On the dynamic extent of delimited continuations

Dariusz Biernacki [a], Olivier Danvy [a], and Chung-chieh Shan [b]

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

[b] *Division of Engineering and Applied Sciences, Harvard University*
33 Oxford Street, Cambridge MA, USA

Abstract

We show that breadth-first traversal exploits the difference between the static delimited-control operator `shift` (alias $\mathcal{S}$) and the dynamic delimited-control operator `control` (alias $\mathcal{F}$). For the last 15 years, this difference has been repeatedly mentioned in the literature but it has only been illustrated with one-line toy examples. Breadth-first traversal fills this vacuum.

1 Introduction

Continuation-passing style (CPS) is a time-honored and logic-based format for functional programs where all intermediate results are named, all calls are tail calls, and programs are evaluation-order independent [16, 17, 19]. While this format has been an active topic of study, it also has been felt as a straightjacket both from a semantics point of view [8, 10] and from a programming point of view [6], where one would like to relax the tail-call constraint and compose continuations.

In direct style, continuations are accessed with control operators such as Reynolds's `escape` [17] and Scheme's `call/cc`. These control operators give access to the current continuation as a first-class value. Activating such a first-class continuation has the effect of resuming the computation at the point where this continuation was captured; the then-current continuation is *abandoned*. Such first-class continuations *do not return to the point of their activation*—they model jumps, i.e., tail calls [19, 20].

In direct style, composable continuations are also accessed with control operators such as Felleisen et al.'s `control` (alias $\mathcal{F}$) [10] and Danvy and Filinski's `shift` (alias $\mathcal{S}$) [6]. These control operators also give access to the current continuation as a first-class value; activating such a first-class continuation also has the effect of resuming the computation at the point where this continuation was captured; the then-current continuation, however, *is then resumed*. Such first-class continuations *return to the point of their activation*—they model non-tail calls.

For a first-class continuation to return to the point of its activation, one must declare its point of completion, since this point is no longer at the very end of the overall computation, as with traditional, undelimited first-class continuations. In direct style, this declaration is achieved with a new kind of operator, due to Felleisen [8]: a control delimiter. The control delimiter corresponding to `control` is called `prompt` (alias #). The control delimiter corresponding to `shift` is called `reset` (alias $\langle\cdot\rangle$) and its continuation-passing counterpart is a classical backtracking idiom in functional programming [21]. Other, more advanced, delimited-control operators exist [12, 14]; we return to them in the conclusion.

In the present work, we focus on `shift` and `control`.

Email addresses: `dabi@brics.dk` (Dariusz Biernacki), `danvy@brics.dk` (Olivier Danvy), `ccshan@post.harvard.edu` (Chung-chieh Shan).

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

Overview: In Section 2, we present an environment-based abstract machine that specifies the behaviors of `shift` and `control`, and we show how the extent of a `shift`-abstracted delimited continuation is static whereas that of a `control`-abstracted delimited continuation is dynamic. In Section 3, we present an array of solutions to the traditional samefringe example and to its breadth-first counterpart, using Filinski's implementation of `shift` and `reset` in ML [11] and Shan's subsequent implementation of `control` and `prompt` in Scheme [18]. Filinski's implementation takes the form of an ML functor mapping the type of intermediate answers to a structure containing an instance of the control operators at that type:

```
signature SHIFT_AND_RESET
= sig
    type intermediate_answer
    val shift : (('a -> intermediate_answer) -> intermediate_answer) -> 'a
    val reset : (unit -> intermediate_answer) -> intermediate_answer
  end
```

For the purpose of this article, we have adapted Shan's implementation from Scheme macros to an ML functor with a similar signature:

```
signature CONTROL_AND_PROMPT
= sig
    type intermediate_answer
    val control : (('a -> intermediate_answer) -> intermediate_answer) -> 'a
    val prompt : (unit -> intermediate_answer) -> intermediate_answer
  end
```

Prerequisites and preliminaries: Besides some awareness of CPS and the CPS transformation [6, 16, 19], we assume a passing familiarity with defunctionalization [7, 17]. Our programming language of discourse is Standard ML.

2 An operational characterization

In our previous work [2], we derived an environment-based abstract machine for the λ-calculus with `shift` and `reset` by defunctionalizing its definitional interpreter [6]. We use this abstract machine to explain the static extent of the delimited continuations abstracted by `shift` and the dynamic extent of the delimited continuations abstracted by `control`.

