]\!])
\end{array}$$

- Expressible values (closures and primitive values):

$$\begin{array}{rcl}
v & ::= & [C,\,e] \mid pv \\
pv & ::= & m \mid \mathit{true} \mid \mathit{false}
\end{array}$$

- Initial transition, transition rules, and final transition:

C	$\Rightarrow$	$\langle nil,\, \texttt{mt} :: nil,\, C\rangle$
$\langle V,\, e :: E,\, \texttt{pushclosure}\, C'; C\rangle$	$\Rightarrow$	$\langle [C',\, e] :: V,\, e :: E,\, C\rangle$
$\langle V,\, E,\, \texttt{pushconst}\, l; C\rangle$	$\Rightarrow$	$\langle l :: V,\, E,\, C\rangle$
$\langle [C',\, e] :: V,\, E,\, \texttt{call}; C\rangle$	$\Rightarrow$	$\langle V,\, e :: E,\, C'; C\rangle$
$\langle V,\, e :: E,\, \texttt{return}; C\rangle$	$\Rightarrow$	$\langle V,\, E,\, C\rangle$
$\langle V,\, e :: E,\, \texttt{push}\, x; C\rangle$	$\Rightarrow$	$\langle pv :: V,\, e :: E,\, C\rangle,\ \textit{if}\ pv = e(x)$
$\langle V,\, e :: E,\, \texttt{push}\, x; C\rangle$	$\Rightarrow$	$\langle V,\, e' :: e :: E,\, C'; C\rangle,\ \textit{if}\ [C',\, e'] = e(x)$
$\langle v :: V,\, e :: E,\, \texttt{bind}\, x; C\rangle$	$\Rightarrow$	$\langle V,\, e[x \mapsto v] :: E,\, C\rangle$
$\langle m :: V,\, E,\, \texttt{incr}; C\rangle$	$\Rightarrow$	$\langle m + 1 :: V,\, E,\, C\rangle$
$\langle \textit{true} :: V,\, E,\, \texttt{test}\, (C_1,\, C_2); C\rangle$	$\Rightarrow$	$\langle V,\, E,\, C_1; C\rangle$
$\langle \textit{false} :: V,\, E,\, \texttt{test}\, (C_1,\, C_2); C\rangle$	$\Rightarrow$	$\langle V,\, E,\, C_2; C\rangle$
$\langle v :: V,\, E,\, nil\rangle$	$\Rightarrow$	v

This machine lends itself to a factorization similar to that of Section 4.1. Again, by inlining the functor instantiation, simplifying, and uncurrying, we obtain the following evaluator.

```
structure Eval_VEC
= struct
    datatype primval = NAT of int
                     | BOOL of bool

    datatype expval
        = PRIMVAL of primval
        | CLOSURE of (expval list * expval Env.env list ->
                      expval list * expval Env.env list) * expval Env.env

    (*  eval : term * expval list * expval env list  *)
    (*          ->  expval list * expval env list  *)
    fun eval (APP (t0, t1), V, e :: E)
        = let val ((CLOSURE (c', e')) :: V', E')
              = eval (t0,
                      (CLOSURE (fn (V', E')
                                  => let val (V, e :: E) = eval (t1, V', E')
                                     in (V, E)
                                     end, e)) :: V,
                      e :: E)
          in c' (V', e' :: E')
          end
      | eval (LAM_N (x, t), V, e :: E)  (* call by name *)
        = ((CLOSURE
            (fn (r :: V, e :: E)
                => let val (V, e :: E)
                           = eval (t, V, (Env.extend (x, r, e)) :: E)
                   in (V, E)
                   end,
             e)) :: V, e :: E)
      | eval (LAM_V (x, t), V, e :: E)  (* call by value *)
        = ((CLOSURE
            (fn ((CLOSURE (c', e)) :: V, E)
                => let val (r :: V', e' :: E') = c' (V, e :: E)
                       val (V'', e'' :: E'')
                           = eval (t, V', (Env.extend (x, r, e'))) :: E')
                   in (V'', E'')
                   end,
               e)) :: V, e :: E)
      | eval (VAR x, V, E)
        = (case Env.lookup (x, hd E)
             of (PRIMVAL g)
                => ((PRIMVAL g) :: V, E)
              | (CLOSURE (c', e))
                => c' (V, e :: E))
      | eval (CONST (Source.BOOL b), V, E)
        = ((PRIMVAL (BOOL b)) :: V, E)
      | eval (CONST (Source.INT n), V, E)
        = ((PRIMVAL (NAT n)) :: V, E)
      | eval (SUCC t, V', E')
        = let val ((PRIMVAL (NAT m)) :: V, E) = eval (t, V', E')
          in ((PRIMVAL (NAT (m + 1))) :: V, E)
          end
      | eval (IF (t0, t1, t2), V', E')
        = (case eval (t0, V', E')
             of ((PRIMVAL (BOOL true)) :: V, E)
                => eval (t1, V, E)
              | ((PRIMVAL (BOOL false)) :: V, E)
                => eval (t2, V, E))

    (*  main : term -> expval  *)
    fun main t
        = let val (v :: V, E) = eval (t, nil, [Env.mt])
          in v
          end
end
```

