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

\[(\text{cons } a \{\text{cons } b \{\text{cons } c \{}()\})]\]

the continuation of \((\text{cons } c \{}())\) is to cons the symbol b, then the symbol a, onto the intermediate result. This continuation is represented by a delimited context \((\text{cons } a \{\text{cons } b \_\})\), 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

\[(\text{cons } a \{\text{cons } b \{\text{cons } c \{}()\})\]

the continuation of \((\text{cons } c \{}())\), 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 \((\text{cons } b \_\)].\]

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.

Expressions \(E ::= \cdots | \text{(shift } f \ E) | \text{(reset } E)\) (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

\[(\text{cons } a \{\text{reset } \text{cons } b
\text{(shift } f \{\text{cons } l \{f \{\text{cons } c \{}()\})\})\})\]

evaluates to the list \(\{a \ b \ b \ c\}\), because shift binds \(f\) to the value \(\text{(lambda } x\{\text{reset } \text{cons } b\ x)\}\), which represents the delimited context \([\text{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 \(\{a \ b \ b \ c\}\).

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...
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 call/cc [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 other control operators after all. As a consequence, any direct implementation of shift and reset, such as Gasbichler and Sperber’s, can help us implement other control operators better.

1 By “macro-express” we mean Filinski’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].

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 Filinski [18]:

\[ M[\{\text{reset } V\}] \triangleright M[V] \]

\[ M[\{\text{reset } C[\{\text{shift } E\}]}] \triangleright M[\{\text{reset } E′\}] \]

where \( E′ = E\{f \mapsto (\text{lambda } (x) (\text{reset } C[x]))\} \)

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 \( V ::= (\text{lambda } (x) E) \) \( \cdots \)

Contexts \( C[::] ::= [] | C[[[ E]]] | C[(V [ ])] \) \( \cdots \)

Met CONTEXTs \( M[::] ::= C[[x]] | M[\{\text{reset } C[::]\}] \)

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”.

As Danvy and others have long observed [10], the syntactic definitions above of contexts and metacategories are not rabbits out of hats. Rather, contexts are defunctionalized representations of the continuation functions in 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 \( \lambda \)-calculus, then the output of the transform is also well-typed in the simply-typed \( \lambda \)-calculus: every source type at the top level or to the right of a function arrow is

2To 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.

3The 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 \( \lambda \)-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.
mapped to a type of the form \((\tau \rightarrow \omega_1) \rightarrow \omega_2\), where \(\omega_1\) and \(\omega_2\) 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

\[(\text{shift } f \text{ (if } (f \ 'a) 1 2))\]

translates to a term of the type \((\text{Sym} \rightarrow \text{Bool}) \rightarrow \text{Int}\). 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, which it is subsequently invoked. The difference between \(\text{shift}\) and \(\text{reset}\) makes any piece of code appear pure to the outside, that is, devoid of control effects. On the second line, the captured context is preserved after the capture, so the context from a single \(\text{reset}\) outward is protected from manipulation by any number of dynamically enclosed \(\text{shift}\) invocations. Informally speaking, \(\text{reset}\) 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 \(\text{reset}\), so \(f\) is bound to a pure function.

Neither occurrence of \(\text{reset}\) 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 \(\text{reset}\) on the right hand side of (3). Three such variations on shift are possible, namely control0, \(\text{reset}\), and \(\text{control}\) below.

\[
M[\text{reset } C][\text{control } E)] | M[\text{reset } E'] \\
\text{where } E' = E[f \mapsto (\lambda (x) C[x])] \tag{8}
\]

\[
M[\text{reset } C][\text{shift0 } f ] | M[E'] \\
\text{where } E' = E[f \mapsto (\lambda (x) C[x])] \tag{9}
\]

\[
M[\text{reset } C][\text{control0 } f ] | M[E'] \\
\text{where } E' = E[f \mapsto (\lambda (x) C[x])] \tag{10}
\]

Figure 1. A continuation-passing style transform for \text{shift} and \text{reset}

Each \(\text{reset}\) operation captures a delimited context like \(\text{shift}\) does, but removes the delimiting \(\text{reset}\). For example, the program

\[
\text{(reset } (\text{cons } 'a (\text{reset } (\text{shift } f (\text{shift } g '())))))
\]

evaluates to (a)\(^5\), whereas the program

\[
\text{(reset } (\text{let } ((y \ (\text{control } f (\text{cons } 'a (f '()))))) (\text{control } g y)))
\]

evaluates to ()\(^6\). Sitaram’s \(f\)\(^{48}\) is closely related to control in nature. These authors refer to \text{reset} as prompt.run, #, or \%.