2.1 An abstract machine for `shift` and `reset`

The abstract machine is displayed in Figure 1. The set of possible values consists of closures and captured contexts. The machine extends Felleisen et al.'s CEK machine [9] with a meta-context C_2, the two transitions for $\langle\cdot\rangle$ and $\mathcal{S}$, and the transition for applying a captured context to a value in a context and a meta-context. Intuitively, a context represents the rest of the computation up to the nearest enclosing delimiter, and a meta-context represents all of the remaining computation [5]. We describe this machine in more detail in an extended version of this article [4].

2.2 An abstract machine for `control` and `prompt`

Unlike `shift` and `reset`, whose definition is based on CPS, `control` and `prompt` are specified by representing delimited continuations as a list of stack frames and their composition as the concatenation of these representations [10]. Such a concatenation function $\star$ is defined as follows:

$$\begin{aligned} \mathsf{END} \star C_1' &= C_1' \\ (\mathsf{ARG}\,((t,e),\ C_1)) \star C_1' &= \mathsf{ARG}\,((t,e),\ C_1 \star C_1') \\ (\mathsf{FUN}\,(v,\ C_1)) \star C_1' &= \mathsf{FUN}\,(v,\ C_1 \star C_1') \end{aligned}$$

- Terms: $t ::= x \mid \lambda x.t \mid t_0\, t_1 \mid \langle t \rangle \mid \mathcal{S}k.t$
- Values (closures and captured continuations): $v ::= [x,\, t,\, e] \mid C_1$
- Environments: $e ::= e_{\mathit{empty}} \mid e[x \mapsto v]$
- Contexts: $C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,((t,e),\; C_1) \mid \mathsf{FUN}\,(v,\; C_1)$
- Meta-contexts: $C_2 ::= \bullet \mid C_1 \cdot C_2$
- Initial transition, transition rules, and final transition:

$$
\begin{array}{|rcl|}
\hline
t & \Rightarrow & \langle t,\, e_{\mathit{empty}},\, \mathsf{END},\, \bullet \rangle_{\mathit{eval}} \\
\hline
\langle x,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, e\,(x),\, C_2 \rangle_{\mathit{cont_1}} \\
\langle \lambda x.t,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, [x,\, t,\, e],\, C_2 \rangle_{\mathit{cont_1}} \\
\langle t_0\, t_1,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}} & \Rightarrow & \langle t_0,\, e,\, \mathsf{ARG}\,((t_1, e),\; C_1),\, C_2 \rangle_{\mathit{eval}} \\
\langle \langle t \rangle,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}} & \Rightarrow & \langle t,\, e,\, \mathsf{END},\, C_1 \cdot C_2 \rangle_{\mathit{eval}} \\
\langle \mathcal{S}k.t,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}} & \Rightarrow & \langle t,\, e[k \mapsto C_1],\, \mathsf{END},\, C_2 \rangle_{\mathit{eval}} \\
\hline
\langle \mathsf{END},\, v,\, C_2 \rangle_{\mathit{cont_1}} & \Rightarrow & \langle C_2,\, v \rangle_{\mathit{cont_2}} \\
\langle \mathsf{ARG}\,((t, e),\; C_1),\, v,\, C_2 \rangle_{\mathit{cont_1}} & \Rightarrow & \langle t,\, e,\, \mathsf{FUN}\,(v,\; C_1),\, C_2 \rangle_{\mathit{eval}} \\
\langle \mathsf{FUN}\,([x,\, t,\, e],\; C_1),\, v,\, C_2 \rangle_{\mathit{cont_1}} & \Rightarrow & \langle t,\, e[x \mapsto v],\, C_1,\, C_2 \rangle_{\mathit{eval}} \\
\langle \mathsf{FUN}\,(C_1',\; C_1),\, v,\, C_2 \rangle_{\mathit{cont_1}} & \Rightarrow & \langle C_1',\, v,\, C_1 \cdot C_2 \rangle_{\mathit{cont_1}} \\
\hline
\langle C_1 \cdot C_2,\, v \rangle_{\mathit{cont_2}} & \Rightarrow & \langle C_1,\, v,\, C_2 \rangle_{\mathit{cont_1}} \\
\hline
\langle \bullet,\, v \rangle_{\mathit{cont_2}} & \Rightarrow & v \\
\hline
\end{array}
$$

Fig. 1. A call-by-value environment-based abstract machine for the λ-calculus extended with `shift` ($\mathcal{S}$) and `reset` ($\langle\cdot\rangle$)

