On the Static and Dynamic Extents
doing Delimited Continuations*

Dariusz Biernacki & Olivier Danvy  Chung-chieh Shan
BRICS†
Department of Computer Science  Department of Computer Science
University of Aarhus‡  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 \texttt{shift} (alias \texttt{S}) and the dynamic delimited-control
operator \texttt{control} (alias \texttt{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.

---

A substantially shorter version appeared in Information Processing Letters [7].
†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 .................................................................. 1  
  1.1 Background ................................................................. 1  
  1.2 Overview ................................................................. 1  

2 An operational characterization ........................................ 2  
  2.1 An abstract machine for shift and reset ...................... 2  
  2.2 An abstract machine for control and prompt ............... 4  
  2.3 Simulating shift in terms of control and prompt .......... 5  
  2.4 Simulating control in terms of shift and reset ............. 6  
  2.5 Three examples in ML ................................................. 7  

3 Programming with delimited continuations ....................... 8  

4 The samefringe problem ................................................. 10  
  4.1 Depth first ............................................................ 11  
    4.1.1 An eager traversal ............................................. 11  
    4.1.2 A lazy traversal ............................................... 12  
    4.1.3 A continuation-based traversal ......................... 12  
    4.1.4 A direct-style traversal with shift and reset ........ 12  
    4.1.5 A stack-based traversal ..................................... 14  
  4.2 Breadth first ........................................................... 15  
    4.2.1 A queue-based traversal .................................... 15  
    4.2.2 A direct-style traversal with control and prompt .... 16  
  4.3 Summary and conclusion ........................................... 17  

5 Labeling a tree ................................................................ 18  
  5.1 Breadth-first numbering ............................................ 18  
    5.1.1 A queue-based traversal ................................... 19  
    5.1.2 A direct-style traversal with control and prompt ... 20  
  5.2 Depth-first numbering .............................................. 21  
    5.2.1 A stack-based traversal ..................................... 21  
    5.2.2 A continuation-based traversal ......................... 23  
    5.2.3 A direct-style traversal with shift and reset ........ 23  
  5.3 Summary and conclusion ........................................... 24  

6 Conclusion and issues .................................................... 24  

A An implementation of shift and reset ............................. 26  

B An implementation of control and prompt ....................... 27
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 $F$) [28, 31, 32, 71] and Danvy and Filinski’s shift (alias $S$) [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 {}) 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

\[
\begin{align*}
&x_1 :: x_2 :: x_3 \\
&(x_1 :: x_2 :: \text{nill}) @ x_3 \\
&[x_1, x_2] @ x_3
\end{align*}
\]

where :: denotes infix list construction and @ denotes infix list concatenation. In an environment where \(x_1\) denotes 1, \(x_2\) denotes 2, and \(x_3\) 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 \(\lambda\)-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 \(\{\}\) and shift is noted \(S\). 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 \(\{\}\) and \(S\), 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-state;
- an abstraction \(\lambda x.t\) is evaluated into a closure \([x, t, e]\) and the machine switches to a cont-state;
• Terms: \( t ::= x \mid \lambda x.t \mid t_0 t_1 \mid \langle \rangle \mid S k.t \)

• Values (closures and captured continuations): \( v ::= [x, t, e] \mid C \)

• Environments: \( e ::= e_{empty} \mid e[x \mapsto v] \)

• Evaluation contexts: \( C_1 ::= \text{END} \mid \text{ARG}((t,e), C_1) \mid \text{FUN}(v, C_1) \)

• Meta-contexts: \( C_2 ::= \bullet \mid C_1 \cdot C_2 \)

• Initial transition, transition rules, and final transition:

<table>
<thead>
<tr>
<th>( t )</th>
<th>( {t, e_{empty}, \text{END}, \bullet}_{\text{eval}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {x, e, C_1, C_2}_{\text{eval}} )</td>
<td>( {C_1, e(x), C_2}_{\text{cont}} )</td>
</tr>
<tr>
<td>( \lambda x.t, e, C_1, C_2)_{eval}</td>
<td>( {C_1, [x, t, e], C_2}_{\text{cont}} )</td>
</tr>
<tr>
<td>( t_0 t_1, e, C_1, C_2)_{eval}</td>
<td>( {t_0, e, \text{ARG}((t_1, e), C_1), C_2}_{\text{eval}} )</td>
</tr>
<tr>
<td>( \langle \rangle, e, C_1, C_2)_{eval}</td>
<td>( {t, e, \text{END}, C_1 \cdot C_2}_{\text{eval}} )</td>
</tr>
<tr>
<td>( \langle \rangle, e, C_1, C_2)_{eval}</td>
<td>( {t, e, \text{FUNCTION}(v, C_1, C_2)}_{\text{eval}} )</td>
</tr>
<tr>
<td>( \langle \text{END}, v, C_2\rangle_{\text{cont}} )</td>
<td>( {C_2, v}_{\text{cont}} )</td>
</tr>
<tr>
<td>( \langle \text{ARG}((t,e), C_1), v, C_2\rangle_{\text{cont}} )</td>
<td>( {t, e, \text{FUNCTION}(v, C_1, C_2)}_{\text{eval}} )</td>
</tr>
<tr>
<td>( \langle \text{FUNCTION}([x,t,e], C_1), v, C_2\rangle_{\text{cont}} )</td>
<td>( {t, e[x \mapsto v], C_1, C_2}_{\text{eval}} )</td>
</tr>
<tr>
<td>( \langle \text{FUNCTION}(C_1', C_1), v, C_2\rangle_{\text{cont}} )</td>
<td>( {C_1', v, C_1 \cdot C_2}_{\text{cont}} )</td>
</tr>
<tr>
<td>( \langle C_1 \cdot C_2, v\rangle_{\text{cont}} )</td>
<td>( {C_1, v, C_2}_{\text{cont}} )</td>
</tr>
<tr>
<td>( \langle \bullet, v\rangle_{\text{cont}} )</td>
<td>( v )</td>
</tr>
</tbody>
</table>

Figure 1: A call-by-value environment-based abstract machine for the \( \lambda \)-calculus extended with shift \( (S) \) and reset \( (\langle \rangle) \)

• an application \( t_0 t_1 \) is processed by pushing \( t_1 \) and the environment onto the context and switching to a new \text{eval}-state to process \( t_0 \);

• a reset-expression \( \langle \rangle \) is processed by pushing the current context on the current meta-context and switching to a new \text{eval}-state to process \( t \) in an empty context, as an intermediate computation;

• a shift-expression \( S k.t \) is processed by capturing the context \( C_1 \) and binding it to \( k \), and switching to a new \text{eval}-state to process \( t \) in an empty context.

The transitions of a \text{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 \textit{cont}_2-state;

• a context ARG \((t, e, C_1)\) specifies the evaluation of an argument; it is processed by switching to an \textit{eval}-state to process \(t\) in a new context;

• a context FUN \([x, t, e, C_1]\) specifies the application of a closure; it is processed by switching to an \textit{eval}-state to process the term \(t\) with an extension of the environment \(e\);

• a context 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 \textit{cont}_1-state to process \(C'_1\).

The transitions of a \textit{cont}_2-state are defined by cases on its first component:

• an empty meta-context • 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 \textit{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 \texttt{shift} and \texttt{reset}, whose definition is based on CPS, \texttt{control} and \texttt{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 \(\ast\) is defined as follows:

\[
\begin{align*}
\text{END} \ast C'_1 &= C'_1 \\
(\text{ARG} \((t, e, C_1)\)) \ast C'_1 &= \text{ARG} \((t, e, C_1 \ast C'_1)\) \\
(\text{FUN} \((v, C_1)\)) \ast C'_1 &= \text{FUN} \((v, C_1 \ast C'_1)\)
\end{align*}
\]

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.,

\[
\{\text{FUN} \((C'_1, C_1), v, C_2\)\}_{\text{cont}_2} \Rightarrow \{C'_1, v, C_1 \cdot C_2\}_{\text{cont}_1}
\]

with a transition that applies a captured context \(C'_1\) by concatenating it with the current context \(C_1\):

\[
\{\text{FUN} \((C'_1, C_1), v, C_2\)\}_{\text{cont}_1} \Rightarrow \{C'_1 \ast C_1, v, C_2\}_{\text{cont}_1}
\]

This change gives \(S\) (alias \texttt{shift}) the behavior of \(F\) (alias \texttt{control}). In contrast, \(\langle \rangle\) (alias \texttt{reset}) and \# (alias \texttt{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 $\text{prompt} \ (\text{fn} \ O \Rightarrow k \ v)$ when $v$ denotes a value, and every other occurrence of $k$ should be replaced with $\text{fn} \ x \Rightarrow \text{prompt} \ (\text{fn} \ O \Rightarrow k \ x)$.)