This interpreter reveals that the underlying denotational model of the VEC machine is a stack semantics. The meaning of a term is a partial endofunction over a stack of intermediate values and a stack of environments. All parameters are passed as thunks. The interpreter and the virtual machine are not properly tail-recursive.

4.3 The Zinc abstract machine

Like the CAM and the VEC machine, the Zinc abstract machine is also in derived form, i.e., it is described in the form of a compositional compiler and a virtual machine [27, 40]. We have just constructed the corresponding interpreter as this technical report is going to press.

4.4 Summary, related work, and conclusions

We have illustrated three instances of existing compilers and virtual machines that are already in derived form, and we have shown how to construct the corresponding interpreter. We have constructed the interpreters corresponding to the CAM, the VEC machine, and the Zinc abstract machine. To the best of our knowledge, the three interpreters are new.

5 Virtual machines that correspond to abstract machines

In the companion article [2], we exhibit the evaluators that correspond to the CLS abstract machine and the SECD abstract machine. In this section, we factor each of these evaluators into a compiler and a virtual machine. The results are the virtual-machine counterparts of the CLS abstract machine, and the SECD abstract machine.

5.1 The CLS machine

Our starting point is the evaluator corresponding to the CLS abstract machine [2, 30]. This evaluator encodes a stack semantics of the λ-calculus. It operates on triples consisting of a term, a stack of environments, and a stack of expressible values. When evaluating an application, the CLS machine duplicates the top element of the environment stack before evaluating the operator and the operand.

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

    (*  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 (e, t)) :: 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 (ENV e, t)) :: 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 usual sequence of steps—closure conversion, CPS transformation, and defunctionalization—would take us from this evaluator to the transition function of the CLS machine (i.e., to the CLS abstract machine). Instead, we curry this evaluator, make it compositional, and factor it into a sequence of combinators. The result is the following compiler and virtual machine, where t denotes terms, i denotes instructions, c denotes lists of instructions, e denotes environments, l denotes stacks of environments, v denotes expressible values, and s denotes stacks of expressible values:

- Source and target syntax:

$$\begin{array}{rcl} t & ::= & n \mid \lambda t \mid t_0\, t_1 \\ i & ::= & \texttt{access}\, n \mid \texttt{lam}\, c \mid \texttt{ap} \mid \texttt{push} \end{array}$$

- Compiler:

$$\begin{array}{rcl} [\![n]\!] & = & \texttt{access}\, n \\ [\![\lambda t]\!] & = & \texttt{lam}\, [\![t]\!] \\ [\![t_0\, t_1]\!] & = & \texttt{push}; [\![t_0]\!]; [\![t_1]\!]; \texttt{ap} \end{array}$$

- Expressible values (closures):

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

- Initial transition, transition rules, and final transition:

c	$\Rightarrow$	$\langle c,\, nil :: nil,\, nil\rangle$
$\langle \texttt{access}\, 0; c,\, (v :: e) :: l,\, s\rangle$	$\Rightarrow$	$\langle c,\, l,\, v :: s\rangle$
$\langle \texttt{access}\, (n+1); c,\, (v :: e) :: l,\, s\rangle$	$\Rightarrow$	$\langle \texttt{access}\, n; c,\, e :: l,\, s\rangle$
$\langle \texttt{lam}\, c; c',\, e :: l,\, s\rangle$	$\Rightarrow$	$\langle c',\, l,\, [c,\, e] :: s\rangle$
$\langle \texttt{ap}; c,\, l,\, v :: [c',\, e] :: s\rangle$	$\Rightarrow$	$\langle c'; c,\, (v :: e) :: l,\, s\rangle$
$\langle \texttt{push}; c,\, e :: l,\, s\rangle$	$\Rightarrow$	$\langle c,\, e :: e :: l,\, s\rangle$
$\langle nil,\, l,\, v :: s\rangle$	$\Rightarrow$	v

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

This version of the CLS machine coincides with the stratified CLS machine in Hannan's article at PEPM'91 [28, Figure 2]. (The names `lam`, `ap`, and `push` originate in this presentation.)