It is then simple to modify the abstract machine to compose delimited continuations by concatenating their representation: in Figure 1, one merely replaces the transition applying a captured context C_1' by pushing the current context C_1 onto the meta-context C_2, i.e.,

$$\boxed{\langle \mathsf{FUN}\,(C_1',\; C_1),\, v,\, C_2 \rangle_{\mathit{cont_1}} \;\Rightarrow\; \langle C_1',\, v,\, C_1 \cdot C_2 \rangle_{\mathit{cont_1}}}$$

with a transition that applies a captured context C_1' by concatenating it with the current context C_1:

$$\boxed{\langle \mathsf{FUN}\,(C_1',\; C_1),\, v,\, C_2 \rangle_{\mathit{cont_1}} \;\Rightarrow\; \langle C_1' \star C_1,\, v,\, C_2 \rangle_{\mathit{cont_1}}}$$

This change gives `shift` the behavior of `control`. In contrast, `reset` and `prompt` have the same definition. The rest of the machine does not change.

In our previous work [2, Sec. 4.5], we have pointed out that the dynamic behavior of `control` is incompatible with CPS because the modified abstract machine no longer corresponds to a defunctionalized continuation-passing evaluator [7]. Indeed `shift` is static, whereas `control` is dynamic in the following sense:

- `shift` captures a delimited continuation in a representation C_1 that remains distinct from the current context C_1', when it is applied. Consequently, the current context C_1' *cannot* be accessed from C_1 by another use of `shift`.
- `control` captures a delimited continuation in a representation C_1 that grafts itself to the current context C_1', when it is applied. Consequently, the current context C_1' *can* be accessed from C_1 by another use of `control`.

This difference of behavior can be observed with delimited continuations that, when applied, capture the current continuation. A `control`-abstracted delimited continuation dynamically captures the current continuation, above and beyond its point of activation, whereas a `shift`-given delimited continuation statically captures the current continuation up to its point of resumption.

3 The samefringe problem

We present a spectrum of solutions to the samefringe problem, both in its traditional depth-first form and in its breadth-first counterpart. We work on Lisp-like binary trees of integers (S-expressions):

```
datatype tree = LEAF of int | NODE of tree * tree
```

The samefringe problem is traditionally stated as follows. Given two trees of integers, one wants to know whether they have the same sequence of leaves when read from left to right. Let us consider, for example, the two following trees:

These two trees arise from evaluating `NODE (NODE (LEAF 1, LEAF 2), LEAF 3)` and `NODE (LEAF 1, NODE (LEAF 2, LEAF 3))`. Even though they are shaped differently, they have the same fringe `[1, 2, 3]` (representing it as a list). Computing a fringe is done by traversing a tree depth-first and from left to right.

By analogy, we also address the breadth-first counterpart of the samefringe problem. Given two trees of integers, we want to know whether they have the same fringe when traversed in left-to-right breadth-first order. For example, the breadth-first fringe of the left tree just above is `[3, 1, 2]` and that of the right tree just above is `[1, 2, 3]`.

We express the samefringe function generically by abstracting the representation of sequences of leaves with a data type `sequence` and a notion of computation (to compute the next element in a sequence):

```
signature GENERATOR
= sig
    type 'a computation
    datatype sequence = END | NEXT of int * sequence computation
    val make_sequence : tree -> sequence
    val compute : sequence computation -> sequence
  end
```

Given a generator satisfying this signature, we can write a samefringe function that maps two given trees into two sequences of integers and iteratively traverses these sequences, stopping as soon as one of the two sequences is exhausted or two differing leaves are found.

3.1 Depth first

3.1.1 A lazy traversal: The usual solution to the samefringe problem is to construct the sequences lazily and to traverse them on demand. In the following generator, the data type `sequence` implements lazy sequences; the construction of the rest of the lazy sequence is delayed with a thunk of type `unit -> sequence`; and `make_sequence` is defined as an accumulator-based flatten function:

```
structure Lazy_generator : GENERATOR
= struct
    datatype sequence = END | NEXT of int * sequence computation
    withtype 'a computation = unit -> 'a

    (*  visit : tree * sequence computation -> sequence  *)
    fun visit (LEAF i, a)
        = NEXT (i, a)
      | visit (NODE (t1, t2), a)
        = visit (t1, fn () => visit (t2, a))

    fun make_sequence t = visit (t, fn () => END)

    fun compute thunk = thunk ()
  end
```

The construction of the sequence in `Lazy_generator` and the comparisons in `same_ fringe` are interleaved. This choice is known to be efficient because if two leaves differ, the remaining two sequences are not built at all.

