Shift to control

Chung-chieh Shan
Harvard University
ccshan@post.harvard.edu

Abstract

Delimited control operators abound, but their relationships are ill-understood, and it remains unclear which (if any) to consider canonical. Although all delimited control operators ever proposed can be implemented using undelimited continuations and mutable state, Gasbichler and Sperber [28] showed that an implementation that does not rely on undelimited continuations can be much more efficient. Unfortunately, they only implemented Felleisen's `control` and `prompt` [18, 19, 21, 22, 49] and (from there) Danvy and Filinski's `shift` and `reset` [11–13], not other proposed operators with which an expression may capture its context beyond an arbitrary number of dynamically enclosing delimiters.

We show that `shift` and `reset` can macro-express `control` and `prompt`, as well as the other operators, without capturing undelimited continuations or keeping mutable state. This translation is previously unknown in the literature. As a consequence, research on implementing `shift` and `reset`, such as Gasbichler and Sperber's, transfers to the other operators. Moreover, we treat all these operators by extending a standard CPS transform (defying some skepticism in the literature whether such a treatment exists), so they can be incorporated into CPS-based language implementations.

1 Introduction

The *continuation* is the rest of the computation, represented by the *context* of the current expression being evaluated. For example, in the program

```
(cons 'a (cons 'b (cons 'c '())))
```

the continuation of `(cons 'c '())` is to `cons` the symbol `b`, then the symbol `a`, onto the intermediate result. This continuation is represented by the context `(cons 'a (cons 'b _))`, where _ is a hole waiting to be plugged in.

Continuations can exist in a program at two levels. First, code may be written in *continuation-passing style* (CPS), in which continuations are managed explicitly as values at all times. Second, the underlying control flow of a program can be treated in terms of continuations. Scheme provides `call-with-current-continuation` (hereafter `call/cc`) to access these implicit continuations as first-class values [35]. Implicit continuations can be made explicit by a CPS transform on programs; explicit continuations can be made implicit by a corresponding *direct-style* transform [7, 14, 15, 46].

A *delimited* (or *composable*, or *partial*) continuation is a prefix of the rest of the computation, represented by a delimited part of the context of the current expression being evaluated. For example, in the program

```
(cons 'a [(cons 'b (cons 'c '()))])
```

the continuation of `(cons 'c '())`, as delimited by the square brackets, is to `cons` the symbol `b` onto the intermediate result. This delimited continuation is represented by the delimited context `[(cons 'b _)]`.

Delimited continuations, like undelimited ones, can be explicit (in CPS code) or implicit (in direct-style code). Since Felleisen's work [18, 19], many control operators have been proposed to access implicit delimited continuations as first-class values. A typical proposal provides, first, some way to delimit contexts, and second, some way to capture the current context up to an enclosing delimiter. For example, Danvy and Filinski [11–13] proposed two control operators `shift` and `reset`, with the following syntax.

$$\text{Expressions} \quad E ::= \cdots \mid \texttt{(shift f } E\texttt{)} \mid \texttt{(reset } E\texttt{)} \qquad (1)$$

Contexts are captured by `shift` and delimited by `reset`. More specifically, `shift` captures the current context up to the nearest dynamically enclosing `reset`, replaces it *abortively* with the empty delimited context [_], and binds `f` to the captured delimited context as a functional value. For example, the program

```
(cons 'a (reset (cons 'b
  (shift f (cons 1 (f (f (cons 'c '()))))))))
```

evaluates to the list `(a 1 b b c)`, because `shift` binds `f` to the value `(lambda (x) (reset (cons 'b x)))`, which represents the delimited context `[(cons 'b _)]` captured by `shift`. At the same time, `shift` also removes that context from evaluation—in other words, it *aborts* the current computation up to the delimiting `reset`—so the result is not `(a b 1 b b c)`.

Continuations have found a wide variety of applications. Delimited continuations, in particular, have been used in direct-style representations of monads [23–25], partial evaluation [8, 17, 26, 38, 52], Web interactions [29, 43, 44], mobile code [50], the CPS transform itself [11–13], and linguistics [3, 47]. However, the proliferation of delimited control operators remains a source of confusion for users and work for implementors. Even though all delimited control operators in the literature can be implemented using `call/cc` and mutable state, we would prefer a *direct* implementation—that is, an implementation that does not rely on undelimited continuations—in hope of reaping the efficiency gains recently shown by Gasbichler and Sperber [28] with their direct implementation. Unfortunately, Gasbichler and Sperber only implement Felleisen's `control` and `prompt` [18, 19] and (from there) Danvy and Filinski's `shift` and `reset` [11–13], not other proposed operators that allow an expression to capture its context beyond an arbitrary number of dynamically enclosing delimiters [30–32, 45]. Although it is clear that the latter operators can macro-express[1] the former ones in pure Scheme without `call/cc` or `set!`, the converse "seems not to be known" [30, 31]. Hence it is unclear how an improved implementation of `shift` and `reset`, such as Gasbichler and Sperber's, can help us implement other control operators better.

Because the "static" control operators `shift` and `reset` correspond closely to a standard CPS transform [12], to macro-express other, "dynamic" control operators in terms of `shift` and `reset` is to extend that transform. In the literature, dynamic control operators like `control` and `prompt` are often treated, as if by necessity, using a non-standard CPS transform in which continuations are represented as sequences of activation frames [21, 22, 42]. By contrast, we show in this paper that a standard CPS transform suffices, as one might expect from Filinski's representation of monads in terms of `shift` and `reset` [23–25] (see Section 3.1). What distinguishes dynamic control operators is that the continuation is *recursive*. Thus, in a language supporting recursion like (pure) Scheme, `shift` and `reset` can macro-express the other control operators after all. As a consequence, any direct implementation of `shift` and `reset`, such as Gasbichler and Sperber's, gives rise to a direct implementation of the other operators.[2] Moreover, because our translation of all these operators extends a standard CPS transform, they can be incorporated into CPS-based language implementations.

[1] By "macro-express" we mean Felleisen's notion of macro expressibility [20], but we surround each program by a "`top-level`" construct to mark its syntactic top level. We also impose an additional requirement: given any space consumption bound s, there must exist another space consumption bound s', such that every program within s translates to a program within s'. This requirement is intended to rule out

- implementing delimited continuations by capturing undelimited ones; and
- keeping mutable state by modeling memory in a single storage cell, which `shift` and `reset` can simulate (while accumulating garbage in the simulated store).