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 $C'_1 \star \text{END} = C'_1$. 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:

```ml
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:

```ml
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

\[
\text{reset}(\lambda () \Rightarrow \text{shift}(\lambda k \Rightarrow 10 + (k 100)) + \text{shift}(\lambda k' \Rightarrow 1))
\]
evaluates to 11, whereas (replacing \text{reset} by \text{prompt} and \text{shift} by \text{control})

\[
\text{prompt}(\lambda () \Rightarrow \text{control}(\lambda k \Rightarrow 10 + (k 100)) + \text{control}(\lambda k' \Rightarrow 1))
\]
evaluates to 1 and (delimiting the application of \(k\) with \text{prompt})

\[
\text{prompt}(\lambda () \Rightarrow \text{control}(\lambda k \Rightarrow 10 + \text{prompt}(\lambda () \Rightarrow k 100)) + \text{control}(\lambda k' \Rightarrow 1))
\]
evaluates to 11.

In the first case, \(\text{shift}(\lambda k \Rightarrow 10 + (k 100))\) is evaluated with a continuation that could be written functionally as \(\lambda v \Rightarrow v + \text{shift}(\lambda k' \Rightarrow 1)\). When \(k\) is applied, the expression \(\text{shift}(\lambda k' \Rightarrow 1)\) is evaluated in a context that could be represented functionally as \(\lambda v \Rightarrow 100 + v\) and in a meta-context that could be represented as \((\lambda v \Rightarrow 10 + v) :: \text{nil}\); 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., \(\lambda v \Rightarrow 10 + v\) 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, \(\text{control}(\lambda k \Rightarrow 10 + (k 100))\) is evaluated with a continuation that could be written functionally as \(\lambda v \Rightarrow v + \text{control}(\lambda k' \Rightarrow 1)\). When \(k\) is applied, the expression \(\text{control}(\lambda k' \Rightarrow 1)\) is evaluated in a context that results from composing \(\lambda v \Rightarrow 10 + v\) and \(\lambda v \Rightarrow 100 + v\) (and therefore could be represented functionally as \(\lambda v \Rightarrow 10 + (100 + v)\), 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, \(\text{control}(\lambda k \Rightarrow 10 + \text{prompt}(\lambda () \Rightarrow k 100))\) is evaluated with a continuation that could be written functionally as \(\lambda v \Rightarrow v + \text{control}(\lambda k' \Rightarrow 1)\). When \(k\) is applied, the expression \(\text{control}(\lambda k' \Rightarrow 1)\) is evaluated in a context that results from composing \(\lambda v \Rightarrow v\) and \(\lambda v \Rightarrow 100 + v\) (and therefore could be represented functionally as \(\lambda v \Rightarrow 100 + v\), and in a meta-context which could be represented as \((\lambda v \Rightarrow 10 + v) :: \text{nil}\); 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., \( \text{fn } v \rightarrow 10 + v \) 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:

\[
\begin{align*}
\text{let val } k &= \text{fn } v \rightarrow \text{let val } k' &= \text{fn } v' \rightarrow v + v' \\
&\text{in } 1 \\
&\text{end} \\
&\text{in } 10 + (k\ 100) \\
&\text{end}
\end{align*}
\]

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 \( \text{unit } \rightarrow 'a \) (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:

```
  3
 ↙  ↙  ↙  ↙  ↙  ↙
  1   2
      ↙  ↙
     1   2
```

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
The 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 \texttt{sequence} and a notion of computation (to compute the next element in a sequence):

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
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

in

end
\end{verbatim}

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 \texttt{make_sequence} as an accumulator-based flatten function:

\begin{verbatim}
structure Eager_generator : GENERATOR = struct
  datatype sequence = END
                         | NEXT of int * sequence computation
  withtype 'a computation = 'a
\end{verbatim}
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:

```plaintext
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, visit (t2, a))

  fun make_sequence t = visit (t, 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:

```ml
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

\[
\text{visit (NODE (t1, t2))} = \text{let val () = visit t1 in visit t2 end}
\]

is transformed into:

\[
\text{visit (NODE (t1, t2), a)} = \text{visit (t1, fn () \Rightarrow visit (t2, a))}
\]

- the clause

\[
\text{visit (LEAF i)} = \text{shift (fn a \Rightarrow NEXT (i, a))}
\]

is transformed into:

\[
\text{visit (LEAF i, a)} = \text{NEXT (i, a)}
\]

