On the Static and Dynamic Extents of Delimited Continuations*

Dariusz Biernacki & Olivier Danvy
BRICS[†]
Department of Computer Science
University of Aarhus[‡]

Chung-chieh Shan
Department of Computer Science
Rutgers University[§]

December 10, 2005 at 18:00 (DK time)

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.

We also point out where static delimited continuations naturally give rise to the notion of control stack whereas dynamic delimited continuations can be made to account for a notion of 'control queue.'

Keywords

Delimited continuations, direct style, continuation-passing style (CPS), CPS transformation, defunctionalization, control operators, shift and reset, control and prompt.

*To appear in Science of Computer Programming.
A substantially shorter version appeared in Information Processing Letters [?].
†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: `{dabi,danvy}@brics.dk`
§110 Frelinghuysen Road, Piscataway, NJ 08854, USA
Email: `ccshan@cs.rutgers.edu`

Contents

1 Introduction

To distinguish between the static extent and the dynamic extent of delimited continuations, we first need to review the notions of continuation and of delimited continuation.

1.1 Background

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 [38, 54, 61, 65, 73]. While this format has been an active topic of study [5, 6, 9, 28, 35, 37, 40, 49, 51, 57, 59, 62, 66, 69, 70, 76], it also has been felt as a straightjacket both from a semantics point of view [28, 29, 31, 32, 44, 45, 71] and from a programming point of view [18, 19, 21, 22], where one would like to relax the tail-call constraint and compose continuations.

In direct style, continuations are accessed with a variety of control operators such as Landin's `J` [50], Reynolds's `escape` [65], Scheme's `call/cc` [17, 46], and Standard ML of New Jersey's `callcc` and `throw` [26]. 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 [73, 74].

In direct style, delimited continuations are also accessed with control operators such as Felleisen et al.'s `control` (alias $\mathcal{F}$) [28, 31, 32, 71] and Danvy and Filinski's `shift` (alias $\mathcal{S}$) [21–23]. 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 [28, 29]: 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 [1, 14, 16, 53, 67, 75], one that is currently enjoying a renewal of interest [10, 24, 43, 48, 72, 78, 82]. Other, more advanced, delimited-control operators exist [27, 39, 42, 56, 58, 64]; we return to them in the conclusion.

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

1.2 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. We show how `shift` can be trivially simulated in terms of `control` and `prompt`, which is a well-known result [11], and we review recently discovered simulations of control and prompt in terms of shift and reset [12, 47, 68]. In Section 3, we present a roadmap of Sections 4 and 5, where we show how the static extent of a delimited continuation is compatible with a control stack and depth-first traversal, and how the dynamic extent of a delimited continuation can be made to account for a 'control queue' and breadth-first traversal.

Prerequisites and preliminaries: Besides some awareness of CPS and the CPS transformation [23, 61, 73], we assume a passing familiarity with defunctionalization [25, 65].

Our programming language of discourse is Standard ML [55]. In the following sections, we will make use of the notational equivalence of expressions such as

```
x1 :: x2 :: xs
(x1 :: x2 :: nil) @ xs
[x1, x2] @ xs
```

where `::` denotes infix list construction and `@` denotes infix list concatenation. In an environment where `x1` denotes `1`, `x2` denotes `2`, and `xs` denotes `[3, 4, 5]`, each of the three expressions above evaluates to `[1, 2, 3, 4, 5]`.

2 An operational characterization

In our previous work [7], we derived an environment-based abstract machine for the λ-calculus with `shift` and `reset` by defunctionalizing the corresponding definitional interpreter [22]. 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; `reset` is noted $\langle\cdot\rangle$ and `shift` is noted $\mathcal{S}$. The set of possible values consists of closures and captured contexts. The machine extends Felleisen et al.'s CEK machine [30] 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 an evaluation context and a meta-context. Intuitively, an evaluation context represents the rest of the computation up to the nearest enclosing delimiter, and a meta-context represents all of the remaining computation [20].

Given a term t, the machine is initialized in an *eval*-state with an empty environment e_{empty}, an empty context END, and an empty meta-context $\bullet$. The transitions out of an *eval*-state are defined by cases on its first component:

- a variable x is looked up in the current environment and the machine switches to a $cont_1$-state;
- an abstraction $\lambda x.t$ is evaluated into a closure $[x,\ t,\ e]$ and the machine switches to a $cont_1$-state;

- 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: $e ::= e_{empty} \mid e[x \mapsto v]$
- Evaluation 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:

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

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

- an application $t_0\, t_1$ is processed by pushing t_1 and the environment onto the context and switching to a new *eval*-state to process t_0;
- a reset-expression $\langle t\rangle$ is processed by pushing the current context on the current meta-context and switching to a new *eval*-state to process t in an empty context, as an intermediate computation;
- a shift-expression $\mathcal{S}k.t$ is processed by capturing the context C_1 and binding it to k, and switching to a new *eval*-state to process t in an empty context.

The transitions of a $cont_1$-state are defined by cases on its first component:

- an empty context END specifies that an intermediate computation is completed; it is processed by switching to a $cont_2$-state;
- a context $\mathsf{ARG}\,((t, e),\ C_1)$ specifies the evaluation of an argument; it is processed by switching to an *eval*-state to process t in a new context;
- a context $\mathsf{FUN}\,([x,\ t,\ e],\ C_1)$ specifies the application of a closure; it is processed by switching to an *eval*-state to process the term t with an extension of the environment e;
- a context $\mathsf{FUN}\,(C_1',\ C_1)$ specifies the application of a captured context; it is processed by pushing C_1 on top of the meta-context and switching to a new $cont_1$-state to process C_1'.