Space consumption can be defined along the lines of Clinger [5], for an abstract machine such as Biernacka et al.'s for `shift` and `reset` [4].

[2] A reviewer suggests that Gasbichler and Sperber's technique can be easily adapted to other control operators. For example, to implement the (dynamic) `shift0` operator below, it seems that one need only replace the reset flag in every frame with a reset count, and decrement it after shifting. Given how many delimited control operators have been (and will be?) proposed—several, like `cupto` [30, 31], are related but not identical to the four considered in this paper—macro-expressibility results like ours are attractive because they do not require changing the Scheme implementation at all before new operators can be introduced.

The rest of this paper is structured as follows. Section 2 introduces the static control operators `shift` and `reset`, and their dynamic counterparts. Section 3 expresses dynamic control in terms of static control with recursive continuations. Section 4 then concludes and mentions additional related work.

2 A tale of two resets

Danvy and Filinski's `shift` and `reset` [11–13] can be defined operationally as well as denotationally. Operationally, we can specify transition rules in the style of Felleisen [18]:[3]

$$M[(\texttt{reset}\ V)] \rhd M[V] \tag{2}$$

$$M[(\texttt{reset}\ C[(\texttt{shift f}\ E)])] \rhd M[(\texttt{reset}\ E')]$$
$$\text{where } E' = E\{\texttt{f} \mapsto (\texttt{lambda (x) (reset}\ C[\texttt{x}]))\} \tag{3}$$

Here V stands for a value, C stands for an evaluation context that does not cross a `reset` boundary, and M stands for an evaluation context that may cross a `reset` boundary:

Values	$V ::= (\texttt{lambda (x)}\ E) \mid \cdots$	(4)
Contexts	$C[\ ] ::= [\ ] \mid C[([\ ]\ E)] \mid C[(V\ [\ ])] \mid \cdots$	(5)
Metacontexts	$M[\ ] ::= C[\ ] \mid M[(\texttt{reset}\ C[\ ])]$	(6)

Denotationally, we can specify a CPS transform to map programs that use `shift` and `reset` to programs that do not. The core of this transform is shown in Figure 1; its first three lines are what this paper means by "a standard (call-by-value) CPS transform".[4]

As Danvy and others have long observed [10], the syntactic definitions above of contexts and metacontexts are not rabbits out of hats. Rather, contexts are defunctionalized representations of the continuation functions in Figure 1.

Contexts of the form:	represent continuations of the form:
$[\ ]$	`(lambda (v) v)`
$C[([\ ]\ E)]$	`(lambda (f)` $(E'$ `(lambda (x) ((f x)` C'`))))`
$C[(V\ [\ ])]$	`(lambda (x) ((`V' `x)` C'`))`

Similarly, metacontexts (such as `(reset (f (reset (g [ ]))))`) are defunctionalized representations of the implicit metacontinuations in Figure 1—that is, of the continuations that can be made explicit by CPS-transforming the right hand side of Figure 1.

The CPS transform relates not just terms but also types between the source and target languages. If the source program is a well-typed term in, say, the simply-typed λ-calculus, then the output of the transform is also well-typed in the simply-typed λ-calculus: every source type at the top level or to the right of a function arrow is

[3] To help the exposition below, these transition rules do not handle the case when a `shift` term is evaluated with no dynamically enclosing `reset`. Danvy and Filinski's original proposal amounts here to enclosing the entire program in a top-level `reset`.

[4] The right-hand-sides for `shift` and `reset` in Figure 1 contain non-tail calls, as do (18–19) in Section 3.1 below. Thus these equations do not really constitute a CPS transform, only a continuation-*composing*-style transform that extends a standard CPS transform on the pure λ-calculus. In particular, the output of this transform is sensitive to the evaluation order of the target language. Danvy and Filinski [12] regain CPS by CPS-transforming the output of this transform a second time. We can do so but need not, since by Section 3.2 our equations' right-hand-sides will be in CPS again, with all arguments pure.

$$\overline{\texttt{x}} = \texttt{(lambda (c) (c x))}$$
$$\overline{\texttt{(lambda (x) }E\texttt{)}} = \texttt{(lambda (c) (c (lambda (x) }\overline{E}\texttt{)))}$$
$$\overline{\texttt{(}E_1\ E_2\texttt{)}} = \texttt{(lambda (c) (}\overline{E_1}\texttt{ (lambda (f) (}\overline{E_2}\texttt{ (lambda (x) ((f x) c))))))}$$
$$\overline{\texttt{(reset }E\texttt{)}} = \texttt{(lambda (c) (c (}\overline{E}\texttt{ (lambda (v) v))))}$$
$$\overline{\texttt{(shift f }E\texttt{)}} = \texttt{(lambda (c) (let ((f (lambda (x) (lambda (c2) (c2 (c x))))))} \quad \texttt{(}\overline{E}\texttt{ (lambda (v) v))))}$$

Figure 1. A continuation-passing-style transform for `shift` and `reset`

mapped to a type of the form $(\tau \to \omega_1) \to \omega_2$, where ω_1 and ω_2 are *answer types* [39]. Moreover, the type system of the target language can be regarded as a type system for the source language. For example, the expression

```
(shift f (if (f 'a) 1 2))
```

translates to a term of the type $(\mathsf{Sym} \to \mathsf{Bool}) \to \mathsf{Int}$. In words, the expression can appear in a context that produces a boolean when plugged with a symbol, and produce an integer as the final answer. We can take such descriptions as the types of source terms, as Danvy and Filinski [11] do. They write the typing judgment

$$\cdot, \mathsf{Bool} \vdash \texttt{(shift f (if (f 'a) 1 2))} : \mathsf{Sym}, \mathsf{Int} \qquad (7)$$

to mean that the expression behaves locally like a symbol, but incurs a control effect that changes the answer type from Bool to Int.

The transition rule (3) for `shift` mentions `reset` twice on its right hand side. On the first line, the `reset` that delimits the captured context is preserved after the capture, so the context from a single `reset` outward is protected from manipulation by any number of dynamically enclosed `shift` invocations. Informally speaking, `reset` makes any piece of code appear pure to the outside, that is, devoid of control effects. On the second line, the captured context is surrounded by `reset`, so `f` is bound to a pure function.