3.1.2 A continuation-based traversal: Alternatively to viewing the thunk of type `unit -> sequence`, in the lazy traversal of Section 3.1.1, as a functional device to implement laziness, we can view it as a delimited continuation that is initialized in the initial call to `visit` in `make_sequence`, extended in the induction case of `visit`, captured in the base case of `visit`, and resumed in `compute`. From that viewpoint, the lazy traversal is also a continuation-based one.

3.1.3 A direct-style traversal with `shift` and `reset`: In direct style, the initialization of the delimited continuation `a` of Section 3.1.2 is obtained with the control delimiter `reset`, its extension is obtained by functional sequencing, its capture is obtained with the delimited-control operator `shift`, and its resumption is obtained by function application.

Using Filinski's functor `Shift_and_Reset`, one can therefore define the lazy generator in direct style as follows:

```
structure Lazy_generator_with_shift_and_reset : GENERATOR
= struct
    datatype sequence = END | NEXT of int * sequence computation
    withtype 'a computation = unit -> 'a

    local structure SR = Shift_and_Reset (type intermediate_answer = sequence)
    in val shift = SR.shift
       val reset = SR.reset
    end

    (*  visit : tree -> unit  *)
    fun visit (LEAF i)
        = shift (fn a => NEXT (i, a))
      | visit (NODE (t1, t2))
        = let val () = visit t1 in visit t2 end

    fun make_sequence t = reset (fn () => let val () = visit t in END end)

    fun compute thunk = thunk ()
  end
```

CPS-transforming `visit` and `make_sequence` yields the definitions of Section 3.1.1. The key point of this CPS transformation is that given a continuation `k`, the expression `let val () = visit t1 in visit t2 end` is transformed into `visit (t1, fn () => visit (t2, k))`.

3.1.4 A stack-based traversal: Alternatively to writing the lazy solution in direct style, we can defunctionalize its computation (which has type `sequence computation`, i.e., `unit -> sequence`) and obtain a first-order solution [7, 17]. The inhabitants of the function space `unit -> sequence` are instances of the function abstractions in the initial call to `visit` (i.e., `fn () => END`) and in the induction case of `visit` (i.e., `fn () => visit (t2, a)`). We therefore represent this function space by (1) a sum corresponding to these two possibilities, and (2) the corresponding apply function, `continue`, to interpret each of the summands. We represent this sum with an ML data type, which is recursive because of the recursive call to `visit`. This data type is isomorphic to that of a list of subtrees, which we use for simplicity in the code below. The result is essentially McCarthy's solution [13]:

```
structure Lazy_generator_stack_based : GENERATOR
= struct
    datatype sequence = END | NEXT of int * sequence computation
    withtype 'a computation = tree list

    (*  visit : tree * tree list -> sequence  *)
    fun visit (LEAF i, a)
        = NEXT (i, a)
      | visit (NODE (t1, t2), a)
        = visit (t1, t2 :: a)
    (*  continue : tree list * unit -> sequence  *)
    and continue (nil, ())
        = END
      | continue (t :: a, ())
        = visit (t, a)

    fun make_sequence t = visit (t, nil)

    fun compute a = continue (a, ())
  end
```

This solution traverses a given tree incrementally by keeping a stack of its subtrees. To make this point more explicit, and as a stepping stone towards breadth-first traversal, let us fold the definition of `continue` in the induction case of `visit` so that `visit` always calls `continue`:

```
| visit (NODE (t1, t2), a)
  = continue (t1 :: t2 :: a, ())
```

(Unfolding the call to `continue` gives back the definition above.)

We now clearly have a stack-based definition of depth-first traversal, and furthermore we have shown that this stack corresponds to the continuation of a function implementing a recursive descent.

3.2 Breadth first

3.2.1 A queue-based traversal: Replacing the (last-in, first-out) stack, in the definition of Section 3.1.4, by a (first-in, first-out) queue yields a definition that implements breadth-first, rather than depth-first, traversal:

```
structure Lazy_generator_queue_based : GENERATOR
= struct
    datatype sequence = END | NEXT of int * sequence computation
    withtype 'a computation = tree list
```

```
    (*  visit : tree * tree list -> sequence  *)
    fun visit (LEAF i, a)
        = NEXT (i, a)
      | visit (NODE (t1, t2), a)
        = continue (a @ [t1, t2], ())
    (*  continue : tree list * unit -> sequence  *)
    and continue (nil, ())
        = END
      | continue (t :: a, ())
        = visit (t, a)

    fun make_sequence t = visit (t, nil)

    fun compute a = continue (a, ())
  end
```

