



A functional correspondence between call-by-need evaluators and lazy abstract machines

Mads Sig Ager, Olivier Danvy*, Jan Midtgaard

*BRICS, Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark*¹

Received 10 September 2003; received in revised form 19 February 2004

Communicated by J. Chomicki

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.

© 2004 Elsevier B.V. All rights reserved.

Keywords: Functional programming; Program derivation; Interpreters; Abstract machines; Closure conversion; CPS transformation; Defunctionalization

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 the CEK machine from an ordinary call-by-value evaluator, and to construct evaluators that correspond to the SECD machine, the CLS machine, the VEC machine, and the CAM. 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, its flow of control is made manifest as a continuation. By defunctionalizing this continuation, the flow of control

* Corresponding author.

E-mail addresses: mads@brics.dk (M.S. Ager), danvy@brics.dk (O. Danvy), jmi@brics.dk (J. Midtgaard).

¹ Basic Research in Computer Science (<http://www.brics.dk>), funded by the Danish National Research Foundation.

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* [13].

The derivation also makes it possible to map refinements and variations from an evaluator to the corresponding abstract machine. For example, one can derive “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 [7]. 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 λ -calculus, written in Standard ML. To make it follow call by need [14,17,27], 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, p. 333], [28, p. 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 λ -calculus. We represent λ -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 `denvval` 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 [25], 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. 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)
end

```

```

| 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 [20,25].

2.3. Representing functions with closures

In Eval1, the inhabitants of the function space in `expval` are all instances of the λ -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 λ -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 λ -abstractions [9,20,25].

```

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 [5,22], 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 [23].

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 [8].

```

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 [23], 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 [9,25].

2.5. Representing continuations using defunctionalization

In Eval3, the function space of continuations is inhabited by instances of three λ -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 λ -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 [5,22], 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 [11].

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 ℓ 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: $t ::= n|\lambda t|t_0t_1$.
- Expressible values (closures): $v ::= [t, e]$.
- Storable values: $w ::= D(t, e) | C(v)$.
- Initial transition, transition rules, and final transition:

$$\begin{array}{c}
 \hline
 t \Rightarrow_{\text{init}} \langle t, \text{nil}, \text{Heap.empty}, \text{nil} \rangle \\
 \hline
 \langle n, e, h, k \rangle \Rightarrow_{\text{eval}} \langle t, e', h, (C_1\ell) :: k \rangle \\
 \quad \text{if } \text{List.nth}(e, n) = \ell \\
 \quad \text{and } \text{Heap.dereference}(h, \ell) = D(t, e') \\
 \langle n, e, h, k \rangle \Rightarrow_{\text{eval}} \langle k, v, h \rangle \\
 \quad \text{if } \text{List.nth}(e, n) = \ell \\
 \quad \text{and } \text{Heap.dereference}(h, \ell) = C(v) \\
 \langle \lambda t, e, h, k \rangle \Rightarrow_{\text{eval}} \langle k, [t, e], h \rangle \\
 \langle t_0t_1, e, h, k \rangle \Rightarrow_{\text{eval}} \langle t_0, e, h', (C_2\ell) :: k \rangle \\
 \quad \text{where } \text{Heap.allocate}(h, D(t_1, e)) = (h', \ell) \\
 \hline
 \langle (C_1\ell) :: k, v, h \rangle \Rightarrow_{\text{apply}} \langle k, v, h' \rangle \\
 \quad \text{where } \text{Heap.update}(h, \ell, C(v)) = h' \\
 \langle (C_2\ell) :: k, [t, e], h \rangle \Rightarrow_{\text{apply}} \langle t, \ell :: e, h, k \rangle \\
 \hline
 \langle \text{nil}, v, h \rangle \Rightarrow_{\text{final}} \langle v, h \rangle \\
 \hline
 \end{array}$$

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 [6]: (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 Instruction Machine [11].

3. Variants

In Section 2.2, the order of operations in `Eva11` can be changed and `Eva11` can be optimized by fold-unfold transformation [18]:

- (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 [24, Section 2.5.4.10, pp. 109–110].) In terms of the evaluator of Section 2.1, this optimization corresponds to η -reducing a thunk.
- (c) If the operand of an application is a λ -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.

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.

4. Related work

Designing abstract machines is a favorite among functional programmers [10]. 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 [6,15,16].

Constructing call-by-need abstract machines for the λ -calculus has traditionally been done by constructing call-by-name abstract machines for the λ -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 [6,11,26].

A forerunner of our work is Sestoft's derivation of a lazy abstract machine from Launchbury's natural semantics for lazy evaluation [21,26]. Sestoft starts from a natural semantics whereas we start from a compositional environment-based evaluator. He considers λ -terms with recursive bindings whereas we only consider λ -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. 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 [1,2,7].

Friedman, Ghuloum, Siek and Winebarger consider optimizations of a lazy Krivine machine [12]. 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.

Josephs gave a continuation semantics for lazy functional languages [19]. 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, a wealth of machines based on graph reduction also exist [10]. 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 λ -calculus. The derivation originates in our previous work [1,7]. 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 λ -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.

Acknowledgements

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. An extended version of this article is available as the BRICS technical report RS-04-3.

