**Basic Research in Computer Science**



# A Functional Correspondence between Call-by-Need Evaluators and Lazy Abstract Machines

**Mads Sig Ager**
**Olivier Danvy**
**Jan Midtgaard**

**BRICS Report Series** **RS-04-3**

**ISSN 0909-0878** **February 2004**

**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/04/3/`

# A Functional Correspondence between Call-by-Need Evaluators and Lazy Abstract Machines *

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

February 2004

**Abstract**

We bridge the gap between compositional evaluators and abstract machines for the lambda-calculus, using closure conversion, transformation into continuation-passing style, and defunctionalization of continuations. This article is a followup of our article at PPDP 2003, where we consider call by name and call by value. Here, however, we consider call by need.

We derive a lazy abstract machine from an ordinary call-by-need evaluator that threads a heap of updatable cells. In this resulting abstract machine, the continuation fragment for updating a heap cell naturally appears as an 'update marker', an implementation technique that was invented for the Three Instruction Machine and subsequently used to construct lazy variants of Krivine's abstract machine. Tuning the evaluator leads to other implementation techniques such as unboxed values. The correctness of the resulting abstract machines is a corollary of the correctness of the original evaluators and of the program transformations used in the derivation.

*To appear in Information Processing Letters (extended version).
†Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.
‡IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: {`mads,danvy,jmi`}`@brics.dk`

# Contents

# 1 Background and introduction

In previous work [1], we reported a simple derivation that makes it possible to derive Krivine's machine from an ordinary call-by-name evaluator and Felleisen et al.'s CEK machine from an ordinary call-by-value evaluator, and to construct evaluators that correspond to Landin's SECD machine, Hannan and Miller's CLS machine, Schmidt's VEC machine, and Curien et al.'s Categorical Abstract Machine. This derivation consists of three successive off-the-shelf program transformations: closure conversion, transformation into continuation-passing style (CPS), and Reynolds's defunctionalization. By closure-converting the evaluator, its expressible, denotable, and storable values are made first order. By transforming it into continuation-passing style (CPS), its flow of control is made manifest as a continuation. By defunctionalizing this continuation, the flow of control is materialized as a first-order data structure. The result is a transition function, i.e., an abstract machine. We are not aware of any other derivation that accounts for independently designed evaluators and abstract machines, even though closure conversion, CPS transformation, and defunctionalization are each far from being new and their combination can be found, e.g., in the textbook *Essentials of Programming Languages* [17, 18].

The derivation also makes it possible to map refinements and variations from an evaluator to the corresponding abstract machine. For example, one can derive arbitrarily many "new" abstract machines by inlining monads in an evaluator expressed in Moggi's computational meta-language [2]. One can also reflect refinements and variations from an abstract machine to the corresponding evaluator. For example, the optimization leading to a properly tail-recursive SECD machine is not specific to abstract machines; it has a natural counterpart in the corresponding evaluator [10]. More generally, one can intervene at any point in the derivation to interject a concept and derive the corresponding evaluator and abstract machine.

So far, we have only considered call by name and call by value. In the present work, we consider call by need and we derive a lazy abstract machine from a call-by-need evaluator. We then outline possible variants, review related work, and conclude. This article can be read independently of our earlier work.

# 2 From evaluator to abstract machine

We start from a call-by-name evaluator for the $\lambda$-calculus, written in Standard ML [30]. To make it follow call by need [19, 23, 42], we thread a heap of updatable cells (Section 2.2). Threading this heap is akin to inlining a state monad [2]. Using updatable cells to implement call by need is traditional [3, page 333] [36, page 176] [43, page 81]. We then closure-convert the evaluator (Section 2.3), CPS-transform it (Section 2.4), and defunctionalize the continuations (Section 2.5). The result is the transition functions of a lazy abstract machine (Section 2.6).

## 2.1 A compositional evaluator

Our starting point is a call-by-name, higher-order, and compositional evaluator for the $\lambda$-calculus. We represent $\lambda$-terms as elements of the following inductive data type. A program is a closed term.