In contrast to Section 3.1.4, where the clause for nodes was (essentially) concatenating the two subtrees in front of the list of subtrees

```
| visit (NODE (t1, t2), a)
  = continue ([t1, t2] @ a, ())   (* then *)
```

the clause for nodes is concatenating the two subtrees in the back of the list of subtrees:

```
| visit (NODE (t1, t2), a)
  = continue (a @ [t1, t2], ())   (* now *)
```

Nothing else changes in the definition of the generator. In particular, subtrees are still removed from the front of the list of subtrees by `continue`. With this last-in, first-out policy, the generator yields a sequence in breadth-first order.

Because the `::`-constructors of the list of subtrees are not solely consumed by `continue` but also by `@`, this definition *is not in the range of defunctionalization* [2, 7]. Therefore, even though `visit` is tail-recursive and constructs a data structure that is interpreted in `continue`, it does not correspond to a continuation-passing function. And indeed, it is well-known that traversing an inductive data structure breadth-first does not mesh with the visitor pattern of functional programming, i.e., compositional recursive descent (catamorphism).

3.2.2 A direct-style traversal with `control` and `prompt`: The critical operation in the definition of `visit`, in Section 3.2.1, is the enqueuing of the subtrees `t1` and `t2` to the current queue `a`, which is achieved by the list concatenation `a @ [t1, t2]`. We observe that this concatenation matches the concatenation of stack frames in the specification of `control` in Section 2.2.

Therefore—and this is a eureka step—one can write `visit` in direct style using `control` and `prompt`. To this end, we represent both queues `a` and `[t1, t2]` as dynamic delimited continuations in such a way that their composition represents the concatenation of `a` and `[t1, t2]`. The direct-style traversal reads as follows:

```
structure Lazy_generator_with_control_and_prompt : GENERATOR
= struct
    datatype sequence = END | NEXT of int * sequence computation
    withtype 'a computation = unit -> 'a

    local structure CP = Control_and_Prompt (type intermediate_answer = sequence)
    in val control = CP.control
       val prompt = CP.prompt
    end
```

```
  (*  visit : tree -> unit  *)
  fun visit (LEAF i)
      = control (fn a => NEXT (i, a))
    | visit (NODE (t1, t2))
      = control (fn a => let val END = a ()
                             val () = visit t1
                             val () = visit t2
                         in END
                         end)

  fun make_sequence t
      = prompt (fn () => let val () = visit t
                         in END
                         end)

  fun compute a = prompt (fn () => a ())
end
```

In the induction case, the current delimited continuation (representing the current control queue) is captured, bound to `a`, and applied to `()`. The implicit continuation of this application visits `t1` and then `t2`, and therefore represents the queue `[t1, t2]`. Applying `a` seals it to the implicit continuation so that any continuation captured by a subsequent recursive call to `visit` in `a` captures both the rest of `a` and the traversal of `t1` and `t2`, i.e., the rest of the new control queue.

3.3 Summary and conclusion

We first have presented a spectrum of solutions to the traditional samefringe problem. The one using `shift` and `reset` is new. We believe that connecting the lazy solution with McCarthy's stack-based solution by defunctionalization is new as well.

By replacing the stack with a queue in the stack-based program, we then have obtained a solution to the breadth-first counterpart of the samefringe problem. Viewing this queue as a 'data-structure continuation' [22, page 179], we have observed that the operations upon it correspond to the operations induced by the composition of dynamic delimited continuations. We have then re-expressed this program using `control` and `prompt`.

In the induction clause of `visit` in Section 3.2.2, if we returned *after* visiting `t1` and `t2` instead of before,

```
  | visit (NODE (t1, t2))
    = control (fn a => let val () = visit t1
                           val () = visit t2
                       in a ()
                       end)
```

we would obtain depth-first traversal. This modified clause can be simplified into

```
  | visit (NODE (t1, t2))
    = let val () = visit t1
      in visit t2
      end
```

which coincides with the corresponding clause in Section 3.1.3. The resulting pattern of use of `control` and `prompt` in the modified definition is the traditional one used to simulate `shift` and `reset` [3].

It is therefore simple to program depth-first traversal with `control` and `prompt`. But conversely, obtaining a breadth-first traversal using `shift` and `reset` would require a far less simple encoding of `control` and `prompt` in terms of `shift` and `reset` [18].