Neither occurrence of `reset` on the right hand side of (3) is accidental; they are necessary for the operational semantics to match the transform in Figure 1. Despite the appeal of this match, many other delimited control operators have been proposed (historically, both before and after Danvy and Filinski's work) that remove one or both occurrences of `reset` on the right hand side of (3). Three such variations on `shift` are possible, namely `control`, `shift0`, and `control0` below.

$$M[\texttt{(reset }C[\texttt{(control f }E\texttt{)}]\texttt{)}] \rhd M[\texttt{(reset }E'\texttt{)}] \quad \text{where } E' = E\{\texttt{f} \mapsto \texttt{(lambda (x) }C[\texttt{x}]\texttt{)}\} \qquad (8)$$

$$M[\texttt{(reset }C[\texttt{(shift0 f }E\texttt{)}]\texttt{)}] \rhd M[E'] \quad \text{where } E' = E\{\texttt{f} \mapsto \texttt{(lambda (x) (reset }C[\texttt{x}]\texttt{))}\} \qquad (9)$$

$$M[\texttt{(reset }C[\texttt{(control0 f }E\texttt{)}]\texttt{)}] \rhd M[E'] \quad \text{where } E' = E\{\texttt{f} \mapsto \texttt{(lambda (x) }C[\texttt{x}]\texttt{)}\} \qquad (10)$$

Felleisen's `control` operator [18, 19, 21, 22, 49], the first delimited control operator in the literature, captures a delimited context without surrounding it with `reset`, so `f` may operate on the contexts in which it is subsequently invoked. The difference between `shift` and `control` can be observed as follows: the program

```
(reset (let ((y (shift f (cons 'a (f '())))))
         (shift g y)))
```

evaluates to `(a)`,[5] whereas the program

```
(reset (let ((y (control f (cons 'a (f '())))))
         (control g y)))
```

evaluates to `()`.[6] Sitaram's `fcontrol` [48] is closely related to `control` in nature. These authors refer to `reset` as `prompt`, `run`, `#`, or `%`.

The `shift0` operator captures a delimited context like `shift` does, but removes the delimiting `reset`. For example, the program

```
(reset (cons 'a
         (reset (shift f (shift g '())))))
```

evaluates to `(a)`,[7] whereas the program

```
(reset (cons 'a
         (reset (shift0 f (shift0 g '())))))
```

evaluates to `()`.[8] Danvy and Filinski [11] consider this `shift0` operator briefly. Also, Hieb and Dybvig's `spawn` [32] can be thought of as a `reset` that, each time it is invoked to insert a new delimiter, creates a specific `shift0` operator for that new delimiter.

The `control0` operator is like `control` but removes the delimit-

[5] The reduction sequence begins:

```
(reset (cons 'a
         ((lambda (x)
            (reset (let ((y x)) (shift g y))))
          '())))
(reset (cons 'a
         (reset (let ((y '())) (shift g y)))))
(reset (cons 'a (reset (shift g '()))))
(reset (cons 'a (reset '())))
```

Here `shift f` introduces a `reset` under the `lambda`, which stops `shift g` from capturing `cons 'a`.

[6] The reduction sequence begins:

```
(reset (cons 'a
         ((lambda (x)
            (let ((y x)) (control g y)))
          '())))
(reset (cons 'a
         (let ((y '())) (control g y))))
(reset (cons 'a (control g '())))
(reset '())
```

Here `control f` allows `control g` to capture `cons 'a`.

[7] The reduction sequence begins:

```
(reset (cons 'a (reset (shift g '()))))
(reset (cons 'a (reset '())))
```

[8] The reduction sequence is:

```
(reset (cons 'a (shift0 g '())))
'()
```

ing `reset`. It is essentially Gunter et al.'s `cupto` [30, 31] stripped down to one prompt variable, and closely related to Queinnec and Serpette's `splitter` [45].

Described operationally as in (8–10), these variations on `shift` seem like minor changes with little sense of purpose. Because adding `reset` is easy, `control` and `shift0` can obviously macro-express `shift`, and `control0` can macro-express them all, without `call/cc` or mutable state. The opposite direction—whether `shift` can simulate any of its `reset`-removed cousins, for example—"seems not to be known" to Gunter et al. [30, 31]. Since no version of `shift` is clearly "right", Gunter et al. choose to take `control0` as primitive.

Concomitant with the apparent difficulty of using `shift` to simulate the other control operators is an apparent difficulty of devising denotational semantics for these operators under a standard CPS transform. More precisely, unlike with `shift`, it is unclear how to translate `control`, `shift0`, or `control0` away using a transform that coincides on pure λ-terms with the first three lines of Figure 1, where contexts are represented as continuation functions. Instead, semantics for these operators in the literature either rely on complex mutable data structures (in essence defining the operators by implementing them in Scheme) or represent contexts as sequences of activation frames,[9] termed *abstract continuations* [21, 22, 42]. Standard continuation semantics is declared "inadequate" [21] and "insufficient" [22],[10] as `control` is said to "admit no such simple static interpretation" [13]. Such claims are surprising in hindsight of Filinski's representation of monads in terms of `shift` and `reset` [23–25]—surely even including `control0` in a language would not disqualify it from Moggi's notions of computation [40]?

Danvy and Filinski [11–13] informally classify their `shift` and `reset` operators as *lexical* and *static*, and other delimited control operators such as `control` as *dynamic*. They use these words to draw an analogy to lexical versus dynamic scoping for variables: roughly speaking, `shift` and `reset`, unlike the other operators, can be defined and implemented without traversing arbitrarily deeply into data structures at run-time. The next section shows that, as soon as we allow traversing arbitrarily deeply into data structures at run-time, dynamic control operators can be treated with the same transform as static ones. That is, continuation semantics is sufficient after all, as long as the continuation can be recursive.[11]

Our development below of recursive continuations is guided by recursive types. For example, if α is a type, then the type $\mathsf{List}\ \alpha$ of singly-linked α-lists can be defined by

$$\mathsf{List}\ \alpha \;=\; 1+\alpha\times\mathsf{List}\ \alpha, \tag{11}$$

where 1 is the unit type and $\times$ constructs product types. For brevity, we take the unfolding of a recursive type to give not just isomorphic but in fact equivalent types. For example, (11) states an equation between types, not just an isomorphism. To use terms coined by Crary et al. [6, 27], this paper shows *equi-recursive* types, but *iso-recursive* types can be used too.