5.2 The SECD machine

Our starting point is the following evaluator corresponding to the SECD abstract machine [39]. This evaluator encodes a stack semantics of the λ-calculus. Compared to what we have presented elsewhere [2, 16], we have simplified away a control delimiter, which is an unnecessary artefact of the encoding.

```
structure Eval_SECD
= struct
    datatype value = INT of int
                   | CLOSURE of value list * value Env.env ->
                                value list * value Env.env

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

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

A derivation similar to that of Section 5.1 leads one to the same compiler as in Henderson's textbook [34, Chapter 6] and the following virtual machine, where t denotes terms, i denotes instructions, c denotes lists of instructions, e denotes environments, v denotes expressible values, and s denotes stacks of expressible values. (The names `access`, `close`, and `call` are due to Henderson.)

- Source and target syntax:

$$\begin{array}{rcl} t & ::= & x \mid \lambda x.\, t \mid t_0\, t_1 \\ i & ::= & \texttt{access}\, x \mid \texttt{close}(x, c) \mid \texttt{call} \end{array}$$

- Compiler:

$$\begin{array}{rcl} [\![x]\!] & = & \texttt{access}\,x \\ [\![\lambda x.\,t]\!] & = & \texttt{close}(x, [\![t]\!]) \\ [\![t_0\,t_1]\!] & = & [\![t_1]\!]; [\![t_0]\!]; \texttt{call} \end{array}$$

- Expressible values (closures):

$$v \;::=\; [x,\, c,\, e]$$

- Initial transition, transition rules, and final transition:

$$\begin{array}{|rcl|} \hline c & \Rightarrow & \langle nil,\, mt,\, c\rangle \\ \hline \langle s,\, e,\, \texttt{access}\,x; c\rangle & \Rightarrow & \langle e(x) :: s,\, e,\, c\rangle \\ \langle s,\, e,\, \texttt{close}(x, c'); c\rangle & \Rightarrow & \langle [x,\, c',\, e] :: s,\, e,\, c\rangle \\ \langle [x,\, c',\, e'] :: v :: s,\, e,\, \texttt{call}; c\rangle & \Rightarrow & \langle s,\, e'[x \mapsto v],\, c'; c\rangle \\ \hline \langle v :: s,\, e,\, nil\rangle & \Rightarrow & v \\ \hline \end{array}$$

Variables x are represented by their name, and the virtual machine operates on triples consisting of a stack of expressible values, an environment, and a list of instructions. This machine is an SEC machine since, due to the fact that we simplified away the control delimiter, there is no dump [2, 16].

5.3 Summary, related work, and conclusions

We have presented the virtual-machine counterparts of the CLS and SECD abstract machines, starting from the evaluator corresponding to these abstract machines.

The SECD virtual machine is a dump-free version of the one presented in Henderson's book [34, Chapter 6]. The SECD compiler is the same. Both are unmotivated in Henderson's book and we are not aware of any subsequent derivation of them except for the present one.

The CLS compiler and virtual machine were constructed by a stratification (pass separation) algorithm over a representation of the CLS abstract machine [28, 29]. We do not know whether Hannan's pass separation has been applied to Krivine's machine, the CEK machine, and the SECD machine, and whether it has been identified that the CAM, the VEC machine, and the Zinc abstract machine can be seen as the result of pass separation.

6 Conclusion and issues

Over thirty years ago, in his seminal article about definitional interpreters [49], Reynolds warned the world that direct-style interpreters depend on the evaluation order of their meta-language. Indeed, if the defining language of a definitional meta-circular interpreter follows call by value, the defined language follows call by value; and if it follows call by name, the defined language follows call by name. Continuation-passing style, however, provides evaluation-order independence.

As a constructive echo of Reynolds's warning, we have shown that starting from an evaluation function for the λ-calculus, if one (a) closure-converts it, i.e., defunctionalizes it in place, (b) CPS-transforms it, (c) defunctionalizes the continuations (a transformation which is also due to Reynolds [49]), and (d) factorizes it into a compiler and a virtual machine (as has been largely studied in the literature [50], most significantly by Wand [54]), one obtains:

- Krivine's machine if the CPS transformation is call by name, and
- the CEK machine if the CPS transformation is call by value.

Furthermore, we have shown that starting from a normalization function for the λ-calculus, the same steps lead one to:

- a generalization of Krivine's machine performing strong normalization if the CPS transformation is call by name, and
- a generalization of the CEK machine performing strong normalization if the CPS transformation is call by value.

Krivine's machine was discovered [38] and the CEK machine was invented [21]. Both have been studied (not always independently) and each has been amply documented in the literature. Neither has been related to the standard semantics of the λ-calculus in the sense of Milne and Strachey. Now that we see that they can be derived from the same evaluation function, we realize that they provide a new illustration of Reynolds's warning about meta-languages.