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

The evaluator is defined recursively over the structure of terms. It is compositional in the sense of denotational semantics because it defines the meaning of a term as a composition of the meaning of its parts. It is also higher-order because the `expval` and `denval` data types contain functions. An environment is represented as a list of values. The function `List.nth` returns the element at a given index in an environment. A program is evaluated in an empty environment.

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

    type env = denval list

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

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

As identified by Reynolds in his seminal article on definitional interpreters [38], a direct-style evaluator inherits the evaluation order of its meta-language. Since the meta-language is ML, a naive direct-style evaluator would entail call by value. In order to model call by name, we have used thunks [22]. In the next section, we model call-by-need evaluation by threading a heap of updatable cells in the evaluator.

## 2.2 Representing call by need by threading a heap of updatable cells

In order to model call-by-need evaluation, we introduce a heap structure with three operations:

- `Heap.allocate` stores a given value in a fresh heap cell and returns the location of this cell;
- `Heap.dereference` fetches the value stored in a given heap cell; and
- `Heap.update` updates a given heap cell with a given value.

We thread a heap through the evaluator. The evaluator uses the heap to store computed values and delayed computations. Variables now denote locations of cells in the heap. Evaluation of a variable bound to the location of a delayed computation forces this computation and updates the heap cell with the computed value. Evaluation of a variable bound to the location of a computed value immediately yields that value. Evaluation of an abstraction yields a value, which is a function expecting a heap and the location of its argument in that heap. For an application, a delayed computation representing the argument is allocated in the heap, the operator is evaluated, and both the location of the delayed argument and the heap are passed to the resulting function.

```
structure Eval1
= struct
    type env = Heap.location list

    datatype expval = FUN of Heap.location * heap -> expval * heap
         and stoval = DELAYED of heap -> expval * heap
                    | COMPUTED of expval
    withtype heap   = stoval Heap.heap

    (*  eval : term * env * heap -> expval * heap  *)
    fun eval (IND n, e, h)
        = let val l = List.nth (e, n)
          in case Heap.dereference (h, l)
               of (DELAYED u)
                  => let val (v, h') = u h
                         val h'' = Heap.update (h', l, COMPUTED v)
                     in (v, h'')
                     end
                | (COMPUTED v)
                  => (v, h)
          end
      | eval (ABS t, e, h)
        = (FUN (fn (l, h) => eval (t, l :: e, h)), h)
```

```
      | eval (APP (t0, t1), e, h)
        = let val (h', l)
                  = Heap.allocate (h, DELAYED (fn h => eval (t1, e, h)))
              val (FUN f, h'') = eval (t0, e, h')
          in f (l, h'')
          end

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

This call-by-need evaluator, like the one in `Eval0`, is compositional. It is also higher-order because of its storable values `stoval` and its expressible values `expval`. In the next section, we make it first order by defunctionalizing `stoval` and `expval`, as first suggested by Landin and Reynolds [27, 38].

## 2.3 Representing functions with closures

In `Eval1`, the inhabitants of the function space in `expval` are all instances of the $\lambda$-abstraction `fn (l, h) => eval (t, l :: e, h)` used in the meaning of an abstraction. Similarly, the inhabitants of the function space in `stoval` are all instances of the $\lambda$-abstraction `fn h => eval (t1, e, h)` used in the meaning of an application. In order to make the evaluator first order, we closure convert (i.e., we defunctionalize) these function spaces into tuples holding the values of the free variables of the corresponding $\lambda$-abstractions [13, 27, 38].

```
structure Eval2
= struct
    type env = Heap.location list

    datatype expval = CLO of term * env

    datatype stoval = DELAYED of term * env
                    | COMPUTED of expval

    type heap = stoval Heap.heap

    (*  eval : term * env * heap -> expval * heap  *)
    fun eval (IND n, e, h)
        = let val l = List.nth (e, n)
          in case Heap.dereference (h, l)
               of (DELAYED (t, e'))
                  => let val (v, h') = eval (t, e', h)
                         val h'' = Heap.update (h', l, COMPUTED v)
                     in (v, h'')
                     end
                | (COMPUTED v)
                  => (v, h)
          end
```

