Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

A static simulation of dynamic delimited control

Chung-chieh Shan

the date of receipt and acceptance should be inserted later

Abstract We present a continuation-passing-style (CPS) transformation for some dynamic delimited-control operators, including Felleisen's `control` and `prompt`, that extends a standard call-by-value CPS transformation. Based on this new transformation, we show how Danvy and Filinski's static delimited-control operators `shift` and `reset` simulate dynamic operators, allaying in passing some skepticism in the literature about the existence of such a simulation. The new CPS transformation and simulation use recursive delimited continuations to avoid undelimited control and the overhead it incurs in implementation and reasoning.

Keywords delimited control operators; macro expressibility; continuation-passing style (CPS); shift and reset; control and prompt

1 Introduction

Delimited continuations are widely useful: to name a few applications, in backtracking search [11, 23, 59, 76], direct-style representations of monads [37–39], the continuation-passing-style (CPS) transformation itself [22–24], partial evaluation [6, 7, 9, 19, 29, 40, 46, 63, 82], Web interactions [45, 70], mobile code [66, 74, 80], and linguistics [8, 75]. However, the proliferation of delimited-control operators [22–24, 31, 32, 35, 36, 47–49, 71, 76, 77] remains a source of confusion for users and work for implementers. We want to translate control operators to each other so that they may share implementations, programming examples, reasoning principles, and program transformations. In particular, we want to take advantage of CPS as we use, build, and reason about all delimited-control operators, not just `shift` and `reset` [22–24].

Informally speaking, this paper presents

1. a CPS transformation for delimited-control operators, including `control` and `prompt`, that extends a standard transformation; and
2. a simulation of these operators in terms of `shift` and `reset` that does not resort to undelimited control.

Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854, USA.

E-mail: ccshan@cs.rutgers.edu

The remainder of this section introduces these delimited-control operators: we first specify `shift` and `reset` in terms of CPS, then turn to an operational account using syntactic delimited contexts that accommodates the other operators more easily. Historically, the first delimited-control operators were `control` and `prompt` rather than `shift` and `reset`, and they were specified in terms of syntactic contexts rather than semantic continuations. But pedagogically, we begin with `shift` and `reset` and CPS to emphasize their connection, which is crucial to our results.

After this introduction, we make our claims precise in Section 2 and substantiate them with implementations in Section 3. The main innovation in the implementations is to use *recursive* delimited continuations. Section 4 proves our account of `control` and `prompt` correct with respect to previous accounts. Finally, Section 5 discusses related work and concludes.

1.1 Undelimited continuations

A *continuation* is a function that maps the intermediate result of a computation to its final result [79]. For example, the Scheme program

```
(not (> (/ 5 3) 4))
```

performs a computation that we can divide into three parts: first, compute $5/3$; second, check if it is greater than 4; finally, negate the result. The continuation of computing $5/3$, then, is to check if it is not greater than 4. In other words, the continuation of dividing 5 by 3 in the complete program above is the mathematical function

```
(lambda (v) (not (> v 4)))
```

in Scheme notation. In operational semantics, it is popular to represent a continuation by a syntactic *context*, or a complete program with a hole [31]. For example, this continuation is represented by the context `(not (> [ ] 4))`, where $[\,]$ is a hole to be filled.

We can program using continuations explicitly or implicitly. Explicitly, we can represent continuations by λ-abstractions and manage them as values at all times by coding in *continuation-passing style* (CPS). In CPS code, every non-primitive call occurs in a *tail* position (which is roughly a non-argument position) [78, page 58]. (A primitive call must terminate without side effects.) For example, the program

```
(define (/c x y c) (c (/ x y)))
(/c 5 3 (lambda (v) (not (> v 4))))
```

performs the same computation as described above, but by explicitly invoking the λ-abstraction `(lambda (v) (not (> v 4)))`, which represents a continuation. To take another example, the "safe division" function `/0c` defined by

```
(define (/0c x y c0 c) (if (zero? y) (c0 x) (c (/ x y))))
```