Besides echoing Reynolds's warning in a constructive fashion, we have pointed at the difference between virtual machines (which have an instruction set) and abstract machines (which operate directly on source terms), and we have shown how to derive a compiler and a virtual machine from an evaluation function, i.e., an interpreter. We have illustrated this derivation with a number of examples, reconstructing known evaluators, compilers, and virtual machines, and obtaining new ones. We have also shown that the evaluation functions underlying Krivine's machine and the CEK machine correspond to a standard semantics of the λ-calculus, and that the evaluation functions underlying the CAM, the VEC machine, the CLS machine and the SECD machine are partial endofunctions corresponding to a stack semantics. Finally, we have derived virtual machines performing strong normalization by starting from a normalization function.

We conclude on three more points:

1. The compositional nature of the compilers makes it simple to construct byte-code verifiers and to prove their soundness and completeness. Let us take a simple example.

   Definition 1 (Verifier) *Given the inference rules*

$$\frac{m \vdash^{at} c}{m \vdash^{nf} c} \qquad \frac{m+1 \vdash^{nf} c}{m \vdash^{nf} \texttt{grab}; c}$$

$$\frac{n < m}{m \vdash^{at} \texttt{access}\, n; nil} \qquad \frac{m \vdash^{nf} c \qquad m \vdash^{at} c'}{m \vdash^{at} \texttt{push}\, c; c'}$$

   we say that a program c compiled for Krivine's virtual machine is verified *whenever the judgment* $0 \vdash^{nf} c$ *is derivable.*

   We want to apply this verifier to any list of instructions for Krivine's virtual machine to directly determine whether it denotes the compiled counterpart of a closed term in normal form (instead of, e.g., decompiling it first into a λ-term and verifying that this λ-term is in normal form).

   Lemma 1 *For any integer m and for any list of instructions c*

$$\begin{cases} m \vdash^{nf} c \text{ is derivable} & \Rightarrow \quad \forall k{>}m \;\; k \vdash^{nf} c \\ m \vdash^{at} c \text{ is derivable} & \Rightarrow \quad \forall k{>}m \;\; k \vdash^{at} c. \end{cases}$$

   Proof: By simultaneous structural induction over derivation trees. □

   Theorem 1 (Soundness and completeness) *A list of instructions c denotes the compiled counterpart of a closed term in normal form if and only if c is verified in the sense of Definition 1.*

   Proof: Let NF denote the set of terms in normal form, AT denote the set of atomic terms in normal form. For a term t, let $P(t)$ denote the set of paths from the root of the term to each occurrence of a variable in t. For such a path p, let $v(p)$ denote the variable (represented by its de Bruijn index) ending the path and $n_\lambda(p)$ denote the number of λ-abstractions on the path.

   - If $0 \vdash^{nf} c$ is derivable then there exists a closed term t in normal form such that $[\![t]\!] = c$.

By simultaneous structural induction over derivation trees, we prove that for any integer m and for any list of instructions c

$$\begin{cases} m \vdash^{nf} c \textit{ is derivable} & \Rightarrow \quad \exists t {\in} NF \ \ [\![t]\!] = c \textit{ and} \\ & \qquad \forall p {\in} P(t) \ \ v(p) - n_\lambda(p) < m \\ m \vdash^{at} c \textit{ is derivable} & \Rightarrow \quad \exists t {\in} AT \ \ [\![t]\!] = c \textit{ and} \\ & \qquad \forall p {\in} P(t) \ \ v(p) - n_\lambda(p) < m. \end{cases}$$

Therefore, if $0 \vdash^{nf} c$ then $\exists t {\in} NF \ \ [\![t]\!] = c \textit{ and } \forall p {\in} P(t) \ \ v(p) < n_\lambda(p)$, which means that t is a closed term.

- If t is a closed term in normal form then $0 \vdash^{nf} [\![t]\!]$ is derivable.

  By simultaneous structural induction on source terms, we prove that for any term t

$$\begin{cases} t \in NF & \Rightarrow \quad \underset{p \in P(t)}{max} \, (v(p) - n_\lambda(p) + 1) \ \vdash^{nf} [\![t]\!] \\ t \in AT & \Rightarrow \quad \underset{p \in P(t)}{max} \, (v(p) - n_\lambda(p) + 1) \ \vdash^{at} [\![t]\!]. \end{cases}$$

  Therefore, if t is in normal form then $\underset{p \in P(t)}{max} \, (v(p) - n_\lambda(p) + 1) \ \vdash^{nf} [\![t]\!]$, and if t is closed then $\forall p {\in} P(t) \ \ (v(p) - n_\lambda(p) + 1) \leq 0$. Hence, $0 \vdash^{nf} c$ by Lemma 1. □

2. We observe that the derivation lends itself to a systematic factorization of typed source interpreters into type-preserving compilers and typed virtual machines with a typed instruction set.

3. The derivation is not restricted to functional programming languages. For example, we have factored interpreters for imperative languages and derived the corresponding compilers and virtual machines. We are currently studying interpreters for object-oriented languages and for logic programming languages.

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

References

[1] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalization by evaluation. *Mathematical Structures in Computer Science*, 2003. Accepted for publication.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. Technical Report BRICS RS-03-13, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2003.