3 Recursive continuations

In this central section of the paper, we treat dynamic control operators by extending the standard CPS transform, and by translating them into `shift` and `reset`. The key to these treatments is to represent delimited contexts as functions whose types are recursive: When a delimited context is captured with a dynamic control operator, then invoked, it may take control over the delimited context at the invocation site. Hence, the former context must take the latter context as an argument in our CPS transform. Roughly speaking, then, the type of contexts must mention itself, that is, be recursive.

Let us first review delimited contexts captured by `shift` and `reset`. The CPS transform in Figure 1 represents a delimited context as a continuation, that is, a function of type $\tau\to\omega$. Danvy and Filinski identify τ with the type of the intermediate result (that is, the hole in the context) and ω with the type of the answer (that is, the context once plugged). For comparison with other control operators below, we define the types

$$\mathsf{Context}\ \tau\ \omega \;=\; \tau\to\omega, \tag{12}$$
$$\mathsf{Answer}\ \omega \;=\; \omega, \tag{13}$$

such that

$$\mathsf{Context}\ \tau\ \omega \;=\; \tau\to\mathsf{Answer}\ \omega. \tag{14}$$

To take an example, the delimited context [`(< 1 _)`] takes the type $\mathsf{Context\ Int\ Bool}$ (or equivalently, $\mathsf{Int}\to\mathsf{Bool}$) when captured with `shift`, because plugging the hole _ with an integer gives an answer that is a boolean. In other words, the function

```
(lambda (x)
  (reset (< 1 x)))
```

(which represents that context, as captured by `shift`) maps integers to booleans. For another example, the delimited context [`(let ((y _)) (shift g (< 1 y)))`], when captured by `shift`, also has the type $\mathsf{Context\ Int\ Bool}$. In other words, the function

```
(lambda (x)
  (reset (let ((y x)) (shift g (< 1 y)))))
```

(which represents that context, as captured by `shift`) also maps integers to booleans. In fact, these two contexts captured by `shift` are observationally equivalent, because the `shift g` above has only the empty delimited context [_] to capture.

3.1 control

The context [`(let ((y _)) (control g (< 1 y)))`] captured with `control` is not equivalent to [`(< 1 _)`], because the function

```
(lambda (x)
  (let ((y x)) (control g (< 1 y))))
```

(which represents the first context, as captured by `control`) wipes out its surrounding delimited context when invoked, whereas the function

```
(lambda (x)
  (< 1 x))
```

[9] Or an algebra thereof.

[10] A reviewer states that these declarations are objections to the non-tail calls in Figure 1 (as continuation semantics for `shift` and `reset`) and (18–19) in Section 3.1 (as continuation semantics for `control` and `prompt`). However, see footnote 4.

[11] One way to see the connection between dynamic control operators and recursive continuations may be to observe how the following program enters an infinite loop.

```
(prompt (begin (control f (begin (f 0) (f 0)))
               (control f (begin (f 0) (f 0)))))
```

(which represents the second context, as captured by `control`) does not. In general, when a delimited context captured by `control` is invoked, it may further capture the surrounding delimited context (up to the nearest dynamically enclosing `reset`) at the point of invocation. Thus a delimited context captured by `control`, unlike one captured by `shift`, is not a function from an intermediate result (with which to plug a hole) to a final answer. Rather, a `control`-captured context can be thought of as a function from an intermediate result *and any surrounding delimited context* to a final answer. The surrounding context may be the empty context [_] (if the captured context is invoked immediately within `reset`) or not empty. Accordingly, we let a delimited context captured by `control` whose hole is of type τ and answer is of type ω take the type $\mathsf{Context}'\ \tau\ \omega$, where

$$\mathsf{Context}'\ \tau\ \omega \;=\; \tau \to \mathsf{Maybe}\,(\mathsf{Context}'\ \omega\ \omega) \to \omega. \qquad (15)$$

In this recursive type definition, $\mathsf{Maybe}\ \alpha$ means either an α-value or the special token `#f`, like the discriminated union types `Maybe a` in Haskell. We use `#f` to represent the empty surrounding context.

The function `send` below plugs an intermediate answer `v` (of type ω) into a delimited context `mc` (of type $\mathsf{Maybe}\,(\mathsf{Context}'\ \omega\ \omega)$) by calling `mc` with `v` and the trivial delimited context `#f`. If `mc` is the special token `#f`, then we are plugging `v` into the empty context, so the final answer is just `v`.

```
(define (send v)
  (lambda (mc) (if mc ((mc v) #f) v)))
```

This function is of type $\mathsf{Context}'\ \omega\ \omega$: it is itself a delimited context, namely the empty one. If our target language lets us compare values against `send` (even intensionally using `eq?`, say), then we can do so rather than comparing values against `#f`, and drop our use of Maybe. That is, we could implement `send` as

```
(define (send v)
  (lambda (mc)
    (if (eq? send mc) v ((mc v) send))))
```

but do not, for clarity.

When two `shift`-captured contexts are composed as functions at the source level, the result corresponds to concatenating continuations by function composition in the target language. By contrast, to concatenate `control`-captured contexts of the recursive type defined in (15), we define a recursive function, of type $\big(\mathsf{Context}'\ \tau\ \omega \times \mathsf{Maybe}\,(\mathsf{Context}'\ \omega\ \omega)\big) \to \mathsf{Context}'\ \tau\ \omega$:

```
(define (compose c mc1)
  (if mc1 (lambda (v)
            (lambda (mc2)
              ((c v) (compose mc1 mc2))))
          c))
```

According to (15), the type $\mathsf{Context}'\ \tau\ \omega$ is a function type, and τ only appears in its domain, not codomain. In other words, a context captured by `control` whose hole type is τ has the function type of a τ-continuation, just like delimited contexts captured by `shift`, except for the recursive answer type $\mathsf{Answer}'\ \omega$ defined by

$$\begin{aligned} \mathsf{Answer}'\ \omega \;&=\; \mathsf{Maybe}\,(\mathsf{Context}'\ \omega\ \omega) \to \omega \\ &=\; \mathsf{Maybe}\,(\omega \to \mathsf{Answer}'\ \omega) \to \omega, \qquad (16) \end{aligned}$$

such that

$$\begin{aligned} \mathsf{Context}'\ \tau\ \omega \;&=\; \tau \to \mathsf{Answer}'\ \omega \\ &=\; \mathsf{Context}\ \tau\ (\mathsf{Answer}'\ \omega). \qquad (17) \end{aligned}$$