chooses between two explicit continuation representations, `c0` and `c`: if the divisor `y` is zero, it invokes `c0` with the dividend `x`; otherwise, it invokes `c` with the ratio `(/ x y)`. To stop the program when dividing by zero, we can provide the identity function `(lambda (v) v)` to `/0c` as the continuation `c0`. For example, the program

```
(/0c 5 3 (lambda (v) v) (lambda (v) (not (> v 4))))
```

yields `#t`, whereas the program

$$\overline{x} = \texttt{(lambda (c) (c } x\texttt{))}$$
$$\overline{\texttt{(lambda (}x\texttt{) }E\texttt{)}} = \texttt{(lambda (c) (c (lambda (}x\texttt{) }\overline{E}\texttt{)))}$$
$$\overline{\texttt{(}E_1\ E_2\texttt{)}} = \texttt{(lambda (c) (}\overline{E_1}\texttt{ (lambda (f) (}\overline{E_2}\texttt{ (lambda (x) ((f x) c))))))}$$

Fig. 1 A call-by-value CPS transformation for the pure λ-calculus

```
call/cc = (lambda (c) (c (lambda (f)
                           (lambda (c1)
                            ((f (lambda (v) (lambda (c2) (c1 v))))
                             c1)))))
```
(with `call/cc` overlined on the left-hand side)

Fig. 2 Extending the CPS transformation in Figure 1 with `call/cc`

```
(/0c 5 0 (lambda (v) v) (lambda (v) (not (> v 4))))
```

yields `5`. The calls to `/c`, `/0c`, `c`, and `c0` above occur in tail position, unlike the calls to `/` above.

Implicitly, we can manipulate continuations to regulate the control flow of a program. Scheme provides the *control operator* `call-with-current-continuation` (hereafter abbreviated `call/cc`) to access implicit continuations as first-class values [53]. For example, the "safe division" function `/0` defined by

```
(define (/0 x y c0) (if (zero? y) (c0 x) (/ x y)))
```

chooses between two continuations, `c0` and the current (implicit) continuation: if the divisor `y` is zero, it jumps to `c0` with the dividend `x`; otherwise, it returns normally with the ratio `(/ x y)`. For example, the program

```
(call/cc (lambda (c0) (not (> (/0 5 3 c0) 4))))
```

yields `#t`, whereas the program

```
(call/cc (lambda (c0) (not (> (/0 5 0 c0) 4))))
```

yields `5`.

Implicit continuations can be made explicit by a *CPS transformation* on programs [68]. Figure 1 shows the core of a standard call-by-value CPS transformation. It can be extended to deal with functions that take multiple arguments (such as `/`) and primitive operations (such as `not`). More importantly, as Figure 2 shows, it can be extended to translate programs that use `call/cc` to programs that do not. In the opposite direction, explicit continuations can be made implicit by a corresponding *direct-style* or *un-CPS* transformation [18, 25, 73].

1.2 Delimited continuations

A *delimited* (or *composable*, *functional*, or *partial*) continuation is a function that maps the intermediate result of a computation to a later, but not necessarily final, result. Take for example the program at the beginning of Section 1.1: until the point in its execution between comparing against 4 and negating the result, the delimited continuation of dividing 5 by 3 is to compare against 4. In other words, the continuation of the division, delimited before the negation, is the mathematical function

```
(lambda (v) (> v 4))
```

in Scheme notation. In operational semantics, it is popular to represent a delimited continuation by a syntactic *delimited context*, or an expression—not necessarily a complete program—with a hole. For example, we may represent the delimited continuation above by the context `(> [ ] 4)`. Whether a continuation function or a context data-structure is delimited is not a matter of its constitution but a matter of the role it plays in a program.