[3] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In Joseph Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 203–210, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[4] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.

[5] 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.

[6] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[7] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Term rewriting for normalization by evaluation. *Information and Computation*, 2002. Accepted for publication.

[8] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[9] Charles Consel and Renaud Marlet. Architecturing software using a methodology for language development. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Tenth International Symposium on Programming Language Implementation and Logic Programming*, number 1490 in Lecture Notes in Computer Science, pages 170–194, Pisa, Italy, September 1998. Springer-Verlag.

[10] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

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

[12] 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.

[13] Djordje Čubrić, Peter Dybjer, and Philip J. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.

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

[15] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[16] 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.

[17] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995.

[18] 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.

[19] Olivier Danvy, Morten Rhiger, and Kristoffer Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.

[20] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

[21] 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.

[22] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

[23] 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.

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

[25] Mayer Goldberg. Gödelization in the λ-calculus. *Information Processing Letters*, 75(1-2):13–16, 2000.

[26] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[27] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, pages 235–246, Pittsburgh, Pennsylvania, September 2002. ACM Press.

[28] John Hannan. Staging transformations for abstract machines. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 130–141, New Haven, Connecticut, June 1991. ACM Press.

[29] John Hannan. On extracting static semantics. In Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, number 2566 in Lecture Notes in Computer Science, pages 157–171. Springer-Verlag, 2002.

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

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

[32] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual*

ACM Symposium on Principles of Programming Languages, pages 458–471, Portland, Oregon, January 1994. ACM Press.

[33] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(2):303–319, 1997. Extended version available as the technical report BRICS RS-97-7.

[34] Peter Henderson. *Functional Programming – Application and Implementation.* Prentice-Hall International, 1980.

[35] John Hughes. Super combinators: A new implementation method for applicative languages. In Daniel P. Friedman and David S. Wise, editors, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, Pennsylvania, August 1982. ACM Press.

[36] 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/`.

[37] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *The Handbook of Logic in Computer Science.* North-Holland, 1992.

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

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

[40] Xavier Leroy. The Zinc experiment: an economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, Le Chesnay, France, February 1990.

[41] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.

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

[43] Torben Æ. Mogensen. Gödelization in the untyped lambda-calculus. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 19–24, San Antonio, Texas, January 1999.

[44] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[45] 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.

[46] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

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

[48] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[49] 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).

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

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

[52] Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.

[53] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

[54] Mitchell Wand. Semantics-directed machine architecture. In Richard DeMillo, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241. ACM Press, January 1982.

[55] Mitchell Wand. From interpreter to compiler: a representational derivation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 306–324, Copenhagen, Denmark, October 1985. Springer-Verlag.

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