Thus $\mathsf{Context}'$ can be written in terms of Context! Hence, delimited contexts captured by `control` can be represented as ordinary, if recursive, continuations. The equations below extend the first three lines of Figure 1 to `control`. It maps every source type τ, at the top level or to the right of a function arrow, to a type of the form $(\tau \to \mathsf{Answer}'\ \omega) \to \mathsf{Answer}'\ \omega$. To distinguish the `reset` for `control` here from the `reset` for `shift` above, we write `prompt` instead of `reset`.

$$\overline{\texttt{(prompt }E\texttt{)}} =$$

```
  (lambda (c) (c ((Ē send) #f)))                         (18)
```

$$\overline{\texttt{(control f }E\texttt{)}} =$$

```
    (lambda (c1)
      (lambda (mc1)
        (let ((f (lambda (x)
                   (lambda (c2)
                     (lambda (mc2)
                       (((compose c1 mc1) x)
                        (compose c2 mc2)))))))
          ((Ē send) #f))))                               (19)
```

Because this transform extends a standard call-by-value CPS transform on the pure λ-calculus, it shows how to treat `control` and `prompt` as operations in the continuation monad (with answer type $\mathsf{Answer}'\ \omega$). Then, because `shift` and `reset` expresses all operations in the continuation monad, we can define `control` and `prompt` in direct style as macros in terms of `shift` and `reset`.

```
(define-syntax prompt
  (syntax-rules ()
    ((_ e) ((reset (send e)) #f))))

(define-syntax control
  (syntax-rules ()
    ((_ f e)
     (shift c1
       (lambda (mc1)
         (let ((f (lambda (x)
                    (shift c2
                      (lambda (mc2)
                        (((compose c1 mc1) x)
                         (compose c2 mc2)))))))
           ((reset (send e)) #f)))))))
```

These source-level macros correspond directly to the target-level equations (18–19), except:

- Where the target-level equations abstract over a continuation argument, the source-level macros use `shift` rather than `lambda`.
- Where the equations pass the continuation `send` to $\overline{E}$, the macros say (`reset (send` E`)`), so as to place E in the delimited context [(`send _`)].

This implementation of `control` and `prompt` uses neither `call/cc` nor mutable state; in particular, it does not capture any continuation beyond the outermost delimiting `prompt`.

Another way to view the same definitions in hindsight is to recognize that a denotational semantics given by Felleisen et al. [21, Section 4] encodes `control` and `prompt` in a monad that maps each type τ to the type $(\tau \to \mathsf{Answer}'\ \omega) \to \omega$. This monad is not the continuation monad, because the answer types $\mathsf{Answer}'\ \omega$ and ω are different; hence, Felleisen et al.'s equations for their denotational semantics do not give a standard CPS transform. Nevertheless, we can still use Filinski's representation of monads in terms of `shift` and `reset` [23–25] to represent `control` and `prompt`—essentially as above, in fact. As an anonymous reviewer hints, this observation is one way to show our definitions to correctly implement `control` and `prompt`.

Sitaram and Felleisen [49] implement `control` and `prompt` in terms of `call/cc` in Scheme. That implementation uses both `call/cc` and mutable state. Our implementation of `control` and `prompt` using `shift` and `reset` can be composed with Filinski's implementation of `shift` and `reset` using `call/cc` [23] to yield a more modular implementation of `control` and `prompt` using `call/cc`. Sitaram and Felleisen's implementation maintains a global, mutable *run-stack*. The run-stack is comprised of *sub-stacks*, one for each dynamically active `prompt`. Each sub-stack is a list of invocation points (that is, undelimited continuations captured by `call/cc`). These data structures can be correlated with our implementation: The run-stack is a sequence of "`mc`" functions (of type $\mathsf{Maybe}\,(\mathsf{Context}'\ \omega\ \omega)$), one for each dynamically active `prompt`. Each `mc` function is a sub-stack, the result of concatenating `control`-captured contexts using `compose`.

3.2 `shift0`

When `shift0` captures a delimited context, it does not replace it with the trivial delimited context as `shift` does. Instead, it removes the captured context along with its delimiting `reset`, exposing the next-outer delimited context up to the next-nearest dynamically enclosing `reset`. With `shift0` in the language, `reset` is not idempotent: `(reset` E`)` is not equivalent to `(reset (reset` E`))`, because each `reset` only "defends against" one `shift0`. For example, the program

```
(reset (cons 'a
         (reset (shift0 f (shift0 g '())))))
```

evaluates to `()`, but the program

```
(reset (cons 'a
         (reset
           (reset (shift0 f (shift0 g '()))))))
```

evaluates to `(a)`.

Because `shift0` removes the delimiting `reset` when capturing a delimited context, the context

```
[(let ((y _))
   (shift0 f (shift0 g (< 1 y))))]
```

captured with `shift0` is not equivalent to the contexts

```
[(let ((y _)) (shift0 g (< 1 y)))]
[(< 1 _)]
```

captured with `shift0`. That is, the function

```
(lambda (x)
  (reset (let ((y x))
           (shift0 f (shift0 g (< 1 y))))))
```

wipes out its surrounding delimited context when invoked, whereas the functions

```
(lambda (x)
  (reset (let ((y x)) (shift0 g (< 1 y)))))
(lambda (x)
  (reset (< 1 x)))
```

do not.

Appendix C of Danvy and Filinski's technical report [11] considers this variation on `shift` briefly. They model it denotationally by passing around a list of delimited contexts, which can be thought of as a sequence of activation frames, except each frame corresponds to a `reset` rather than a function call.[12] In our formulation, a delimited context captured by `shift0` whose hole type is τ and whose answer type is ω has the type $\mathsf{Context}_0\ \tau\ \omega$, where

$$\mathsf{Context}_0\ \tau\ \omega \ = \ \tau \to \mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega) \to \omega. \tag{20}$$

In this recursive type definition, $\mathsf{List}\ \alpha$ means a singly-linked list of α-values, either a `cons` cell or the empty list `()`. A list of type $\mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega)$ contains delimited contexts from innermost to outermost, separated by control delimiters.