References

- [1] M.S. Ager, D. Biernacki, O. Danvy, J. Midtgaard, A functional correspondence between evaluators and abstract machines, in: D. Miller (Ed.), *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, ACM Press, New York, 2003, pp. 8–19.
- [2] M.S. Ager, O. Danvy, J. 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, 2003.
- [3] A.W. Appel, *Modern Compiler Implementation in {C, Java, ML}*, Cambridge University Press, New York, 1998.
- [4] Z.M. Ariola, M. Felleisen, J. Maraist, M. Odersky, P. Wadler, The call-by-need lambda calculus, in: P. Lee (Ed.), *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, ACM Press, New York, 1995, pp. 233–246.
- [5] A. Banerjee, N. Heintze, J.G. Riecke, Design and correctness of program transformations based on control-flow analysis, in: N. Kobayashi, B.C. Pierce (Eds.), *Theoretical Aspects of Computer Software*, 4th International Symposium, TACS 2001, Sendai, Japan, in: *Lecture Notes in Computer Science*, vol. 2215, Springer, Berlin, 2001, pp. 420–447.

- [6] P. Crégut, An abstract machine for lambda-terms normalization, in: M. Wand (Ed.), *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, ACM Press, New York, 1990, pp. 333–340.
- [7] O. 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, 2003.
- [8] O. Danvy, J. Hatcliff, On the transformation between direct and continuation semantics, in: S. Brookes, M. Main, A. Melton, M. Mislove, D. Schmidt (Eds.), *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, New Orleans, LA, in: *Lecture Notes in Computer Science*, vol. 802, Springer, Berlin, 1993, pp. 627–648.
- [9] O. Danvy, L.R. Nielsen, Defunctionalization at work, in: H. Søndergaard (Ed.), *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, Firenze, Italy, ACM Press, New York, 2001, pp. 162–174.
- [10] S. Diehl, P. Hartel, P. Sestoft, Abstract machines for programming language implementation, *Future Generation Computer Systems* 16 (2000) 739–751.
- [11] J. Fairbairn, S. Wray, TIM: a simple, lazy abstract machine to execute supercombinators, in: G. Kahn (Ed.), *Functional Programming Languages and Computer Architecture*, Portland, OR, in: *Lecture Notes in Computer Science*, vol. 274, Springer, Berlin, 1987, pp. 34–45.
- [12] D.P. Friedman, A. Ghuloum, J.G. Siek, L. Winebarger, Improving the lazy Krivine machine, *Higher-Order and Symbolic Computation* (2004), in press.
- [13] D.P. Friedman, M. Wand, C.T. Haynes, *Essentials of Programming Languages*, second ed., MIT Press, Cambridge, MA, 2001.
- [14] D.P. Friedman, D.S. Wise, CONS should not evaluate its arguments, in: S. Michaelson, R. Milner (Eds.), *Third International Colloquium on Automata, Languages, and Programming*, Edinburgh University Press, Edinburgh, Scotland, 1976, pp. 257–284.
- [15] J. Hannan, D. Miller, From operational semantics to abstract machines, *Math. Struct. Comput. Sci.* 2 (4) (1992) 415–459.
- [16] T. Hardin, L. Maranget, B. Pagano, Functional runtime systems within the lambda-sigma calculus, *J. Funct. Programming* 8 (2) (1998) 131–172.
- [17] P. Henderson, J.H. Morris Jr, A lazy evaluator, in: S.L. Graham (Ed.), *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, ACM Press, New York, 1976, pp. 95–103.
- [18] T. Johnsson, Fold-unfold transformations on state monadic interpreters, in: K. Hammond, D.N. Turner, P.M. Sansom (Eds.), *Proceedings of the 1994 Glasgow Workshop on Functional Programming, Workshops in Computing*, Ayr, Scotland, Springer, Berlin, 1994.
- [19] M.B. Josephs, The semantics of lazy functional languages, *Theoret. Comput. Sci.* 68 (1989) 105–111.
- [20] P.J. Landin, The mechanical evaluation of expressions, *Comput. J.* 6 (4) (1964) 308–320.
- [21] J. Launchbury, A natural semantics for lazy evaluation, in: S.L. Graham (Ed.), *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, ACM Press, New York, 1993, pp. 144–154.
- [22] L.R. Nielsen, A denotational investigation of defunctionalization, Technical Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, 2000.
- [23] G.D. Plotkin, Call-by-name, call-by-value and the λ -calculus, *Theoret. Comput. Sci.* 1 (1975) 125–159.
- [24] B. Randell, L.J. Russell, *ALGOL 60 Implementation*, Academic Press, New York, 1964.
- [25] J.C. Reynolds, Definitional interpreters for higher-order programming languages, *Higher-Order and Symbolic Computation* 11 (4) (1998) 363–397. Reprinted from the *Proceedings of the 25th ACM National Conference* (1972), with a foreword.
- [26] P. Sestoft, Deriving a lazy abstract machine, *J. Funct. Programming* 7 (3) (1997) 231–264.
- [27] J. Vuillemin, Correct and optimal implementations of recursion in a simple programming language, *J. Comput. System Sci.* 9 (3) (1974) 332–354.
- [28] R. Wilhelm, D. Maurer, *Compiler Design*, Addison–Wesley, Reading, MA, 1995.