As with undelimited continuations, we can program using delimited continuations explicitly or implicitly. Explicitly, we can represent delimited continuations by λ-abstractions and manage them as values at all times by coding in continuation-*composing* style [22, 23], which (unlike CPS) does not require every call to occur in a tail position. For example, the programs

```
(/c 5 3 (lambda (v) (not (> v 4))))
(not (/c 5 3 (lambda (v) (> v 4))))
```

both yield `#t`, but the second program is not in CPS: it calls `/c` in a non-tail position, passing it a delimited continuation that does not contain the last part of the computation (negation). Nevertheless, delimited continuations are useful in practical programming. For example, the program

```
(define (eitherc x y c) (or (c x) (c y)))
(not (eitherc 5 3 (lambda (v) (> v 4))))
```

checks whether it is not the case that either 5 or 3 is greater than 4. The function `eitherc` invokes the delimited continuation `(lambda (v) (> v 4))` twice, in non-tail positions. The answer is `#f`. This example is a simple instance of the backtracking pattern in functional programming.

Implicitly, we can manipulate delimited continuations to regulate the delimited control flow of a program. Danvy and Filinski proposed the *delimited-control operators* `shift` and `reset` to access implicit delimited continuations as first-class values [22–24]. These operators extend a programming language such as the call-by-value λ-calculus with the following syntax.

$$\text{Expressions} \qquad E ::= \cdots \;\Big|\; \texttt{(shift f } E\texttt{)} \;\Big|\; \texttt{(reset } E\texttt{)} \tag{1}$$

Continuations are delimited by `reset` and captured by `shift`: as we make precise below, `shift` captures the current continuation delimited by the nearest dynamically-enclosing `reset`, binds `f` to the captured delimited continuation as a functional value, and replaces the current delimited continuation *abortively* by the identity function. For example, the program

```
(define (either x y) (shift c (or (c x) (c y))))
(not (reset (> (either 5 3) 4)))
```

also checks whether it is not the case that either 5 or 3 is greater than 4. Here `either` is the direct-style analogue to `eitherc` above. The `shift` on the first line captures the current continuation delimited by the `reset` on the second line, so `c` is bound to the mathematical function `(lambda (v) (> v 4))`.

Like undelimited continuations, implicit delimited continuations can be made explicit by a transformation on programs: as for `call/cc`, we can extend the CPS transformation in Figure 1 to translate programs that use `shift` and `reset` to programs that do not. Figure 3 shows the additional equations. They contain non-tail calls and so are sensitive to the evaluation order of the target language, which we take to be call-by-value. Therefore,

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

Fig. 3 Extending the CPS transformation in Figure 1 with `shift` and `reset`

strictly speaking, Figure 3 does not constitute a CPS transformation, only a continuation-composing-style transformation that extends a standard CPS transformation. If desired, we can eliminate these non-tail calls and regain CPS by applying the CPS transformation in Figure 1 to Figures 1 and 3 [23]. The continuations made explicit at this step are called *metacontinuations*. We effectively perform this further CPS transformation in Sections 3.3 and 3.4, where the equations' right-hand-sides are in CPS again, with tail calls only.

1.3 The original, operational account of delimited control

Before Danvy and Filinski introduced `shift` and `reset` to access delimited continuations, Felleisen introduced delimited-control operators to go beyond continuations [35]—to delimit and capture syntactic contexts [31, 32]. We can define `shift` and `reset` syntactically as well [9, 22, 24, 27, 67], with reduction rules in the style of Felleisen [31]:

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

$$M\big[\texttt{(reset } C\big[\underline{\texttt{(shift f } E\texttt{)}}\big]\texttt{)}\big] \rhd M\big[\texttt{(reset } E'\texttt{)}\big]$$
$$\text{where } E' = E\big\{\texttt{f} \mapsto \texttt{(lambda (}x\texttt{) (}\textbf{reset}\ C[x]\texttt{))}\big\} \text{ and } x \text{ is fresh.} \tag{3}$$

Underlining indicates redexes. Boldface distinguishes one `reset` on the right-hand side to be discussed in Sections 1.5 and 1.6 below. 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:

$$\text{Values} \qquad V ::= \texttt{(lambda (}x\texttt{) } E\texttt{)} \;\big|\; \cdots \tag{4}$$
$$\text{Contexts} \qquad C[\;] ::= [\;] \;\big|\; C[\texttt{(}[\;]\ E\texttt{)}] \;\big|\; C[\texttt{(}V\ [\;]\texttt{)}] \;\big|\; \cdots \tag{5}$$
$$\text{Metacontexts} \qquad M[\;] ::= C[\;] \;\big|\; M[\texttt{(reset } C[\;]\texttt{)}] \tag{6}$$

To help the exposition below, these reduction 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`, which is to provide the program with the identity function as the initial continuation.