- and the expression

\[
\text{reset (fn () \Rightarrow let val () = visit t in END end)}
\]

is transformed into:

\[
\text{visit (t, fn () \Rightarrow END)}
\]

### 4.1.5 A stack-based traversal

Alternatively to writing the lazy solution in direct style, we can defunctionalize its computation (which has type \text{sequence computation}, i.e., \text{unit \rightarrow sequence}) and obtain a first-order solution [25, 65]. The inhabitants of the function space \text{unit \rightarrow sequence} are instances of the function abstractions in the initial call to \text{visit} (i.e., \text{fn () \Rightarrow END}) and in the induction case of \text{visit} (i.e., \text{fn () \Rightarrow visit (t2, a)}). We therefore represent this function space by (1) a sum corresponding to these two possibilities, and (2) the corresponding apply function, \text{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 \text{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]:

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

13
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`:

\[
\text{visit (NODE (t1, t2), a)} = \text{continue (t1 :: t2 :: a, ())}
\]

(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:

```ml
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)
    = 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
```

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

\[
| \text{visit } (\text{NODE } (t_1, t_2), a) = \text{continue } ([t_1, t_2] @ a, ()) (* \text{then} *)
\]

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

\[
| \text{visit } (\text{NODE } (t_1, t_2), a) = \text{continue } (a @ [t_1, t_2], ()) (* \text{now} *)
\]

Nothing else changes in the definition of the generator. In particular, subtrees are still removed from the front of the list of subtrees by \text{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 \text{continue} but also by @, this definition is not in the range of defunctionalization [25]. Therefore, even though \text{visit} is tail-recursive and constructs a data structure that is interpreted in \text{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 \text{control} and \text{prompt}

The critical operation in the definition of \text{visit}, in Section 4.2.1, is the enqueuing of the subtrees \(t_1\) and \(t_2\) to the current queue \(a\), which is achieved by the list concatenation \(a @ [t_1, t_2]\). We observe that this concatenation matches the concatenation of stack frames in the specification of \text{control} in Section 2.2.

Therefore—and this is a eureka step—one can write \text{visit} in direct style using \text{control} and \text{prompt}. To this end, we represent both queues \(a\) and \([t_1, t_2]\) as dynamic delimited continuations in such a way that their composition represents the concatenation of \(a\) and \([t_1, t_2]\). The direct-style traversal reads as follows:

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

15
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 same-fringe 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 \( t_1 \) and \( t_2 \) instead of before,

\[
\mid \text{visit (NODE (t1, t2))} \\
= \text{control (fn a => let val () = visit t1} \\
\text{val () = visit t2} \\
\text{in a ()) end)}
\]

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

\[
\mid \text{visit (NODE (t1, t2))} \\
= \text{let val () = visit t1} \\
\text{in visit t2 end}
\]

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:

\[
\text{datatype tree} = \text{LEAF of int} \\
| \text{NODE of tree * int * tree}
\]

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

\[
\text{NODE (NODE (LEAF 0, 0, LEAF 0), 0, LEAF 0)}
\]

contains 5 labels and should be transformed into

\[
\text{NODE (NODE (LEAF 4, 2, LEAF 5), 1, LEAF 3)}
\]

i.e., the tree should be transformed into

\[
\begin{array}{c}
\begin{array}{c}
0 \\
0 \\
0
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
1 \\
2 \\
3
\end{array}
\end{array}
\]

17
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

\[
\text{last_two_and_before} : \text{int list} \rightarrow \text{int list} \times \text{int} \times \text{int}
\]

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

Okasaki’s solution reads as follows:

\[
\begin{array}{l}
\text{(* breadth_first_label : tree -> tree *)}
\text{fun breadth_first_label t}
\text{  let (* visit : tree * int * tree list -> tree list *)}
\text{      fun visit (LEAF _, i, k)
\text{        = (LEAF i) :: (continue (k, i+1))}
\text{         | visit (NODE (t1, _, t2), i, k)
\text{           = let val (rest, t1', t2')
\text{             = last_two_and_before
\text{              (continue (k @ [t1, t2], i+1))}
\text{               in (NODE (t1', i, t2')) :: rest}
\text{    end)
\text{   end)}
\text{   (* continue : tree list * int -> tree list *)}
\text{and continue (nil, _)
\text{  = nil}
\text{     | continue (t :: k, i)
\text{        = visit (t, i, k)
\text{    end)
\text{   end)}
\text{where last is a function mapping a non-empty list to its last element.}
\text{The above algorithm uses two queues of trees:}
\text{• the input queue, with function visit processing its front element, and with function continue processing its tail, and}
\text{• the output backwards queue, which is enqueued in both clauses of function visit, and which is dequeued by functions last_two_and_before and last.}
\end{array}
\]