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

\[
\text{(reset } (\text{cons } 'a \text{ (reset } \text{shift } f (\text{shift } g '()))))
\]

evaluates to (a)\(^7\), whereas the program

\[
\text{(reset } (\text{cons } 'a \text{ (reset } \text{shift0 } f (\text{shift0 } g '()))))
\]

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

\[
\text{The control0 operator is like control but removes the delimited-}
\]

---

\(^5\)The reduction sequence begins:

\[
\begin{align*}
\text{(reset } & \text{cons } 'a \\
& \text{ (lambda } (x) \\
& \text{ (reset } \text{let } ((y \ x) \text{ (shift g y)))) \\
& '()))))
\end{align*}
\]

\[
\text{(reset } \text{cons } 'a \\
\text{ (reset } \text{let } ((y '())) \text{ (shift g y))))
\]

Here \text{shift} \(f\) introduces a \text{reset} under the \text{lambda}, which stops \text{shift g} from capturing cons ‘a.

\(^6\)The reduction sequence begins:

\[
\begin{align*}
\text{(reset } & \text{cons } 'a \\
& \text{ (lambda } (x) \\
& \text{ (let } ((y \ x) \text{ (control g y))) \\
& '()))))
\end{align*}
\]

\[
\text{(reset } \text{cons } 'a \text{ (control g '())))
\]

Here \text{control} \(f\) allows \text{control g} to capture cons ‘a.

\(^7\)The reduction sequence begins:

\[
\begin{align*}
\text{(reset } & \text{cons } 'a \text{ (reset } \text{shift g '()))} \\
\text{(reset } & \text{cons } 'a \text{ (reset } '()))
\end{align*}
\]

\(^8\)The reduction sequence is:

\[
\begin{align*}
\text{(reset } & \text{cons } 'a \text{ (shift0 g '()))} \\
\text{())}
\end{align*}
\]
ing reset. It is essentially Gunter et al.'s zupto [30, 31] stripped down to one prompt variable, and closely related to Quinennec 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, termed abstract continuations [21, 22, 42]. Standard continuation semantics is declared “inadequate” [21] and “insufficient” [22]. 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 abstract continuations [13]. 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

Context τω = τ → ω, (12)
Answer ω = ω. (13)

such that

Context τω = τ → Answer ω. (14)

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

\[
\text{lambda } (x) \\text{ (reset } (< 1 x))
\]

(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 Context Int Bool. In other words, the function

\[
\text{lambda } (x) \\text{ (reset } (% (y x)) (shift g (< 1 y))))
\]

(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 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 τ → ω. 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

\[
\text{List } \alpha = 1 + \alpha \times \text{List } \alpha,
\]

where 1 is the unit type and × 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

### 3.1 control

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

\[
\text{lambda } (x) \\text{ (let } ((y x)) (control g (< 1 y))))
\]

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

\[
\text{lambda } (x)
\]

(whose type is Int)
values against text, namely the empty one. If our target language lets us compare
Context
According to (15), the type
When two
but do not, for clarity.