It has been noted [9, 20, 26] that these syntactic definitions of contexts and metacontexts are not rabbits pulled out of hats. Rather, Table 1 shows how contexts are the result of defunctionalizing [72] the λ-abstractions in the right-hand side of Figure 1 that represent continuations. Similarly, metacontexts are the result of defunctionalizing the λ-abstractions introduced to represent metacontinuations in the CPS transformation of the right-hand side of Figure 3. Biernacka et al. [9] use this correspondence to provide an abstract machine for `shift` and `reset` and show that it is equivalent to the operational (reduction) and denotational (transformation) semantics above.

Contexts of the form:	represent continuation-representing λ-abstractions of the form:
$[\,]$	`(lambda (v) v)`
$C[$`([ ]` E`)`$]$	`(lambda (f) (`E `(lambda (x) ((f x)` C`))))`
$C[$`(`V `[ ])`$]$	`(lambda (x) ((`V `x)` C`))`

Table 1 Contexts are defunctionalized representations of continuations

1.4 A tale of two `reset`s

The reduction 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 transformation in Figure 3. Many other delimited-control operators have been proposed that remove one or both delimiters on the right-hand side of (3). Three such variations on `shift` and `reset` are possible, namely `control` and `prompt`:

$$M[\underline{\texttt{(prompt }V\texttt{)}}] \rhd M[V] \tag{7}$$

$$M[\texttt{(prompt }C[\underline{\texttt{(control f }E\texttt{)}}]\texttt{)}] \rhd M[\texttt{(prompt }E'\texttt{)}]$$
$$\text{where } E' = E\{\texttt{f} \mapsto \texttt{(lambda (}x\texttt{) }C[x]\texttt{)}\} \text{ and } x \text{ is fresh;} \tag{8}$$

`shift0` and `reset0`:

$$M[\underline{\texttt{(reset0 }V\texttt{)}}] \rhd M[V] \tag{9}$$

$$M[\texttt{(reset0 }C[\underline{\texttt{(shift0 f }E\texttt{)}}]\texttt{)}] \rhd M[E']$$
$$\text{where } E' = E\{\texttt{f} \mapsto \texttt{(lambda (}x\texttt{) (}\mathbf{reset0}\ C[x]\texttt{))}\} \text{ and } x \text{ is fresh;} \tag{10}$$

and `control0` and `prompt0`:

$$M[\underline{\texttt{(prompt0 }V\texttt{)}}] \rhd M[V] \tag{11}$$

$$M[\texttt{(prompt0 }C[\underline{\texttt{(control0 f }E\texttt{)}}]\texttt{)}] \rhd M[E']$$
$$\text{where } E' = E\{\texttt{f} \mapsto \texttt{(lambda (}x\texttt{) }C[x]\texttt{)}\} \text{ and } x \text{ is fresh.} \tag{12}$$

For each variation, we change the definition of metacontexts in (6) to replace `reset` by `prompt`, `reset0`, or `prompt0`. Each pair of these control operators can also be defined by an abstract machine.