```
      | eval (ABS t, e, h)
        = (CLO (t, e), h)
      | eval (APP (t0, t1), e, h)
        = let val (h', l) = Heap.allocate (h, DELAYED (t1, e))
              val (CLO (t0', e'), h'') = eval (t0, e, h')
          in eval (t0', l :: e', h'')
          end

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

As a corollary of the correctness of defunctionalization [5, 31], this closure-converted evaluator is equivalent to the evaluator in `Eval1`, in the sense that `Eval1.main` and `Eval2.main` either both diverge or both converge on the same input; when they converge, `Eval2.main` yields the closure-converted counterpart of the result of `Eval1.main`. In particular, defunctionalizing expressible values and storable values does not change the call-by-need nature of storable values.

After closure conversion, the evaluator is no longer compositional, even though it is still recursive. In the next section, we make it tail recursive by transforming it into continuation-passing style [11, 35, 41].

## 2.4 Representing control with continuations

We CPS transform the evaluator. The auxiliary functions (for the environment and for the heap) are atomic and therefore we leave them in direct style [12].

```
structure Eval3
= struct
    type env = Heap.location list

    datatype expval = CLO of term * env

    datatype stoval = DELAYED of term * env
                    | COMPUTED of expval

    type heap = stoval Heap.heap

    (*  eval : term * env * heap * (expval * heap -> 'a) -> 'a  *)
    fun eval (IND n, e, h, k)
        = let val l = List.nth (e, n)
          in case Heap.dereference (h, l)
               of (DELAYED (t, e'))
                  => eval (t, e', h,
                           fn (v, h')
                              => let val h''
                                         = Heap.update (h', l, COMPUTED v)
                                 in k (v, h'')
                                 end)
```

```
              | (COMPUTED v)
                => k (v, h)
        end
    | eval (ABS t, e, h, k)
      = k (CLO (t, e), h)
    | eval (APP (t0, t1), e, h, k)
      = let val (h', l) = Heap.allocate (h, DELAYED (t1, e))
        in eval (t0, e, h',
                 fn (CLO (t0', e'), h'') => eval (t0', l :: e', h'', k))
        end

  (*  main : term -> expval * heap  *)
  fun main t
      = eval (t, nil, Heap.empty, fn (v, h) => (v, h))
end
```

As a corollary of the correctness of the CPS transformation [35], this CPS-transformed evaluator is equivalent to the evaluator in `Eval2`, in the sense that `Eval2.main` and `Eval3.main` either both diverge or both converge on the same input; when they converge, they yield first-order values that are isomorphic. In particular, CPS transforming the evaluator does not change the call-by-need nature of storable values.

After CPS transformation, the evaluator is higher order because of the continuations. In the next section, we make it first order by defunctionalizing them [13, 38].