The transitions of a $cont_2$-state are defined by cases on its first component:

- an empty meta-context $\bullet$ specifies that the overall computation is completed; it is processed as a final transition;
- a non-empty meta-context specifies that the overall computation is not completed; $C_1 \cdot C_2$ is processed by switching to a $cont_1$-state to process C_1.

All in all, this abstract machine is a straight defunctionalized continuation-passing evaluator [7, 22].

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 [28, 32]. Such a concatenation function $\star$ is defined as follows:

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

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 that applies a captured context C_1' by pushing the current context C_1 onto the meta-context C_2, i.e.,

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

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

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

This change gives $\mathcal{S}$ (alias `shift`) the behavior of $\mathcal{F}$ (alias `control`). In contrast, $\langle\cdot\rangle$ (alias `reset`) and $\#$ (alias `prompt`) have the same definition. The rest of the machine does not change.

In our previous work [7, Section 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 [25]. Indeed `shift` is static, whereas `control` is dynamic, in the following sense:

- `shift` captures a delimited continuation in a representation C_1 that, when applied, remains distinct from the current context C_1'. Consequently, the current context C_1' *cannot* be accessed from C_1 by another use of `shift`. (An analogy: in a statically scoped programming language, the environment of an application remains distinct from the environment of the applied function. A non-local variable in the function refers to the environment of its definition. Consequently, the environment of a function application cannot be accessed before the function completes.)

- `control` captures a delimited continuation in a representation C_1 that, when applied, grafts itself to the current context C_1'. Consequently, the current context C_1' *can* be accessed from C_1 by another use of `control`. (An analogy: in a dynamically scoped programming language, the environment of an application is extended with the environment of the applied function. A non-local variable in the function refers to the environment of its application. Consequently, the environment of a function application can be accessed before the function completes.)

This difference of extent can be observed with delimited continuations that, when applied, capture the current continuation [8, Section 5] [21, Section 6.1] [23, Section 5.3] [32, Section 4]. A `control`-abstracted delimited continuation dynamically captures the current continuation, above and beyond its point of activation, whereas a `shift`-abstracted delimited continuation statically captures the current continuation up to its point of activation.

2.3 Simulating `shift` in terms of `control` and `prompt`

It is simple to obtain the effect of `shift` using `control`: for each captured continuation k, every occurrence of $k\,v$ should be replaced by $\#(k\,v)$ when v is a value, and every other occurrence of k should be replaced with $\lambda x.\#(k\,x)$. (In ML, for each captured continuation `k`, every occurrence of `k v` should be replaced by `prompt (fn () => k v)` when `v` denotes a value, and every other occurrence of `k` should be replaced with `fn x => prompt (fn () => k x)`.)

This way, when k (i.e., some context C_1') is applied, the context of its application is always END and it is a consequence of the definition of $\star$ that $C_1' \star \mathsf{END} = C_1'$. The two first authors have recently given a formal proof of the correctness of this simulation [11].

2.4 Simulating `control` in terms of `shift` and `reset`

Recently it has been shown that `control` and `prompt` can be expressed in terms of `shift` and `reset`, which unexpectedly proves that `shift` is actually as expressive as `control`.

- In his previous article [68], Shan presented a simulation that is based on his observation that dynamic continuations are recursive. His simulation keeps (as a piece of mutable state) the context in which a `control`-captured delimited continuation is applied. This simulation is untyped and implemented in Scheme.

- In their recent article [12], Biernacki, Danvy, and Millikin presented a new simulation that is based on a 'Dynamic Continuation-Passing Style' (DCPS) for dynamic delimited continuations. Their idea is to use a trail of continuations to represent the context in which a `control`-captured delimited continuation is applied, and to compose continuations by concatenating such trails of continuations. This simulation is typed and implemented in ML.

- In his recent article [47], Kiselyov proposed a new simulation that is based on trampolining. In order to let a `control`-captured continuation access the context where it is applied, he reifies such an access in a sum type interpreted by `prompt`. This simulation is untyped and implemented in Scheme.

Concomitant with each solution is a CPS transformation for `control` and `prompt` that conservatively extends the usual call-by-value CPS transformation for the λ-calculus, with the requirement that continuations be recursive (or more precisely, that their answer type be higher-order and recursive).

In Appendix B, we present Shan's implementation of `control` and `prompt` in Standard ML of New Jersey [68]. This implementation is based on Filinski's implementation of `shift` and `reset` in SML [34], which we present in Appendix A. Filinski's implementation takes the form of a functor mapping the type of intermediate answers to a structure containing an instance of `shift` and `reset` 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
```

Likewise, our implementation takes the form of a functor mapping the type of intermediate answers to a structure containing an instance of `control` and `prompt` at that type:

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

2.5 Three examples in ML

Using the implementation of `shift` and `reset` (Appendix A), and of `control` and `prompt` (Appendix B), we present three simple examples illustrating the difference between `shift` and `control`. Let us fix the type of intermediate answers to be `int`:

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

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

The following ML expression

```
reset
 (fn () => shift (fn k => 10 + (k 100))
           + shift (fn k' => 1))
```

evaluates to `11`, whereas (replacing `reset` by `prompt` and `shift` by `control`)

```
prompt
 (fn () => control (fn k => 10 + (k 100))
           + control (fn k' => 1))
```

evaluates to `1` and (delimiting the application of `k` with `prompt`)