The function `propagate` below plugs an intermediate answer `v` (of type ω) into a list of contexts `lc` (of type $\mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega)$) by calling the head of `lc` with `v` and the tail of `lc`. If `c` is empty, then the final answer is simply `v`.

```
(define (propagate v)
  (lambda (lc)
    (if (null? lc) v
                   (((car lc) v) (cdr lc)))))
```

This function is of type $\mathsf{Context}_0\ \omega\ \omega$: it is itself a delimited context, namely the empty one.

Like the type $\mathsf{Context}'\ \tau\ \omega$ in Section 3.1, $\mathsf{Context}_0\ \tau\ \omega$ is a function type in which τ only appears in the domain. Hence a delimited context captured by `shift0` is just like one captured by `shift`, except the answer type $\mathsf{Answer}_0\ \omega$ of the continuation is recursive, defined by

$$\begin{aligned} \mathsf{Answer}_0\ \omega \ &= \ \mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega) \to \omega \\ &= \ \mathsf{List}\,(\omega \to \mathsf{Answer}_0\ \omega) \to \omega, \end{aligned} \tag{21}$$

such that

$$\begin{aligned} \mathsf{Context}_0\ \tau\ \omega \ &= \ \tau \to \mathsf{Answer}_0\ \omega \\ &= \ \mathsf{Context}\ \tau\,(\mathsf{Answer}_0\ \omega). \end{aligned} \tag{22}$$

Thus $\mathsf{Context}_0$ can be written in terms of $\mathsf{Context}$. Therefore, just as with `control`, delimited contexts captured by `shift0` can be represented as ordinary continuations. Following the Appendix C mentioned above, the equations below extend the first three lines of Figure 1 to a CPS transform for `shift0`. It maps every source type τ, at the top level or to the right of a function arrow, to a type of the form $(\tau \to \mathsf{Answer}_0\ \omega) \to \mathsf{Answer}_0\ \omega$. To distinguish the `reset` for `shift0` here from the `reset` for `shift` above, we write `reset0` instead of `reset`.

$$\begin{array}{l} \overline{(\texttt{reset0}\ E)} = \\ \quad \texttt{(lambda (c)} \\ \quad\quad \texttt{(lambda (lc)} \\ \quad\quad\quad \texttt{((}\overline{E}\ \texttt{propagate) (cons c lc))))} \end{array} \tag{23}$$

[12] Johnson and Duggan [34] add control facilities to the programming language GL that provide power similar to that of `shift0` and `reset`, but they make each function call delimit the context (like Landin's SECD machine [9, 10, 37]), so their frames do correspond to function calls.

$\overline{\texttt{(shift0 f }E\texttt{)}} =$

```
(lambda (c1)
  (lambda (lc)
    (let ((f (lambda (x)
               (lambda (c2)
                 (lambda (lc)
                   ((c1 x) (cons c2 lc)))))))
      ((Ē (car lc)) (cdr lc)))))
```

$$(24)$$

As in Section 3.1, these equations[13] can be turned into a direct implementation of `shift0` and `reset0` in terms of `shift` and `reset` that neither captures undelimited continuations nor keeps mutable state.

3.3 `control0`

The `control0` operator removes both occurrences of `reset` on the right hand side of (3); it combines the dynamic properties of `control` and `shift0`. It is thus not surprising that we can treat `control0` with recursive continuations and the CPS transform by combining the ideas from Sections 3.1–2.

A delimited context captured by `control0`, with hole type τ and answer type ω, has the type

$$\begin{aligned}\mathsf{Context}_0'\ \tau\ \omega \;&=\; \tau \to \mathsf{Maybe}\,(\mathsf{Context}_0'\ \omega\ \omega) \to \\ &\qquad \mathsf{List}\,(\mathsf{Context}_0'\ \omega\ \omega) \to \omega, \qquad (25)\end{aligned}$$

in which τ only appears in the domain. A delimited context captured by `control0` is thus just like one captured by `shift` with the recursive answer type

$$\begin{aligned}\mathsf{Answer}_0'\ \omega \;&=\; \mathsf{Maybe}\,(\mathsf{Context}_0'\ \omega\ \omega) \to \\ &\qquad \mathsf{List}\,(\mathsf{Context}_0'\ \omega\ \omega) \to \omega \\ &=\; \mathsf{Maybe}\,(\omega \to \mathsf{Answer}_0'\ \omega) \to \\ &\qquad \mathsf{List}\,(\omega \to \mathsf{Answer}_0'\ \omega) \to \omega, \qquad (26)\end{aligned}$$

such that

$$\begin{aligned}\mathsf{Context}_0'\ \tau\ \omega \;&=\; \tau \to \mathsf{Answer}_0'\ \omega \\ &=\; \mathsf{Context}\ \tau\ (\mathsf{Answer}_0'\ \omega). \qquad (27)\end{aligned}$$

Thus $\mathsf{Context}_0'$ can be written in terms of $\mathsf{Context}$. Informally speaking, the Maybe part of the types above keeps track of the delimited context within the nearest dynamically enclosing `reset`, and the List part keeps track of the delimited contexts beyond that `reset`.

The trivial delimited context of type $\mathsf{Context}_0'\ \omega\ \omega$ is the function `send-propagate` below, which combines `send` and `propagate`.

```
(define (send-propagate v)
  (lambda (mc)
    (if mc ((mc v) #f)
           (lambda (lc)
             (if (null? lc) v
                            ((((car lc) v) #f)
                             (cdr lc)))))))
```

To compose delimited contexts captured by `control0`, we can simply use the code for `compose` above, because—although it is created for `control`—it also has the type

$$\big(\mathsf{Context}_0'\ \tau\ \omega \times \mathsf{Maybe}\,(\mathsf{Context}_0'\ \omega\ \omega)\big) \to \mathsf{Context}_0'\ \tau\ \omega. \quad (28)$$

Finally, we can use `send-propagate` and `compose` to define an ordinary CPS transform for `control0`. Here we write `prompt0` instead of `reset` to mean the `reset` for `control0`.