1.5 Introducing `control` and `prompt`

Felleisen's `control` operator [31, 32, 35, 36, 77] captures a delimited context without surrounding it with a delimiter. Thus, when a context captured by `control` is invoked, it may further capture the context of invocation—unlike a context captured by `shift`. The difference between `shift` and `control` can be observed operationally as follows. The program

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

evaluates to `(a b)` by the following reduction steps:

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

Here `shift f` introduces a **`reset`** (in boldface) under the `lambda`, which stops `shift h` from capturing `cons 'b`. Thus the function

```
(lambda (x) (reset (let ((y x)) (shift h y))))
```

is equivalent to the identity function. In other words, the delimited context

```
(let ((y [])) (shift h y))
```

captured by `shift f` is equivalent to the empty delimited context [ ]. Furthermore, `shift f` maintains the `reset` delimiting the captured context, so `shift g` merely captures the empty context. Thus the initial redex

```
(shift f (shift g (cons 'b (f '()))))
```

is equivalent to just `(shift f (cons 'b (f '())))`.

In contrast, the program

```
(prompt (prompt (cons 'a (prompt
  (let ((y (control f (control g (cons 'b (f '()))))))
    (control h y))))))
```

evaluates to `(a)` by the following reduction steps:

```
▷ (prompt (prompt (cons 'a (prompt
    (control g (cons 'b
      ((lambda (x) (let ((y x)) (control h y))) '()))))))))
▷ (prompt (prompt (cons 'a (prompt
    (cons 'b
      ((lambda (x) (let ((y x)) (control h y))) '())))))))
▷ (prompt (prompt (cons 'a (prompt
    (cons 'b (let ((y '())) (control h y)))))))
▷ (prompt (prompt (cons 'a (prompt
    (cons 'b (control h '()))))))
▷ (prompt (prompt (cons 'a (prompt '()))))
```

Here `control f` allows `control h` to capture and drop `cons 'b`. Thus the function

```
(lambda (x) (let ((y x)) (control h y)))
```

is not equivalent to the identity function. In other words, the delimited context

```
(let ((y [ ])) (control h y))
```

captured by `control f` is not equivalent to the empty delimited context [ ]. Like `shift f` above, though, `control f` keeps the `prompt` delimiting the captured context, so `control g` merely captures the empty context. Thus the initial redex

```
(control f (control g (cons 'b (f '()))))
```

is equivalent to just `(control f (cons 'b (f '())))`.

The literature contains three abstract machines for `control` and `prompt`: Felleisen's original machine [10, 32], Biernacka et al.'s definitional machine [9, 12–15], and Biernacki et al.'s new machine in defunctionalized form [13].

Sitaram's `fcontrol` [76] is closely related to `control` in nature. Felleisen and Sitaram refer to the delimiter as `prompt`, `run`, `#`, or `%`.

1.6 Introducing `shift0` and `reset0`

The `shift0` operator captures a delimited context as `shift` does, but removes the `reset0` that delimits the captured context. For example, the program

```
(reset0 (reset0 (cons 'a (reset0
  (let ((y (shift0 f (shift0 g (cons 'b (f '()))))))
    (shift0 h y))))))
```

evaluates to `(b)` by the following reduction steps:

```
▷ (reset0 (reset0 (cons 'a
    (shift0 g (cons 'b
      ((lambda (x) (reset0 (let ((y x)) (shift0 h y)))) '())))))))
▷ (reset0
    (cons 'b
      ((lambda (x) (reset0 (let ((y x)) (shift0 h y)))) '())))
▷ (reset0 (cons 'b (reset0 (let ((y '())) (shift0 h y)))))
▷ (reset0 (cons 'b (reset0 (shift0 h '()))))
▷ (reset0 (cons 'b '()))
```

Like `shift f` above, `shift0 f` here introduces a **`reset0`** (in boldface) under the `lambda`, which stops `shift0 h` from capturing `cons 'b`. Thus the function

```
(lambda (x) (reset0 (let ((y x)) (shift0 h y))))
```

is equivalent to the identity function. In other words, the delimited context

```
(let ((y [ ])) (shift0 h y))
```

captured by `shift0 f` is equivalent to the empty delimited context [ ]. Unlike `shift f` above, however, `shift0 f` removes the captured context along with its delimiting `reset0`, exposing the next-outer delimited context `cons 'a` to be captured by `shift0 g`. Thus the initial redex

```
(shift0 f (shift0 g (cons 'b (f '()))))
```

is not equivalent to just `(shift0 f (cons 'b (f '())))`.