```
prompt
 (fn () => control (fn k => 10 + prompt (fn () => k 100))
           + control (fn k' => 1))
```

evaluates to `11`.

In the first case, `shift (fn k => 10 + (k 100))` is evaluated with a continuation that could be written functionally as `fn v => v + shift (fn k' => 1)`. When `k` is applied, the expression `shift (fn k' => 1)` is evaluated in a context that could be represented functionally as `fn v => 100 + v` and in a meta-context that could be represented as `(fn v => 10 + v) :: nil`; this context is captured and discarded, and the intermediate answer is `1`; this intermediate answer is plugged into the top context from the meta-context, i.e., `fn v => 10 + v` is applied to `1`; the next intermediate answer is `11`; and it is the final answer since the meta-context is empty.

In the second case, `control (fn k => 10 + (k 100))` is evaluated with a continuation that could be written functionally as `fn v => v + control (fn k' => 1)`. When `k` is applied, the expression `control (fn k' => 1)` is evaluated in a context that results from composing `fn v => 10 + v` and `fn v => 100 + v` (and therefore could be represented functionally as `fn v => 10 + (100 + v)`), and in a meta-context which is empty; this context is captured and discarded, and the intermediate answer is `1`; and it is the final answer since the meta-context is empty.

In the third case, `control (fn k => 10 + prompt (fn () => k 100))` is evaluated with a continuation that could be written functionally as `fn v => v + control (fn k' => 1)`. When `k` is applied, the expression `control (fn k' => 1)` is evaluated in a context that results from composing `fn v => v` and `fn v => 100 + v` (and therefore could be represented functionally as `fn v => 100 + v`), and in a meta-context which could be represented as `(fn v => 10 + v) :: nil`; this context is captured and discarded, and the intermediate answer is `1`; this intermediate answer is plugged into the top context from the meta-context, i.e., `fn v => 10 + v` is applied to `1`; the next intermediate answer is `11`; and it is the final answer since the meta-context is empty.

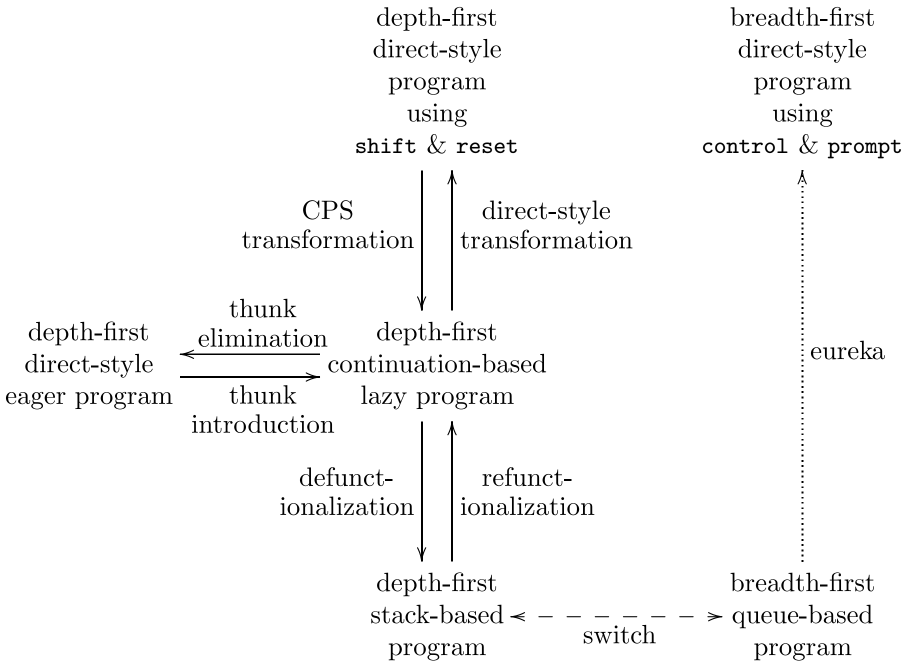
The CPS counterpart of the first ML expression above reads as follows:

```
let val k = fn v => let val k' = fn v' => v + v'
                    in 1
                    end
in 10 + (k 100)
end
```

No such simple functional encoding exists for the second and third ML expressions above [12].

3 Programming with delimited continuations

In Section 4, we present an array of solutions to the traditional samefringe example and to its breadth-first counterpart. In Section 5, we present an array of solutions to Okasaki's breadth-first numbering pearl and to its depth-first counterpart. In both sections, the presentation is structured according to the following diagram:



- Our starting point here is a direct-style eager program (left side of the diagram). We can make this program lazy by using thunks, i.e., functions of type `unit -> 'a` (center of the diagram).
- We can then defunctionalize the thunks in the lazy program, obtaining a stack-based program (bottom center of the diagram).

- Alternatively, we can view the type `unit -> 'a` not as a functional device to implement laziness but as a delimited continuation. The lazy program is then, in actuality, a continuation-based one, and one that is the CPS counterpart of a direct-style program using `shift` and `reset` (top center of the diagram).

- The stack-based program (bottom center of the diagram) implements a depth-first traversal. Replacing the stack with a queue yields a program implementing a breadth-first traversal (bottom right of the diagram).

- By analogy with the rest of the diagram, we infer the direct-style program using `control` and `prompt` (top right of the diagram) from this queue-based program.

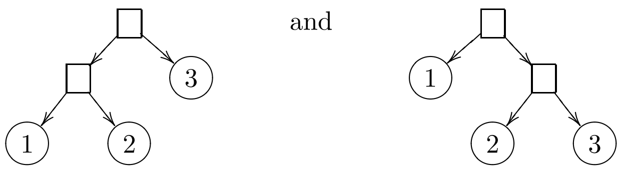
The three nodes in the center of the diagram—the CPS program, its direct-style counterpart, and its defunctionalized counterpart—follow the transformational tradition established in Reynolds's and Wand's seminal articles about continuations [65, 80]. In particular the 'data-structure continuation' [80, page 179] of the depth-first program is a stack. By analogy, the data-structure continuation of the breadth-first program is a queue. We conjecture that the queue-based program could be mechanically obtained from the direct-style one by some kind of 'abstract CPS transformation' [32, 63], but fleshing out this conjecture falls out of the scope of the present article [12].

4 The samefringe problem

We present a spectrum of solutions to the traditional depth-first samefringe problem and 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. For example, the two trees `NODE (NODE (LEAF 1, LEAF 2), LEAF 3)` and `NODE (LEAF 1, NODE (LEAF 2, LEAF 3))` have the same fringe `[1, 2, 3]` (representing it as a list) even though they are shaped differently:



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
```