$\overline{\texttt{(prompt0 }E\texttt{)}} =$

```
(lambda (c)
  (lambda (mc)
    (lambda (lc)
      (((Ē send-propagate) #f)
       (cons (compose c mc) lc)))))
```

$$(29)$$

$\overline{\texttt{(control0 f }E\texttt{)}} =$

```
(lambda (c1)
  (lambda (mc1)
    (lambda (lc)
      (let ((f (lambda (x)
                 (lambda (c2)
                   (lambda (mc2)
                     (((compose c1 mc1) x)
                      (compose c2 mc2)))))))
        (((Ē (car lc)) #f) (cdr lc))))))
```

$$(30)$$

This CPS transform maps every source type τ, at the top level or to the right of a function arrow, to a type of the form $(\tau \to \mathsf{Answer}_0'\ \omega) \to \mathsf{Answer}_0'\ \omega$. Again, these CPS equations can be turned into an implementation of `control0` and `prompt0` using `shift` and `reset` that neither captures undelimited continuations nor keeps mutable state.

4 Conclusion and related work

This paper presents the first CPS transform for dynamic delimited control operators, including Felleisen's `control` and `prompt`, that is consistent with a standard CPS transform. We have shown that Danvy and Filinski's static operators `shift` and `reset` are just as expressive as dynamic ones. For a delimited control operator to be dynamic is for it to require recursive continuations.

Now that we know how to implement dynamic operators in terms of `shift` and `reset` without capturing undelimited continuations or keeping mutable state, direct implementations of `shift` and `reset` like Gasbichler and Sperber's [28] give rise to direct implementations of dynamic operators. Moreover, because our CPS transform extends a standard one, it can be incorporated into CPS-based language implementations.

Besides explicating dynamic control operators, recursive continuations are also useful in practical programming. For example, the iterative interaction pattern between a coroutine and its environment is reflected in a recursive continuation, specifically its recursive answer type [25, Section 4.2], which can be depicted graphically as a flowchart. Two special cases of such interactions are:

- the interaction between a Web server and user agents [16, 29, 43, 44]; and
- the interaction between a cursor iterating over a collection and its client [36], as epitomized in the classic same-fringe problem.

Another potential application of recursive continuations lies in Balat et al.'s type-directed partial evaluator for the λ-calculus with products and sums [2], which computes normal forms for λ-terms

[13] Now in CPS; see footnote 4. Expressions like `((Ē propagate) (cons c lc))` may appear to contain a non-tail call, but should be regarded as a curried call with two arguments.

under $\beta\eta$-equivalence. To normalize terms that use sums, Balat et al.'s algorithm uses Gunter et al.'s `cupto` operator [30, 31], rather than `shift` as in previous work by Balat and Danvy [1]. As Balat et al.'s algorithm evaluates a term, it keeps a list of possible scope locations at which future `case` expressions may be inserted, in the form of prompts for `cupto`. (By contrast, Balat and Danvy's earlier algorithm using `shift` only considers one scope location at which to insert a `case` expression.) If `cupto` is replaced by `shift` with a recursive continuation, then that list of prompts would be pleasingly identified with the stack of control points that Gunter et al. use to implement `cupto` in the first place. A direct implementation of `cupto` or `shift` would also make the algorithm more efficient.

5 Acknowledgements

This paper would not be written without the help and encouragement of Oleg Kiselyov. Thanks also to Chris Barker, John Clements, Olivier Danvy, Matthias Felleisen, Andrzej Filinski, Shriram Krishnamurthi, Stuart Shieber, Sam Tobin-Hochstadt, and six anonymous reviewers for ICFP 2004 and this workshop. This work is supported by the United States National Science Foundation Grant BCS-0236592.

References

[1] Balat, Vincent, and Olivier Danvy. 2002. Memoization in type-directed partial evaluation. In *Proceedings of GPCE 2002: 1st ACM conference on generative programming and component engineering*, ed. Don S. Batory, Charles Consel, and Walid Taha, 78–92. Lecture Notes in Computer Science 2487, Berlin: Springer-Verlag.

[2] Balat, Vincent, Roberto Di Cosmo, and Marcelo Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL '04: Conference record of the annual ACM symposium on principles of programming languages*, 64–76. New York: ACM Press.

[3] Barker, Chris. 2004. Continuations in natural language (extended abstract). In [51], 1–11.

[4] Biernacka, Małgorzata, Dariusz Biernacki, and Olivier Danvy. 2004. An operational foundation for delimited continuations. In [51], 25–33.

[5] Clinger, William D. 1998. Proper tail recursion and space efficiency. In *POPL '98: Conference record of the annual ACM symposium on principles of programming languages*, 174–185. New York: ACM Press.

[6] Crary, Karl, Robert Harper, and Sidd Puri. 1999. What is a recursive module? In *PLDI '99: Proceedings of the ACM conference on programming language design and implementation*, vol. 34(5) of *ACM SIGPLAN Notices*, 50–63. New York: ACM Press.

[7] Danvy, Olivier. 1994. Back to direct style. *Science of Computer Programming* 22(3):183–195.

[8] ———. 1996. Type-directed partial evaluation. In *POPL '96: Conference record of the annual ACM symposium on principles of programming languages*, 242–257. New York: ACM Press.

[9] ———. 2003. A rational deconstruction of Landin's SECD machine. Report RS-03-33, BRICS, Denmark.

[10] ———. 2004. On evaluation contexts, continuations, and the rest of the computation. In [51].

[11] Danvy, Olivier, and Andrzej Filinski. 1989. A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark. `http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz`.

[12] ———. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on Lisp and functional programming*, 151–160. New York: ACM Press.

[13] ———. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2(4):361–391.

[14] Danvy, Olivier, and Julia L. Lawall. 1992. Back to direct style II: First-class continuations. In *Proceedings of the 1992 ACM conference on Lisp and functional programming*, ed. William D. Clinger, vol. V(1) of *Lisp Pointers*, 299–310. New York: ACM Press.

[15] ———. 1996. Back to direct style II: First-class continuations. Report RS-96-20, BRICS, Denmark.

[16] Double, Chris. 2004. Partial continuations. `http://www.double.co.nz/scheme/partial-continuations/partial-continuations.html`.