## 2.5 Representing continuations using defunctionalization

In `Eval3`, the function space of continuations is inhabited by instances of three $\lambda$-abstractions: one with no free variables in the definition of `main` (the initial continuation), one with two free variables in the meaning of a variable, and one with two free variables in the meaning of an application. Defunctionalizing the continuations amounts to introducing a data type `cont` with three summands and defining the corresponding apply function `apply_cont`. Each summand holds the values of the free variables of the corresponding $\lambda$-abstraction.

```
structure Eval4
= struct
    type env = Heap.location list

    datatype expval = CLO of term * env

    datatype stoval = DELAYED of term * env
                    | COMPUTED of expval

    datatype cont = CONT0
                  | CONT1 of Heap.location * cont
                  | CONT2 of Heap.location * cont
```

```
  type heap = stoval Heap.heap

  (*  eval : term * env * heap * cont -> expval * heap  *)
  fun eval (IND n, e, h, k)
      = let val l = List.nth (e, n)
        in case Heap.dereference (h, l)
             of (DELAYED (t, e'))
                => eval (t, e', h, CONT1 (l, k))
              | (COMPUTED v)
                => apply_cont (k, v, h)
        end
    | eval (ABS t, e, h, k)
      = apply_cont (k, CLO (t, e), h)
    | eval (APP (t0, t1), e, h, k)
      = let val (h', l) = Heap.allocate (h, DELAYED (t1, e))
        in eval (t0, e, h', CONT2 (l, k))
        end

  (*  apply_cont : cont * expval * heap -> expval * heap *)
  and apply_cont (CONT0, v, h)
      = (v, h)
    | apply_cont (CONT1 (l, k), v, h)
      = let val h' = Heap.update (h, l, COMPUTED v)
        in apply_cont (k, v, h')
        end
    | apply_cont (CONT2 (l, k), CLO (t, e), h)
      = eval (t, l :: e, h, k)

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

As a corollary of the correctness of defunctionalization [5, 31], this defunctionalized evaluator is equivalent to the evaluator in `Eval3`, in the sense that `Eval3.main` and `Eval4.main` either both diverge or both converge on the same input; when they converge, they yield first-order values that are isomorphic. In particular, defunctionalizing the continuations of the evaluator does not change the call-by-need nature of the storable values.

This evaluator uses two transition functions. It implements an abstract machine where the `cont` datatype elements are evaluation contexts. The `CONT1` evaluation context acts as an 'update marker' in the sense of Fairbairn and Wray's Three Instruction Machine [15].

The evaluation contexts can be given more meaningful names than `CONT0`, `CONT1`, and `CONT2`. We keep these names to stress that the evaluation contexts are not invented but appear naturally through the derivation as defunctionalized continuations.

## 2.6 A lazy abstract machine

In `Eval4`, the `cont` data type is isomorphic to the data type of lists containing two kinds of heap locations. Representing `cont` as such a list (used as a single-threaded stack), we obtain the following stack-based abstract machine. The machine consists of two mutually recursive transition functions performing elementary state transitions. The first transition function operates on quadruples consisting of a term $t$, an environment $e$ (a list of denotable values, i.e., of heap locations), a heap $h$ mapping locations $\ell$ to storable values (tagged with $D$ for delayed or $C$ for computed), and a stack $k$ of evaluation contexts consisting of heap locations tagged with $C_1$ (corresponding to `CONT1`) and $C_2$ (corresponding to `CONT2`). The second transition function operates on triples consisting of a stack, an expressible value, and a heap.

- Source syntax: $t ::= n \mid \lambda t \mid t_0\ t_1$
- Expressible values (closures): $v ::= [t,\ e]$
- Storable values: $w ::= D(t, e) \mid C(v)$
- Initial transition, transition rules, and final transition:

| | | |
|---|---|---|
| $t$ | $\Rightarrow_{init}$ | $\langle t,\ nil,\ Heap.empty,\ nil\rangle$ |
| $\langle n,\ e,\ h,\ k\rangle$ | $\Rightarrow_{eval}$ | $\langle t,\ e',\ h,\ (C_1\ \ell) :: k\rangle$ |
| | | *if* $List.nth(e, n) = \ell$ |
| | | *and* $Heap.dereference(h, \ell) = D(t, e')$ |
| $\langle n,\ e,\ h,\ k\rangle$ | $\Rightarrow_{eval}$ | $\langle k,\ v,\ h\rangle$ |
| | | *if* $List.nth(e, n) = \ell$ |
| | | *and* $Heap.dereference(h, \ell) = C(v)$ |
| $\langle \lambda t,\ e,\ h,\ k\rangle$ | $\Rightarrow_{eval}$ | $\langle k,\ [t,\ e],\ h\rangle$ |
| $\langle t_0\ t_1,\ e,\ h,\ k\rangle$ | $\Rightarrow_{eval}$ | $\langle t_0,\ e,\ h',\ (C_2\ \ell) :: k\rangle$ |
| | | *where* $Heap.allocate(h, D(t_1, e)) = (h', \ell)$ |
| $\langle (C_1\ \ell) :: k,\ v,\ h\rangle$ | $\Rightarrow_{apply}$ | $\langle k,\ v,\ h'\rangle$ |
| | | *where* $Heap.update(h, \ell, C(v)) = h'$ |
| $\langle (C_2\ \ell) :: k,\ [t,\ e],\ h\rangle$ | $\Rightarrow_{apply}$ | $\langle t,\ \ell :: e,\ h,\ k\rangle$ |
| $\langle nil,\ v,\ h\rangle$ | $\Rightarrow_{final}$ | $\langle v,\ h\rangle$ |