The following functor maps a representation of sequences of leaves to a structure containing the samefringe function. Given two trees, `same_fringe` maps them into two sequences of integers (with `make_sequence`) and iteratively traverses these sequences with an auxiliary `loop` function. This function stops as soon as one of the two sequences is exhausted or two differing leaves are found:

```
functor make_Same_Fringe (structure G : GENERATOR)
= struct
    (*  same_fringe : tree * tree -> bool  *)
    fun same_fringe (t1, t2)
        = let (*  loop : G.sequence * G.sequence -> bool  *)
              fun loop (G.END, G.END)
                  = true
                | loop (G.NEXT (i1, a1), G.NEXT (i2, a2))
                  = i1 = i2 andalso loop (G.compute a1, G.compute a2)
                | loop _
                  = false
          in loop (G.make_sequence t1, G.make_sequence t2)
          end
  end
```

In the remainder of this section, we review a variety of generators.

4.1 Depth first

4.1.1 An eager traversal

The simplest solution is to represent sequences as a data type isomorphic to that of lists. To this end, we define `make_sequence` as an accumulator-based flatten function:

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

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

  fun make_sequence t
      = visit (t, END)

  fun compute value
      = value
end
```

In this solution, the sequence of leaves is built eagerly and therefore completely before any comparison takes place. This choice is known to be inefficient because if two leaves differ, the remaining two sequences are not used and therefore did not need to be built.

4.1.2 A lazy traversal

A more efficient solution—and indeed a traditional motivation for lazy evaluation [36, 41]—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
```

Unlike in the eager solution, the construction of the sequence in `Lazy_generator` and the comparisons in `same_fringe` are interleaved. This choice is known to be more efficient because if two leaves differ, the remaining two sequences are not built at all.

4.1.3 A continuation-based traversal

Alternatively to viewing the thunk of type `unit -> sequence`, in the lazy traversal of Section 4.1.2, 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.

4.1.4 A direct-style traversal with `shift` and `reset`

In direct style, the delimited continuation `a` of Section 4.1.3 is initialized with the control delimiter `reset`, extended by functional sequencing, captured by the delimited-control operator `shift`, and resumed by function application.

Using Filinski's functor `Shift_and_Reset` defined in Appendix A, 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 4.1.2.

The key points of this CPS transformation are as follows:

- the clause

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

  is transformed into:

  ```
  visit (NODE (t1, t2), a)
  = visit (t1, fn () => visit (t2, a))
  ```

- the clause

  ```
  visit (LEAF i)
  = shift (fn a => NEXT (i, a))
  ```

  is transformed into:

  ```
  visit (LEAF i, a)
  = NEXT (i, a)
  ```

- and the expression

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

  is transformed into:

  ```
  visit (t, fn () => END)
  ```

4.1.5 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 [25, 65]. 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 [52]:

```
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. (Such a stack is referred to as a 'data-structure continuation' in the literature [80, page 179].)

4.2 Breadth first

4.2.1 A queue-based traversal

Replacing the (last-in, first-out) stack, in the definition of Section 4.1.5, 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 4.1.5, 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* [25]. 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, traversing an inductive data structure breadth-first does not mesh well with compositional recursive descent: how would one write a breadth-first traversal with a fold function?

4.2.2 A direct-style traversal with `control` and `prompt`

The critical operation in the definition of `visit`, in Section 4.2.1, 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.

4.3 Summary and conclusion

We first have presented a spectrum of solutions to the traditional depth-first samefringe problem. Except for the defunctionalized ones, all the solutions are compositional in the sense of denotational semantics (i.e., visiting each subtree is defined as the composition of visiting its own subtrees). 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 have then obtained a solution to the breadth-first counterpart of the samefringe problem. Viewing this queue as a 'data-structure continuation,' we have observed that the operations upon it correspond to the operations induced by the composition of a dynamic delimited continuation and the current (delimited) continuation. We have then written this program compositionally and in direct style using `control` and `prompt`.

In the induction clause of `visit` in Section 4.2.2, 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 4.1.4. The resulting pattern of use of `control` and `prompt` in the modified definition is the traditional one used to simulate `shift` and `reset` [11].

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`, such as those discussed in Section 2.4.

5 Numbering a tree

We now turn to Okasaki's problem of numbering a tree in breadth-first order with successive numbers [60]. We express it in direct style with `control` and `prompt`, and we then outline its depth-first counterpart. Okasaki considers fully-labeled binary trees:

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

5.1 Breadth-first numbering

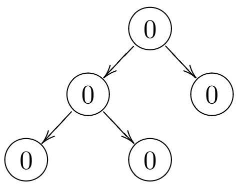
Given a tree T containing $|T|$ labels, we want to create a new tree of the same shape, but with the values in the nodes and leaves replaced by the numbers $1 \ldots |T|$ in breadth-first order. For example, the tree

```
NODE (NODE (LEAF 0, 0, LEAF 0), 0, LEAF 0)
```

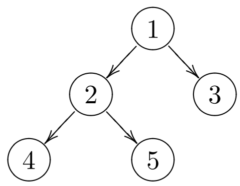
contains 5 labels and should be transformed into