[17] Dybjer, Peter, and Andrzej Filinski. 2002. Normalization and partial evaluation. In *APPSEM 2000: International summer school on applied semantics, advanced lectures*, ed. Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, 137–192. Lecture Notes in Computer Science 2395, Berlin: Springer-Verlag.

[18] Felleisen, Matthias. 1987. The calculi of λ_v-CS conversion: A syntactic theory of control and state in imperative higher-order programming languages. Ph.D. thesis, Indiana University. Also as Tech. Rep. 226, Department of Computer Science, Indiana University.

[19] ———. 1988. The theory and practice of first-class prompts. In [41], 180–190.

[20] ———. 1991. On the expressive power of programming languages. *Science of Computer Programming* 17(1–3):35–75.

[21] Felleisen, Matthias, Daniel P. Friedman, Bruce F. Duba, and John Merrill. 1987. Beyond continuations. Tech. Rep. 216, Computer Science Department, Indiana University.

[22] Felleisen, Matthias, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. 1988. Abstract continuations: A mathematical semantics for handling full jumps. In *Proceedings of the 1988 ACM conference on Lisp and functional programming*, 52–62. New York: ACM Press.

[23] Filinski, Andrzej. 1994. Representing monads. In *POPL '94: Conference record of the annual ACM symposium on principles of programming languages*, 446–457. New York: ACM Press.

[24] ———. 1996. Controlling effects. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Also as Tech. Rep. CMU-CS-96-119.

[25] ———. 1999. Representing layered monads. In *POPL '99: Conference record of the annual ACM symposium on principles of programming languages*, 175–188. New York: ACM Press.

[26] ———. 2001. Normalization by evaluation for the computational lambda-calculus. In *TLCA 2001: Proceedings of the 5th international conference on typed lambda calculi and applications*, ed. Samson Abramsky, 151–165. Lecture Notes in Computer Science 2044, Berlin: Springer-Verlag.

[27] Gapeyev, Vladimir, Michael Y. Levin, and Benjamin C. Pierce. 2000. Recursive subtyping revealed. In [33], 221–231.

[28] Gasbichler, Martin, and Michael Sperber. 2002. Final shift for call/cc: Direct implementation of shift and reset. In *ICFP '02: Proceedings of the ACM international conference on functional programming*, 271–282. New York: ACM Press.

[29] Graunke, Paul Thorsen. 2003. Web interactions. Ph.D. thesis, College of Computer Science, Northeastern University.

[30] Gunter, Carl A., Didier Rémy, and Jon G. Riecke. 1995. A generalization of exceptions and control in ML-like languages. In *Functional programming languages and computer architecture: 7th conference*, ed. Simon L. Peyton Jones, 12–23. New York: ACM Press.

[31] ———. 1998. Return types for functional continuations. `http://pauillac.inria.fr/~remy/work/cupto/`.

[32] Hieb, Robert, and R. Kent Dybvig. 1990. Continuations and concurrency. In *Proceedings of the 2nd ACM SIGPLAN symposium on principles and practice of parallel programming*, 128–136. New York: ACM Press.

[33] ICFP. 2000. *ICFP '00: Proceedings of the ACM international conference on functional programming*, vol. 35(9) of *ACM SIGPLAN Notices*. New York: ACM Press.

[34] Johnson, Gregory F., and Dominic Duggan. 1988. Stores and partial continuations as first-class objects in a language and its environment. In [41], 158–168.

[35] Kelsey, Richard, William D. Clinger, Jonathan Rees, Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, G. J. Rozas, N. I. Adams, IV, Daniel P. Friedman, Eugene Kohlbecker, Guy L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and Mitchell Wand. 1998. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation* 11(1):7–105. Also as *ACM SIGPLAN Notices* 33(9):26–76.

[36] Kiselyov, Oleg. 2004. General ways to traverse collections. `http://okmij.org/ftp/Scheme/enumerators-callcc.html`.

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

[38] Lawall, Julia L., and Olivier Danvy. 1994. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM conference on Lisp and functional programming*, 227–238. New York: ACM Press.

[39] Meyer, Albert R., and Mitchell Wand. 1985. Continuation semantics in typed lambda-calculi (summary). In *Logics of programs*, ed. Rohit Parikh, 219–224. Lecture Notes in Computer Science 193, Berlin: Springer-Verlag.

[40] Moggi, Eugenio. 1991. Notions of computation and monads. *Information and Computation* 93(1):55–92.

[41] POPL. 1988. *POPL '88: Conference record of the annual ACM symposium on principles of programming languages*. New York: ACM Press.

[42] Queinnec, Christian. 1993. A library of high-level control operators. *Lisp Pointers* 6(4):11–26.

[43] ———. 2000. The influence of browsers on evaluators or, continuations to program web servers. In [33], 23–33.

[44] ———. 2001. Inverting back the inversion of control or, continuations versus page-centric programming. Rapport de Recherche LIP6 2001/007, Laboratoire d'Informatique de Paris 6.

[45] Queinnec, Christian, and Bernard Serpette. 1991. A dynamic extent control operator for partial continuations. In *POPL '91: Conference record of the annual ACM symposium on principles of programming languages*, 174–184. New York: ACM Press.

[46] Sabry, Amr, and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation* 6(3–4):289–360.

[47] Shan, Chung-chieh. 2004. Delimited continuations in natural language: Quantification and polarity sensitivity. In [51], 55–64.

[48] Sitaram, Dorai. 1993. Handling control. In *PLDI '93: Proceedings of the ACM conference on programming language design and implementation*, vol. 28(6) of *ACM SIGPLAN Notices*, 147–155. New York: ACM Press.

[49] Sitaram, Dorai, and Matthias Felleisen. 1990. Control delimiters and their hierarchies. *Lisp and Symbolic Computation* 3(1):67–99.

[50] Sumii, Eijiro. 2000. An implementation of transparent migration on standard Scheme. In *Proceedings of the workshop on Scheme and functional programming*, ed. Matthias Felleisen, 61–63. Tech. Rep. 00-368, Department of Computer Science, Rice University.

[51] Thielecke, Hayo, ed. 2004. *CW'04: Proceedings of the 4th ACM SIGPLAN workshop on continuations*. Tech. Rep. CSR-04-1, School of Computer Science, University of Birmingham.

[52] Thiemann, Peter. 1999. Combinators for program generation. *Journal of Functional Programming* 9(5):483–525.