This abstract machine is in one-to-one correspondence with the evaluator of Section 2.5. It is also equivalent to the evaluator of Section 2.2, since it was derived using meaning-preserving transformations that make the evaluation steps explicit without changing the call-by-need nature of evaluation. We recognize it as a lazy and properly tail-recursive variant of Krivine's machine [9, 26]: (the locations of) arguments are pushed on the stack and functions are directly entered and find (the locations of) their arguments on the stack. In particular, $C_1$ acts as the update marker of the Three Instructions Machine [15].

# 3 Variants and extensions

In Section 2.2, the order of operations in `Eval1` can be changed and `Eval1` can be optimized by fold-unfold transformation [24]:

(a) In the application branch of the evaluator, the order of the heap allocation and the evaluation of the operator can be swapped. In the resulting abstract machine, the `CONT2` evaluation context holds the argument term and the environment and the heap allocation takes place in the apply transition function.

(b) If the operand of an application is a variable, we can look it up directly, thereby avoiding the construction of space-leaky chains of thunks. (Actually, this optimization is the compilation model of call by name in Algol 60 for identifiers occurring as actual parameters [37, Section 2.5.4.10, pages 109-110].) In terms of the evaluator of Section 2.1, this optimization corresponds to $\eta$-reducing a thunk.

(c) If the operand of an application is a $\lambda$-abstraction, we can store the corresponding closure as a computed value rather than as a delayed computation. Alternatively we can extend the domain of denotable values with unboxed values and pass the closure directly to the called function.

Putting (b) and (c) together guarantees that a thunk is only used to delay the evaluation of an application. We can exploit this invariant with a specialized version of `eval` for applications, to minimize the number of structural tests over the source syntax. The two versions of `eval` are mutually (tail-)recursive.

Each of these variants gives rise to an abstract machine. The structure of each of these abstract machines reflects the structure of the corresponding evaluator. For example, the two mutually (tail-)recursive versions of `eval` give rise to two $\Rightarrow_{eval}$ transition functions.

Finally, we handle language extensions by extending the original evaluator and deriving the corresponding abstract machine.

# 4 Related work

Designing abstract machines is a favorite among functional programmers [14]. On the one hand, few abstract machines are actually derived with meaning-preserving steps, and on the other hand, few abstract machines are invented from scratch. Instead, most abstract machines are inspired by one formal system or another and they are subsequently proved to correctly implement a given evaluation strategy [8, 9, 20, 21, 26].

Constructing call-by-need abstract machines for the $\lambda$-calculus has traditionally been done by constructing call-by-name abstract machines for the $\lambda$-calculus, and then introducing an update mechanism at the machine level. For

example, Fairbairn and Wray invented update markers for the Three Instruction Machine to make it lazy, and this mechanism was later used by Crégut and Sestoft to construct lazy variants of Krivine's abstract machine [9, 15, 39].

A forerunner of our work is Sestoft's derivation of a lazy abstract machine from Launchbury's natural semantics for lazy evaluation [28, 40]. Sestoft starts from a natural semantics whereas we start from a compositional environment-based evaluator. He considers $\lambda$-terms with recursive bindings whereas we only consider $\lambda$-terms here. He proceeds in several intuitive but non-standard steps, and therefore needs to prove the correctness of each intermediate result, whereas we rely on the well-established correctness of each of the intermediate transformations. First, he introduces a stack containing unevaluated arguments and update markers, whereas we mechanically obtain this stack as a defunctionalized continuation in the sense of Reynolds. Second, he introduces environments and closures, whereas we start from an environment-based evaluator which we closure convert. Third, he introduces variable indices, whereas we start from abstract syntax with de Bruijn indices (compiling names into lexical offsets could be done at any point of our derivation). Sestoft then extends the source language and repeats the derivation, obtaining extended versions of his abstract machine, and so do we for language extensions. Finally, he optimizes his abstract machines into several variants, whereas to a non-trivial extent, we maintain the correspondence between optimized evaluators and abstract machines (for example, we would obtain environment trimming by flat closure conversion instead of by deep closure conversion). More generally, Sestoft concentrates on deriving one particular abstract machine out of one natural semantics, whereas the present article is part of a general investigation of a correspondence between evaluators and abstract machines [1, 2, 6, 7, 10].