The function send below plugs an intermediate answer v (of type α) into a delimited context mc of type Maybe(Context′ωω) 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.

\[
\text{prompt } E = (\lambda (c) c ((E \text{ send}) #f))
\]

\[
\text{control } f E = (\lambda (c1) (\lambda (mc1) (\text{let } ((f (\lambda (x) (\lambda (mc2) (((\text{compose } c1 mc1) x) (\text{compose } c2 mc2))))))) ((E \text{ send}) #f))))
\]

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 Answer′ωω). 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.

\[
\text{define-syntax prompt}
\]

\[
\text{define-syntax control}
\]

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 \(E\), 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 (\(τ \rightarrow \text{Answer′ωω}\)) → \(ω\). This monad is not the continuation monad, because the answer types Answer′ωω 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 Maybe(Context’ωωωω)), 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: \( (\text{reset } E) \) is not equivalent to \( (\text{reset } (\text{reset } E)) \), because each reset only “defends against” one shift0. For example, the program

\[
\text{(reset } (\text{cons } 'a \text{ (reset } (\text{shift0 } f (\text{shift0 } g '())))))\]

evaluates to (), but the program

\[
\text{(reset } (\text{cons } 'a \text{ (reset } (\text{reset } (\text{shift0 } f (\text{shift0 } g '())))))))\]

evaluates to (a).

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

\[
[(\text{let } ((y _)) \text{ (shift0 } f (\text{shift0 } g (< 1 y))))]\]

captured with shift0 is not equivalent to the contexts

\[
[(\text{let } ((y _)) \text{ (shift0 } g (< 1 y)))] \quad [(< 1 _ _)]
\]

captured with shift0. That is, the function

\[
(\lambda x \text{ (reset } (\text{let } ((y x)) \text{ (shift0 } f (\text{shift0 } g (< 1 y))))))\]

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

\[
(\lambda x \text{ (reset } (\text{let } ((y x)) (\text{shift0 } g (< 1 y)))))) \quad (\lambda x \text{ (reset } (< 1 x)))
\]
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. In our formulation, a delimited context captured by shift0 whose hole type is \( \tau \) and whose answer type is \( \omega \) has the type Context0 \( \tau \omega \omega \), where

\[
\text{Context0 } \tau \omega = \tau \rightarrow \text{List } (\text{Context0 } \omega \omega) \rightarrow \omega.
\]

In this recursive type definition, List \( \omega \) means a singly-linked list of \( \omega \)-values, either a cons cell or the empty list (). A list of type List (Context0 \( \omega \omega \)) contains delimited contexts from innermost to outermost, separated by control delimiters.

The function propagate below plugs an intermediate answer \( v \) (of type \( \omega \)) into a list of contexts \( lc \) (of type List (Context0 \( \omega \omega \))) by calling the head of \( lc \) with \( v \) and the tail of \( lc \). If \( c \) is empty, then the final answer is simply \( v \).

\[
\text{(define } (\text{propagate } v) (\lambda c \text{ (cons } c \text{ lc }))\]

\[
(\text{if } (\text{null? } lc) v \quad (((\text{car } lc) v) (\text{cdr } lc))))\]

This function is of type Context0 \( \omega \omega \); it is itself a delimited context, namely the empty one.

Like the type Context0 \( \tau \omega \omega \) in Section 3.1, Context0 \( \tau \omega \omega \) is a function type in which \( \tau \) only appears in the domain. Hence a delimited context captured by shift0 is just like one captured by shift, except the answer type Answer0 \( \omega \) of the continuation is recursive, defined by

\[
\text{Answer0 } \omega = \text{List } (\text{Context0 } \omega \omega) \rightarrow \omega
\]

\[
= \text{List } (\omega \rightarrow \text{Answer0 } \omega) \rightarrow \omega.
\]

such that

\[
\text{Context0 } \tau \omega = \tau \rightarrow \text{Answer0 } \omega
\]

\[
= \text{Context } \tau (\text{Answer0 } \omega).
\]

Thus Context0 can be written in terms of 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 \( \tau \), at the top level or to the right of a function arrow, to a type of the form \( \tau \rightarrow \text{Answer0 } \omega \rightarrow \text{Answer0 } \omega \). To distinguish the reset for shift0 here from the reset for shift above, we write reset0 instead of reset.

\[
(\text{reset0 } E) = (\lambda c \text{ (lambda } (lc) ((\text{E } \text{propagate }) (\text{cons } c \text{ lc}))))\]

\[12\]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.
reset delimited context within the nearest dynamically enclosing curried call with two arguments. To compose delimited contexts captured by control0 below, which combines recursive continuations and the CPS transform by and