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)
            in ((LEAF i) :: ts, 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

\[
\text{NODE (NODE (LEAF 0, 0, LEAF 0), 0, LEAF 0)}
\]

should be transformed into

\[
\text{NODE (NODE (LEAF 3, 2, LEAF 4), 1, LEAF 5)}
\]

i.e., the tree

\[
\begin{array}{c}
\uparrow \\
0 \\
\downarrow \\
\uparrow \\
0 \\
\downarrow \\
0 \\
\end{array}
\]

should be transformed into

\[
\begin{array}{c}
\uparrow \\
1 \\
\downarrow \\
\uparrow \\
2 \\
\downarrow \\
3 \\
\end{array}
\begin{array}{c}
\uparrow \\
\downarrow \\
\uparrow \\
\downarrow \\
4 \\
\end{array}
\]

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

\[
\text{first_two_and_after : int list -> int * int * int list}
\]

such that applying it to the list $[x_1, x_2, x_3, \ldots, x_n]$ yields the triple $(x_1, x_2, [x_3, \ldots, x_n])$.

The depth-first solution reads as follows:

\[
(* \text{depth_first_label : tree -> tree *})
\]

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,

\[
\text{last_two_and_before}
\]

\[
(\text{continue} \ (k \ @ \ [t1, t2], i+1)) \quad (* \text{then} *)
\]

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

\[
\text{first_two_and_after}
\]

\[
(\text{continue} \ ([t1, t2] \ @ \ ts, i+1)) \quad (* \text{now} *)
\]

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

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

5.2.2 A continuation-based traversal

In the induction case of \text{visit}, let us unfold the call to \text{continue} to obtain the following clause:

\[
| \text{visit} \ (\text{NODE} \ (t1, \_, t2), i, ts) = \text{let val} \ (t1', t2', \text{rest}) = \text{first_two_and_after} \\
\hspace{1cm} (\text{visit} \ (t1, i+1, t2 :: ts)) \\
\hspace{1cm} \text{in} \ (\text{NODE} \ (t1', i, t2')) :: \text{rest} \end{lemma}
\]

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

\[
(* \text{depth_first_label'} : \text{tree} \rightarrow \text{tree} *)
\]

\[
\text{fun depth_first_label'} \ t = \text{let} (* \text{visit} : \text{tree} * \text{int} * (\text{int} \rightarrow \text{tree list}) \rightarrow \text{tree list} *)
\]

\[
\begin{align*}
| \text{fun} \ 	ext{visit} \ (\text{LEAF} \ _, i, k) = (\text{LEAF} \ i) :: (k \ (i+1)) \\
| \text{fun} \ 	ext{visit} \ (\text{NODE} \ (t1, \_, t2), i, k) = \text{let val} \ (t1', t2', \text{rest}) = \text{first_two_and_after} \\
\hspace{1cm} (\text{visit} \ (t1, i+1, \text{fn} \ i' = \Rightarrow \text{visit} \ (t2, i', k))) \\
\hspace{1cm} \text{in} \ (\text{NODE} \ (t1', i, t2')) :: \text{rest} \end{align*}
\]

\[
\text{end}
\]

\[
\text{in} \ \text{hd} \ (\text{visit} \ (t, 1, \text{fn} \ i = \Rightarrow \text{nil}))
\]

\[
\text{end}
\]
5.2.3 A direct-style traversal with shift and reset

We view the function of type \( \text{int} \rightarrow \text{tree list} \), 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:

```ml
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, i)
                        in nil
                        end))
                end
            in visit (t, 1)
    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 in a one-week brainstorming session 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)  
       end) handle MISSING_RESET => raise MISSING_PROMPT

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:

\[((\text{intermediate\_answer} \to \text{intermediate\_answer}) \to \text{intermediate\_answer}) \to \text{intermediate\_answer}\]

The third author has therefore cloned the function \text{compose} so that it has the following type:

\[((\tau, \nu) \text{context} \ast ((\nu, \nu) \text{context} \text{option}) \to ((\nu, \nu) \text{context})\]

Consequently, the inferred type of control is the same as that of \text{shift} in Filinski’s implementation:

\[((\tau \to \text{intermediate\_answer}) \to \text{intermediate\_answer}) \to \tau\]

References


[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.


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


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