Friedman, Ghuloum, Siek and Winebarger consider optimizations of a lazy Krivine machine [16]. Their starting point is a machine that corresponds to the evaluator described in Variant (a) of Section 3. To optimize this machine, they use the classic optimization from Variant (b) in Section 3, and storable values are accessed via an extra indirection. This extra indirection makes it possible to avoid building lists of update markers on the stack, thereby eliminating sequences of updates of multiple heap locations with the same value. Instead, the value can be stored in only one location and that location can be shared. Friedman, Ghuloum, Siek and Winebarger prove the correctness of these machines and measure their effectiveness compared to Sestoft's machine.

Josephs gave a continuation semantics for lazy functional languages [25]. The CPS-transformed evaluator in Section 2.4 closely corresponds to this denotational semantics.

Ariola, Felleisen, Maraist, Odersky, and Wadler have developed a call-by-need lambda calculus that models call by need syntactically [4]. They deliberately do not use a heap and assignments to model call-by-need evaluation and their machine is therefore very different from the one derived here.

Besides environment-based machines [29, 33], a wealth of machines based on graph reduction also exist [32, 34]. They illustrate considerable ingenuity and cleverness. A byproduct of our research program is to determine how much

of this cleverness is intrinsic to abstract machines per se and how much can be accounted for as a refined evaluation function. The functional correspondence can also provide guidelines for constructing complex abstract machines. For example, along the lines described in the present article, we can derive a lazy abstract machine handling arbitrarily complex computational effects—e.g., stack inspection—given a call-by-need evaluator equipped with the corresponding monad [2].

# 5 Conclusion and issues

We have presented a derivation of a lazy abstract machine from a call-by-need evaluator for the $\lambda$-calculus. The derivation originates in our previous work [1, 10]. It consists of three successive program transformations: closure conversion, CPS transformation, and defunctionalization. The correctness of the resulting abstract machine is thus a corollary of the correctness of the original evaluator and of the program transformations. The program transformations make evaluation steps explicit and do not change the call-by-need nature of evaluation.

In our previous work, we illustrated the correspondence between a number of evaluators and a number of abstract machines for the $\lambda$-calculus, and we showed how some abstract-machine features originate either in the corresponding evaluator or as an artefact of the derivation (e.g., the control stack being a defunctionalized continuation). The present work shows that the correspondence scales from call by value and call by name to call by need. It illustrates the correspondence between a number of call-by-need evaluators and a number of lazy abstract machines, and it shows how some abstract-machine features originate either in the corresponding evaluator—e.g., unboxed values, or as an artefact of the derivation—e.g., update markers. As a byproduct, one can now straightforwardly construct a range of lazy abstract machines, including lazy variants of Krivine's machine, Landin's SECD machine, Hannan and Miller's CLS machine, Schmidt's VEC machine, and Curien et al.'s Categorical Abstract Machine out of the corresponding call-by-need evaluators.

**Acknowledgments:** We are grateful to the anonymous referees and to Julia Lawall and Peter Sestoft for their comments. This work is supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-03-0545.

# References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN Interna-*

*tional Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, August 2003.

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Technical Report BRICS RS-03-35, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2003.

[3] Andrew W. Appel. *Modern Compiler Implementation in {C, Java, ML}*. Cambridge University Press, New York, 1998.

[4] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 233–246, San Francisco, California, January 1995. ACM Press.

[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, pages 420–447, Sendai, Japan, October 2001. Springer-Verlag.

[6] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report, Department of Computer Science, Queen Mary's College, Venice, Italy, January 2004. To appear.

[7] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. Technical Report BRICS RS-03-25, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. To appear in the proceedings of the 2003 International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003).

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