```
NODE (NODE (LEAF 4, 2, LEAF 5), 1, LEAF 3)
```

i.e., the tree



should be transformed into



5.1.1 A queue-based traversal

In his solution [60], Okasaki relabels a tree by mapping it recursively into a first-in, first-out list of subtrees at call time and constructing the result at return time by reading this queue. To this end, he needs an auxiliary function

```
last_two_and_before : int list -> int list * int * int
```

such that applying it to the list `[xn, ..., x3, x2, x1]` yields the triple `([xn, ..., x3], x2, x1)`.

Okasaki's solution reads as follows:

```
(*  breadth_first_label : tree -> tree  *)
fun breadth_first_label t
    = let (*  visit : tree * int * tree list -> tree list  *)
          fun visit (LEAF _, i, k)
              = (LEAF i) :: (continue (k, i+1))
            | visit (NODE (t1, _, t2), i, k)
              = let val (rest, t1', t2')
                         = last_two_and_before
                             (continue (k @ [t1, t2], i+1))
                in (NODE (t1', i, t2')) :: rest
                end
          (*  continue : tree list * int -> tree list  *)
          and continue (nil, _)
              = nil
            | continue (t :: k, i)
              = visit (t, i, k)
      in last (visit (t, 1, nil))
      end
```

where `last` is a function mapping a non-empty list to its last element.

The above algorithm uses two queues of trees:

- the input queue, with function `visit` processing its front element, and with function `continue` processing its tail, and
- the output backwards queue, which is enqueued in both clauses of function `visit`, and which is dequeued by functions `last_two_and_before` and `last`.

5.1.2 A direct-style traversal with `control` and `prompt`

As in Section 4.2.2, we observe that the concatenation, in the definition of `visit` just above, matches the concatenation of stack frames in the specification of `control` in Section 2.2. One can therefore write the above function in direct style, using `control` and `prompt`. However, the solution requires a change of representation of the intermediate answer type of delimited continuations, i.e., the output queue, from `tree list` to `tree list * int` in order to unify the type `int` of the threaded index and the type `tree list` of the computation.

The direct-style breadth-first numbering program reads as follows:

```
local structure CP = Control_and_Prompt
                      (type intermediate_answer = tree list * int)
in val control = CP.control
   val prompt = CP.prompt
end

(*  breadth_first_label' : tree -> tree  *)
fun breadth_first_label' t
    = let (*  visit : tree * int -> int  *)
          fun visit (LEAF _, i)
              = control
                  (fn k =>
                    let val (ts, i') = prompt (fn () => k (i+1))
                    in ((LEAF i) :: ts, i')
                    end)
            | visit (NODE (t1, _, t2), i)
              = control
                  (fn k =>
                    let val (ts, i')
                            = prompt
                                (fn () => let val (nil, i1) = k (i+1)
                                              val i2 = visit (t1, i1)
                                              val i3 = visit (t2, i2)
                                          in (nil, i3)
                                          end)
                        val (rest, t1', t2') = last_two_and_before ts
                    in ((NODE (t1', i, t2')) :: rest, i')
                    end)
      in last (#1 (prompt (fn () => let val i = visit (t, 1)
                                    in (nil, i)
                                    end)))
      end
```

Again, the queuing effect is obtained in the induction case, where the current delimited continuation (of `visit`) is captured, bound to `k`, and applied to the increased index `i+1`. The implicit continuation of this application visits `t1` and then `t2`. Applying `k` seals it to the implicit continuation so that any continuation captured by an ulterior recursive call to `visit` in `k` captures both the rest of `k` and the visit of `t1` and `t2`.

In the program above, before the last leaf in the tree is visited, the intermediate results represent the current value of the index. After the last leaf in the tree is visited, the intermediate results represent the current output queue. Therefore, we need to fix the intermediate answer type to `tree list * int` so that the intermediate results are represented as pairs, where, depending on the stage of the computation, one of the components contains significant information. Before the last leaf in the tree is visited, the significant information (i.e., the index) is contained only in the second component, and the first component is irrelevant and always equal to `nil`. After the last leaf in the tree is visited, the significant information (i.e., the output queue) is contained only in the first component, and the second component is irrelevant and always equal to $|T|+1$ (where T is the input tree and $|T|$ is the number of its labels).

5.2 Depth-first numbering

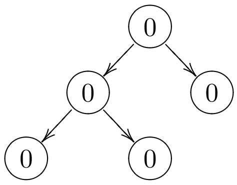
We now turn to the depth-first counterpart of Okasaki's pearl, and present a spectrum of solutions to the problem of depth-first tree numbering. Given a tree T containing $|T|$ labels, we want to create a new tree of the same shape, but with the values in the nodes and leaves replaced by the numbers $1 \ldots |T|$ in depth-first order. For example, the tree

```
NODE (NODE (LEAF 0, 0, LEAF 0), 0, LEAF 0)
```

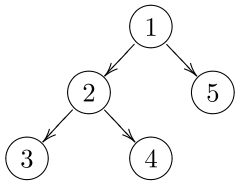
should be transformed into

```
NODE (NODE (LEAF 3, 2, LEAF 4), 1, LEAF 5)
```

i.e., the tree



should be transformed into



5.2.1 A stack-based traversal

It is trivial to write the depth-first counterpart of Okasaki's solution: one should just replace the queue with a stack, and instead of using `last_two_and_before`, use the auxiliary function

```
first_two_and_after : int list -> int * int * int list
```

such that applying it to the list `[x1, x2, x3, ..., xn]` yields the triple `(x1, x2, [x3, ..., xn])`.

The depth-first solution reads as follows:

```
(*  depth_first_label : tree -> tree  *)
fun depth_first_label t
    = let (*  visit :  tree * int * tree list -> tree list  *)
          fun visit (LEAF _, i, ts)
              = (LEAF i) :: (continue (ts, i+1))
            | visit (NODE (t1, _, t2), i, ts)
              = let val (t1', t2', rest)
                          = first_two_and_after
                              (continue (t1 :: t2 :: ts, i+1))
                in (NODE (t1', i, t2')) :: rest
                end
          (*  continue : tree list * int -> tree list  *)
          and continue (nil, _)
              = nil
            | continue (t :: k, i)
              = visit (t, i, k)
      in hd (visit (t, 1, nil))
      end
```

In contrast to Section 5.1.1, where the clause for nodes was concatenating the two subtrees in the back of the list of subtrees, in a first-in, first-out fashion,

```
last_two_and_before
 (continue (k @ [t1, t2], i+1))    (* then *)
```

the clause for nodes is (essentially) concatenating the two subtrees in front of the list of subtrees, in a last-in, first-out fashion:

```
first_two_and_after
 (continue ([t1, t2] @ ts, i+1))   (* now *)
```

We can see that the algorithm uses two stacks of trees:

- the input stack, with function `visit` processing its top element, and with function `continue` processing its tail, and
- the output stack, which is pushed on in both clauses of function `visit`, and which is popped off by functions `first_two_and_after` and `hd`.

5.2.2 A continuation-based traversal

In the induction case of `visit`, let us unfold the call to `continue` to obtain the following clause:

```
| visit (NODE (t1, _, t2), i, ts)
  = let val (t1', t2', rest)
            = first_two_and_after
               (visit (t1, i+1, t2 :: ts))
    in (NODE (t1', i, t2')) :: rest
    end
```

The modified definition is in defunctionalized form: the data type is that of lists and `continue` is the corresponding apply function. The higher-order counterpart of this defunctionalized definition reads as follows:

```
(*  depth_first_label' : tree -> tree  *)
fun depth_first_label' t
    = let (*  visit : tree * int * (int -> tree list) -> tree list  *)
          fun visit (LEAF _, i, k)
              = (LEAF i) :: (k (i+1))
            | visit (NODE (t1, _, t2), i, k)
              = let val (t1', t2', rest)
                        = first_two_and_after
                           (visit (t1, i+1, fn i' => visit (t2, i', k)))
                in (NODE (t1', i, t2')) :: rest
                end
      in hd (visit (t, 1, fn i => nil))
      end
```

5.2.3 A direct-style traversal with `shift` and `reset`

We view the function of type `int -> tree list`, in the definition just above, as a delimited continuation. This delimited continuation is initialized in the initial call to `visit`, extended in the induction case, and captured and resumed in both clauses of `visit`. In direct style, the initialization is obtained with `reset`, the extension is obtained by functional sequencing, the capture is obtained with `shift`, and the activation is obtained by function application. The result is another new example of programming with static delimited-control operators:

```
local structure SR = Shift_and_Reset
                       (type intermediate_answer = tree list)
in val shift = SR.shift
   val reset = SR.reset
end

(*  depth_first_label'' : tree -> tree  *)
fun depth_first_label'' t
    = let (*  visit :  tree * int -> tree list  *)
          fun visit (LEAF _, i)
              = shift
                 (fn k =>
                   (LEAF i) :: (k (i+1)))
            | visit (NODE (t1, _, t2), i)
              = shift
                 (fn k =>
                   let val (t1', t2', rest)
                           = first_two_and_after
                              (reset
                                (fn () => k (let val i' = visit (t1, i+1)
                                             in visit (t2, i')
                                             end)))
                   in (NODE (t1', i, t2')) :: rest
                   end)
      in hd (reset (fn () => let val i = visit (t, 1)
                             in nil
                             end))
      end
```

CPS-transforming `visit` yields the definition of Section 5.2.2.

5.3 Summary and conclusion

Okasaki's solution relabels its input tree in breadth-first order and uses a queue. We have expressed it in direct style using `control` and `prompt`. In so doing, we have internalized the explicit data operations on the queue into implicit control operations. These control operations crucially involve delimited continuations whose extent is dynamic.

The stack-based counterpart of Okasaki's solution relabels its input tree in depth-first order. We have mechanically refunctionalized this program into another one, which is continuation-based, and we have expressed this continuation-based program in direct style using `shift` and `reset`. These control operators crucially involve delimited continuations whose extent is static.

6 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 [12, 47, 68], 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, such as the one in Section 2.5.

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 two examples of breadth-first traversal: the breadth-first counterpart of the traditional samefringe function and Okasaki's breadth-first numbering pearl. We have recently proposed yet another example that exhibits the difference between `shift` and `control` [7, Section 4.6] [11, page 5].

One lesson we have learned here is how helpless one can feel when going beyond CPS. Unlike with `shift` and `reset`, there is no infrastructure for transforming programs that use `control` and `prompt`. We have therefore relied on CPS and on defunctionalization as guidelines, and we have built on the vision of data-structure continuations (stacks for depth-first traversals and queues for breadth-first traversals) proposed by Friedman 25 years ago [80, page 179] to infer the breadth-first traversals. We would have been hard pressed to come up with these examples only by groping for delimited continuations in direct style.[1]

Since `control`, even more dynamic delimited-control operators (some of which generate control delimiters dynamically) have been proposed [27, 39, 42, 56, 58, 64], 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 [2], 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 [3, 4].
- In his PhD thesis [58], 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`. As such, his is 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.

[1] "You are not Superman." – Aunt May (2002)

Acknowledgments: We are grateful to Mads Sig Ager, Małgorzata Biernacka, Andrzej Filinski, Julia Lawall, Kevin Millikin, and the anonymous referees of Information Processing Letters and Science of Computer Programming 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. Thanks are also due to Bernd Grobauer and Julia Lawall for sharing brain cycles about breadth-first numbering in Summer 2000.

The third author contributed to this work while at Harvard University.

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.

A An implementation of `shift` and `reset`

In his seminal article [34], Filinski has presented an ML implementation of `shift` and `reset` in terms of `callcc` and mutable state, along with its correctness proof. This implementation takes the form of a functor `Shift_and_Reset`, which maps a type of intermediate answers into a structure providing instances of `shift` and `reset` at that type:

```
signature ESCAPE
= sig
    type void
    val coerce : void -> 'a
    val escape : (('a -> void) -> 'a) -> 'a
  end

structure Escape : ESCAPE
= struct
    datatype void = VOID of void
    fun coerce (VOID v) = coerce v
    local open SMLofNJ.Cont
    in fun escape f
           = callcc (fn k => f (fn x => throw k x))
    end
  end

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

```
functor Shift_and_Reset (type intermediate_answer) : SHIFT_AND_RESET
= struct
    open Escape

    exception MISSING_RESET

    val mk : (intermediate_answer -> void) ref
           = ref (fn _ => raise MISSING_RESET)

    fun abort x
        = coerce (!mk x)

    type intermediate_answer = intermediate_answer

    fun reset thunk
        = escape (fn k => let val m = !mk
                          in mk := (fn r => (mk := m; k r));
                             abort (thunk ())
                          end)

    fun shift function
        = escape
           (fn k => abort (function (fn v => reset
                                              (fn () => coerce (k v)))))
  end
```

B An implementation of control and prompt

The functor `Control_and_Prompt` maps a type of intermediate answers into a structure providing instances of `control` and `prompt` at that type:

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

functor Control_and_Prompt (type intermediate_answer)
: CONTROL_AND_PROMPT
= struct
    datatype ('t, 'w) context'
             = CONTEXT of 't -> ('w, 'w) context' option -> 'w

    fun send v NONE
        = v
      | send v (SOME (CONTEXT mc))
        = mc v NONE

    fun compose' (CONTEXT c, NONE)
        = CONTEXT c
      | compose' (CONTEXT c, SOME mc1)
        = CONTEXT (fn v => fn mc2 => c v (SOME (compose' (mc1, mc2))))
```

```
  fun compose (CONTEXT c, NONE)
      = CONTEXT c
    | compose (CONTEXT c, SOME mc1)
      = CONTEXT (fn v => fn mc2 => c v (SOME (compose' (mc1, mc2))))

  structure SR
  = Shift_and_Reset
     (type intermediate_answer
           = (intermediate_answer, intermediate_answer) context' option
             -> intermediate_answer)
  val shift = SR.shift
  val reset = SR.reset

  type intermediate_answer = intermediate_answer

  fun prompt thunk
      = reset (fn () => send (thunk ())) NONE

  exception MISSING_PROMPT

  fun control function
      = shift
         (fn c1 =>
           fn mc1 =>
            let val k
                    = fn x =>
                       shift
                        (fn c2 =>
                          fn mc2 =>
                           let val (CONTEXT c1') = compose (CONTEXT c1, mc1)
                           in c1' x (SOME (compose (CONTEXT c2, mc2)))
                           end)
            in reset (fn () => send (function k)) NONE
            end) handle MISSING_RESET => raise MISSING_PROMPT
end
```

A delimited continuation captured by `control` may capture the context in which it is subsequently activated. To simulate this dynamic extent, the captured continuation (of type `('t, 'w) context'`) takes as arguments not just the value (of type `'t`) with which it is activated, but also the context (of type `('w, 'w) context' option`) in which it is activated. Hence the recursive definition of `datatype ('t, 'w) context'`.

Such a captured continuation can no longer be activated by mere function application; instead we define `send v c` to activate the captured continuation `c` with the value `v`. Such a captured continuation can also no longer be composed by mere function composition; instead we define `compose c mc` to concatenate the captured continuation `c` with the outer continuation (activation context) `mc`.

A direct transliteration of Shan's Scheme macros into ML results in an implementation with overly restrictive types. Due to the lack of polymorphic recursion in ML, the function `compose` would have the type:

```
('w, 'w) context' * ('w, 'w) context' option -> ('w, 'w) context'
```

and consequently, the inferred type of `control` would be:

```
((intermediate_answer -> intermediate_answer) -> intermediate_answer)
-> intermediate_answer
```

The third author has therefore cloned the function `compose` so that it has the following type:

```
('t, 'w) context' * ('w, 'w) context' option -> ('t, 'w) context'
```

Consequently, the inferred type of `control` is the same as that of `shift` in Filinski's implementation:

```
(('a -> intermediate_answer) -> intermediate_answer) -> 'a
```

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, 1985.

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

[3] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Xavier Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, pages 64–76, Venice, Italy, January 2004. ACM Press.

[4] Vincent Balat and Olivier Danvy. Memoization in type-directed partial evaluation. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the 2002 ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, number 2487 in Lecture Notes in Computer Science, pages 78–92, Pittsburgh, Pennsylvania, October 2002. Springer-Verlag.

[5] Josh Berdine. *Linear and Affine Typing of Continuation-Passing Style.* PhD thesis, Queen Mary, University of London, 2004.

[6] Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages.* PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

[7] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004).

[8] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. Research Report BRICS RS-05-22, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, July 2005.

[9] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, December 2005.

[10] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[11] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited control. Research Report BRICS RS-05-25, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, August 2005. Theoretical Pearl to appear in the Journal of Functional Programming.

[12] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. Research Report BRICS RS-05-16, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 2005.

[13] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the dynamic extent of delimited continuations. *Information Processing Letters*, 96(1):7–17, 2005.

[14] Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2(4):347–359, 1984.

[15] Robert (Corky) Cartwright, editor. *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 1991. ACM Press.

[16] Eugene Charniak, Christopher Riesbeck, and Drew McDermott. *Artificial Intelligence Programming*. Lawrence Earlbaum Associates, 1980.

[17] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[18] Olivier Danvy. On some functional aspects of control. In Thomas Johnsson, Simon Peyton Jones, and Kent Karlsson, editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pages 445–449. Program Methodology Group, University of Göteborg and Chalmers University of Technology, September 1988. Report 53.

[19] Olivier Danvy. Programming with tighter control. *Special issue of the Bigre journal: Putting Scheme to Work*, 65:10–29, July 1989.

[20] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Hayo 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, January 2004. Invited talk.

[21] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[22] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [81], pages 151–160.

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

[24] Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1):53–73, 2002. Extended version available as the technical report BRICS RS-01-29. A preliminary version was presented at the Second International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2001).

[25] 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. Extended version available as the technical report BRICS RS-01-23.

[26] Bruce F. Duba, Robert Harper, and David B. MacQueen. Typing first-class continuations in ML. In Cartwright [15], pages 163–173.

[27] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for subcontinuations. Technical Report 615, Computer Science Department, Indiana University, Bloomington, Indiana, June 2005.

[28] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987.

[29] Matthias Felleisen. The theory and practice of first-class prompts. In Ferrante and Mager [33], pages 180–190.

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

[31] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana, February 1987.

[32] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988. ACM Press.

[33] Jeanne Ferrante and Peter Mager, editors. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988. ACM Press.

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

[35] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996. Technical Report CMU-CS-96-119.

[36] 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.

[37] Carsten Führmann. *The Structure of Call-by-Value*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 2000.

[38] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[39] Carl Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In Simon 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.

[40] John Hatcliff. *The Structure of Continuation-Passing Styles*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, June 1994.

[41] 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.

[42] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[43] Ralf Hinze. Deriving backtracking monad transformers. In Wadler [79], pages 186–197.

[44] Gregory F. Johnson. GL – a denotational testbed with continuations and partial continuations as first-class objects. In Mark Scott Johnson, editor, *Proceedings*

of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices, Vol. 22, No 7, pages 154–176, Saint-Paul, Minnesota, June 1987. ACM Press.

[45] Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In Ferrante and Mager [33], pages 158–168.

[46] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[47] Oleg Kiselyov. How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Computer Science Department, Indiana University, Bloomington, Indiana, March 2005.

[48] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers. In Benjamin Pierce, editor, *Proceedings of the 2005 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 40, No. 9, pages 192–203, Tallinn, Estonia, September 2005. ACM Press.

[49] Jim Laird. *A semantic analysis of control*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1998.

[50] Peter Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998. Reprinted from a technical report, UNIVAC Systems Programming Research (1965), with a foreword [77].

[51] Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, July 1994.

[52] John McCarthy. Another samefringe. *SIGART Newsletter*, 61, February 1977.

[53] Chris Mellish and Steve Hardy. Integrating Prolog in the POPLOG environment. In John A. Campbell, editor, *Implementations of PROLOG*, pages 147–162. Ellis Horwood, 1984.

[54] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, New York, June 1985. Springer-Verlag.

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

[56] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 182–197, Madrid, Spain, September 1994. Springer-Verlag.

[57] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1990.

[58] Aleksandar 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.

[59] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-7.

[60] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In Wadler [79], pages 131–136.

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

[62] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 2001. Technical Report CMU-CS-01-152.

[63] Christian Queinnec. Value transforming style. In *Proceedings of the Second International Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 20–28, Bordeaux, France, September 1992. IRISA, Rennes, France.

[64] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In Cartwright [15], pages 174–184.

[65] 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.

[66] Amr Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. PhD thesis, Computer Science Department, Rice University, Houston, Texas, August 1994. Technical report 94-242.

[67] Erik Sandewall. An early use of continuations and partial evaluation for compiling rules written in FOPC. *Higher-Order and Symbolic Computation*, 12(1):105–113, 1999.

[68] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Technical report TR600, Computer Science Department, Indiana University, Snowbird, Utah, September 2004.

[69] Chung-chieh Shan. *Linguistic Side Effects*. PhD thesis, Division of Engineering and Applied Science, Harvard University, September 2005.

[70] Dorai Sitaram. *Models of Control and their Implications for Programming Language Design*. PhD thesis, Computer Science Department, Rice University, Houston, Texas, April 1994.

[71] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [81], pages 161–175.

[72] Michael Spivey and Silvija Seres. Combinators for logic programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 177–199. Palgrave Macmillan, 2003.

[73] 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.

[74] Christopher Strachey and Christopher 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.

[75] Gerald J. Sussman and Guy 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.

[76] Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1997. ECS-LFCS-97-376.

[77] Hayo Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.

[78] Philip Wadler. How to replace failure by a list of successes. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128, Nancy, France, September 1985. Springer-Verlag.

[79] Philip Wadler, editor. *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 35, No. 9, Montréal, Canada, September 2000. ACM Press.

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

[81] Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[82] Mitchell Wand and Dale Vaillancourt. Relating models of backtracking. In Kathleen Fisher, editor, *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 39, No. 9, pages 54–65, Snowbird, Utah, September 2004. ACM Press.