The trivial delimited context of type \( \tau \omega \times \omega \) extends a standard one, it can be incorporated into CPS-based languages like Gasbichler and Sperber’s [28] give rise to direct implementations of control0 and prompt0 using 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 \( \tau \) and answer type \( \omega \), has the type

\[
\text{Context}_0' \tau \omega = \tau \rightarrow \text{Maybe}(\text{Context}_0' \omega \omega) \\
\text{List}(\text{Context}_0' \omega \omega) \rightarrow \omega.
\]

in which \( \tau \) only appears in the domain. A delimited context captured by control0 is thus just like one captured by shift with the recursive answer type

\[
\text{Answer}_0' \omega = \text{Maybe}(\text{Context}_0' \omega \omega) \\
\text{List}(\text{Context}_0' \omega \omega) \rightarrow \omega \\
= \text{Maybe}(\omega \rightarrow \text{Answer}_0' \omega) \\
\text{List}(\omega \rightarrow \text{Answer}_0' \omega) \rightarrow \omega.
\]

such that

\[
\text{Context}_0' \tau \omega = \tau \rightarrow \text{Answer}_0' \omega \\
= \text{Context} \tau (\text{Answer}_0' \omega).
\]

This CPS transform maps every source type \( \tau \), at the top level or to the right of a function arrow, to a type of the form \( \tau \rightarrow \text{Answer}_0' \omega \). 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 \( \lambda \)-calculus with products and sums [2], which computes normal forms for \( \lambda \)-terms.
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 encour-
agement of Oleg Kiselyov. Thanks also to Chris Barker, John Clemen-
ts, 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 Founda-
tion Grant BCS-0236592.

References

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.
Extensional normalisation and type-directed partial evalua-
tion for typed lambda calculus with sums. In POPL '04: Con-
ference record of the annual ACM symposium on principles
tended abstract). In [51], 1–11.
[4] Biernacka, Małgorzata, Dariusz Biernacki, and Olivier
Danvy. 2004. An operational foundation for delimited con-
tinuations. In [51], 25–33.
efficiency. In POPL '98: Conference record of the annual
ACM symposium on principles of programming languages,
recursive module? In PLDI '99: Proceedings of the ACM
conference on programming language design and implementa-
tion, vol. 34(5) of ACM SIGPLAN Notices, 50–63. New
York: ACM Press.
Conference record of the annual ACM symposium on principles
machine. Report RS-03-33, BRICS, Denmark.
rest of the computation. In [51].
straction of typed contexts. Tech. Rep. 89/12, DIKU, Univer-
sity of Copenhagen, Denmark. http://www.daimi.au.dk/~
danvy/Papers/fatc.ps.gz.
1990 ACM conference on Lisp and functional programming,
transformation. Mathematical Structures in Computer Sci-
style II: First-class continuations. In Proceedings of the 1992
ACM conference on Lisp and functional programming, ed.
York: ACM Press.
[15] ———. 1996. Back to direct style II: First-class continu-
tions. Report RS-96-20, BRICS, Denmark.
www.double.co.nz/scheme/partial-continuations/
partial-continuations.html.
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 $\lambda$-$\eta$-CS conver-
sion: A syntactic theory of control and state in imperative high-
order programming languages. Ph.D. thesis, Indiana Uni-
versity. Also as Tech. Rep. 226, Department of Computer Sci-
ence, Indiana University.
In [41], 180–190.
[21] Felleisen, Matthias, Daniel P. Friedman, Bruce F. Duba,
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
Conference record of the annual ACM symposium on principles
Computer Science, Carnegie Mellon University. Also as Tech.
Rep. CMU-CS-96-119.
Conference record of the annual ACM symposium on principles
[26] ———. 2001. Normalization by evaluation for the compu-
tational lambda-calculus. In TLCA 2001: Proceedings of the
5th international conference on typed lambda calculi and ap-
plications, ed. Samson Abramsky, 151–165. Lecture Notes in
Computer Science 2044, Berlin: Springer-Verlag.