[9] 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.

[10] Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report BRICS RS-03-33, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2003.

[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. 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.

[13] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press.

[14] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.

[15] Jon Fairbairn and Stuart Wray. TIM: a simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 34–45, Portland, Oregon, September 1987. Springer-Verlag.

[16] Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Lynn Winebarger. Improving the lazy krivine machine. *Higher-Order and Symbolic Computation*, 2004. To appear.

[17] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill, 1991.

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

[19] Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In S. Michaelson and Robin Milner, editors, *Third International Colloquium on Automata, Languages, and Programming*, pages 257–284. Edinburgh University Press, Edinburgh, Scotland, July 1976.

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

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

[22] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3):303–319, 1997.

[23] Peter Henderson and James H. Morris Jr. A lazy evaluator. In Susan L. Graham, editor, *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 95–103. ACM Press, January 1976.

[24] Thomas Johnsson. Fold-unfold transformations on state monadic interpreters. In Kevin Hammond, David N. Turner, and Patrick M. Sansom, editors, *Proceedings of the 1994 Glasgow Workshop on Functional Programming*, Workshops in Computing, Ayr, Scotland, 1994. Springer-Verlag.

[25] Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68:105–111, 1989.

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

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

[28] John Launchbury. A natural semantics for lazy evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, January 1993. ACM Press.

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

[30] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[31] 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.

[32] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987.

[33] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[34] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1992.

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

[36] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, Cambridge, 1996.

[37] B. Randell and L. J. Russell. *ALGOL 60 Implementation*. Academic Press, New York, 1964.

[38] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

[39] Peter Sestoft. *Analysis and efficient implementation of functional programs*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1991. DIKU Rapport 92/6.

[40] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[41] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[42] Jean Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354, 1974.

[43] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1995.

# Recent BRICS Report Series Publications

**RS-04-3** **Mads Sig Ager, Olivier Danvy, and Jan Midtgaard.** ***A Functional Correspondence between Call-by-Need Evaluators and Lazy Abstract Machines*****. February 2004. 17 pp. This report supersedes the earlier BRICS report RS-03-24. Extended version of an article to appear in** ***Information Processing Letters*****.**

**RS-04-2** **Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh.** ***Cache-Oblivious Data Structures and Algorithms for Undirected Breadth-First Search and Shortest Paths*****. February 2004. 19 pp.**

**RS-04-1** **Luca Aceto, Willem Jan Fokkink, Anna Ingólfsdóttir, and Bas Luttik.** ***Split-2 Bisimilarity has a Finite Axiomatization over CCS with Hennessy's Merge*****. January 2004. 16 pp.**

**RS-03-53** **Kyung-Goo Doh and Peter D. Mosses.** ***Composing Programming Languages by Combining Action-Semantics Modules*****. December 2003. 39 pp. Appears in** ***Science of Computer Programming*****, 47(1):2–36, 2003.**

**RS-03-52** **Peter D. Mosses.** ***Pragmatics of Modular SOS*****. December 2003. 22 pp. Invited paper, published in Kirchner and Ringeissen, editors,** ***Algebraic Methodology and Software Technology: 9th International Conference*****, AMAST '02 Proceedings, LNCS 2422, 2002, pages 21–40.**

**RS-03-51** **Ulrich Kohlenbach and Branimir Lambov.** ***Bounds on Iterations of Asymptotically Quasi-Nonexpansive Mappings*****. December 2003. 24 pp.**

**RS-03-50** **Branimir Lambov.** ***A Two-Layer Approach to the Computability and Complexity of Real Numbers*****. December 2003. 16 pp.**

**RS-03-49** **Marius Mikucionis, Kim G. Larsen, and Brian Nielsen.** ***Online On-the-Fly Testing of Real-time Systems*****. December 2003. 14 pp.**

**RS-03-48** **Kim G. Larsen, Ulrik Larsen, Brian Nielsen, Arne Skou, and Andrzej Wasowski.** ***Danfoss EKC Trial Project Deliverables*****. December 2003. 53 pp.**