4 Conclusion and issues

Over the last 15 years, it has been repeatedly claimed that `control` has more expressive power than `shift`. Even though this claim is now disproved [18], it is still unclear how to program with `control`-like dynamic delimited continuations. In fact, in 15 years, only toy examples have been advanced to illustrate the difference between static and dynamic delimited continuations.

In this article, we have filled this vacuum by using dynamic delimited continuations to program breadth-first traversal. We have accounted for the dynamic queuing mechanism inherent to breadth-first traversal with the dynamic concatenation of stack frames that is specific to `control` and that makes it go beyond what is traditionally agreed upon as being continuation-passing style (CPS). We have presented one simple example of breadth-first traversal: the breadth-first counterpart of the traditional samefringe function. The extended version of this article [4] contains another example, breadth-first numbering [15], and we have recently proposed yet another one [2, page 20] [3, page 5].

Since `control`, even more dynamic delimited-control operators have been proposed [12, 14], all of which go beyond CPS but only two of which, to the best of our knowledge, come with motivating examples illustrating their specificity:

- In his PhD thesis [1], Balat uses the extra expressive power of Gunter, Rémy, and Riecke's control operators `set` and `cupto` over that of `shift` and `reset` to prototype a type-directed partial evaluator for the lambda-calculus with sums.

- In his PhD thesis [14], Nanevski introduces two new dynamic delimited-control operators, `mark` and `recall`, and illustrates them with a function partitioning a natural number into the lists of natural numbers that add to it. He considers both depth-first and breadth-first generation strategies, and conjectures that the latter cannot be written using `shift` and `reset`. His is thus our closest related work.

These applications are rare and so far they tend to be daunting. Dynamic delimited continuations need simpler examples, more reasoning tools, and more program transformations.

Acknowledgments: We are grateful to Mads Sig Ager, Małgorzata Biernacka, Andrzej Filinski, Julia Lawall, Kevin Millikin and the anonymous referees for their comments.

The second author would also like to thank Andrzej Filinski, Mayer Goldberg, Julia Lawall, and Olin Shivers for participating to a one-week brain storm about continuations in August 2004, and to BRICS and DAIMI for hosting us during that week.

This work is partially supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`), the Danish Natural Science Research Council, Grant no. 21-03-0545, and the United States National Science Foundation Grant no. BCS-0236592.

References

[1] V. Balat. *Une étude des sommes fortes: isomorphismes et formes normales.* PhD thesis, PPS, Université Denis Diderot (Paris VII), Paris, France, Dec. 2002.

[2] M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the CPS hierarchy (revised version). Technical Report BRICS RS-05-11, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Mar. 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004).

[3] D. Biernacki and O. Danvy. A simple proof of a folklore theorem about delimited control. Technical Report BRICS RS-05-10, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Mar. 2005.

[4] D. Biernacki, O. Danvy, and C. Shan. On the dynamic extent of delimited continuations. Technical Report BRICS RS-05-13, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Apr. 2005. Extended version of the present article.

[5] O. Danvy. On evaluation contexts, continuations, and the rest of the computation. In H. Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, Jan. 2004. Invited talk.

[6] O. Danvy and A. Filinski. Abstracting control. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[7] O. Danvy and L. R. Nielsen. Defunctionalization at work. In H. Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, Sept. 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[8] M. Felleisen. The theory and practice of first-class prompts. In J. Ferrante and P. Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, Jan. 1988. ACM Press.

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

[10] M. Felleisen, M. Wand, D. P. Friedman, and B. F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In R. C. Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988. ACM Press.

[11] A. Filinski. Representing monads. In H.-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, Jan. 1994. ACM Press.

[12] C. Gunter, D. Rémy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In S. Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 12–23, La Jolla, California, June 1995. ACM Press.

[13] J. McCarthy. Another samefringe. *SIGART Newsletter*, 61, Feb. 1977.

[14] A. Nanevski. *Functional Programming with Names and Necessity.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 2004. Technical Report CMU-CS-04-151.

[15] C. Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In P. Wadler, editor, *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 35, No. 9, pages 131–136, Montréal, Canada, Sept. 2000. ACM Press.

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

[17] J. 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.

[18] C. Shan. Shift to control. In O. Shivers and O. Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Snowbird, Utah, Sept. 2004.

[19] G. 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.

[20] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974), with a foreword.

[21] G. J. Sussman and G. L. Steele Jr. Scheme: An interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998. Reprinted from the AI Memo 349, MIT (1975), with a foreword.

[22] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, Jan. 1980.