With `shift0` in the language, `reset0` is not idempotent: `(reset0` E`)` is not equivalent to `(reset0 (reset0` E`))`, because each `reset0` only "defends against" one `shift0`. For example, the program

```
(reset0 (cons 'a
          (reset0 (shift0 f (shift0 g '())))))
```

evaluates to `()`, but the program

```
(reset0 (cons 'a
          (reset0 (reset0 (shift0 f (shift0 g '()))))))
```

evaluates to `(a)`.

Danvy and Filinski [22] consider the `shift0` operator briefly in their work on `shift`. Also, Hieb and Dybvig's `spawn` [49] can be thought of as a `reset0` that, each time it is invoked to insert a new delimiter, creates a specific `shift0` operator for that new delimiter.

1.7 Introducing `control0` and `prompt0`

The `control0` operator is like `control` but removes the `prompt0` that delimits the captured context. For example, the program

```
(prompt0 (prompt0 (cons 'a (prompt0
  (let ((y (control0 f (control0 g (cons 'b (f '()))))))
    (control0 h y))))))
```

evaluates to `()` by the following reduction steps:

```
▷ (prompt0 (prompt0 (cons 'a
    (control0 g (cons 'b
      ((lambda (x) (let ((y x)) (control0 h y))) '())))))))
▷ (prompt0
    (cons 'b
      ((lambda (x) (let ((y x)) (control0 h y))) '())))
▷ (prompt0 (cons 'b (let ((y '())) (control0 h y))))
▷ (prompt0 (cons 'b (control0 h '())))
▷ '()
```

Here `control0 f` allows `control0 g` to capture `cons 'a`, and `control0 h` to capture `cons 'b`. Thus the initial redex

```
(control0 f (control0 g (cons 'b (f '()))))
```

is not equivalent to just `(control0 f (cons 'b (f '())))`. Neither is the delimited context

```
(let ((y [])) (control0 h y))
```

captured by `control0 f` equivalent to the empty delimited context [ ].

The `control0` and `prompt0` operators are essentially Gunter et al.'s `cupto` and `set` [47, 48] stripped down to one delimiter label, and closely related to Queinnec and Serpette's `splitter` [71]. Dybvig et al. [30] introduce a set of control operators that are like `cupto` and `set`, except unlike Gunter et al. (and us) they represent a continuation in an abstract data type (not necessarily as a function) and invoke it with a general expression (not necessarily a value). Dybvig et al. provide an abstract machine for their operators.

1.8 Static versus dynamic operators

Danvy and Filinski [22–24] informally classify their `shift` and `reset` operators as *lexical* and *static*, and other delimited-control operators such as `control` and `prompt` as *dynamic*. They use these words to draw an analogy to lexical versus dynamic scope for variables: Lexical scope segregates the environment of an abstraction from the environment of its application, whereas dynamic scope allows the body of an abstraction to access the environment of application. Similarly, as Biernacki et al. [15, Section 2.2] explain, the context captured by `shift` is segregated from the context of its application, whereas dynamic control operators allow a captured context to access its context of application.

2 Expressing control operators by extending the CPS transformation

Described operationally as in (7)–(12), the variations among static and dynamic delimited-control operators seem like minor changes with little sense of purpose. Because it is easy to add a delimiter, it is easy to express `shift` and `reset` in terms of dynamic operators such as `control` and `prompt` [12], as well as to express all these operators in terms of `control0` and `prompt0`. In the opposite direction, it "seems not to be known" [47, 48] how to remove a delimiter, for example whether `shift` and `reset` can express `control` and `prompt`. Without appeal to CPS, each version of these operators serves equally well the basic purpose of letting the programmer regulate the control flow of a delimited part of a program, so Gunter et al. choose to take `control0` and `prompt0` as primitive.

To be precise, we turn to Felleisen's notion of *macro-expressibility* [33]: a language $\mathscr{L}'$ can macro-express its conservative extension $\mathscr{L}$ if and only if each facility $\mathbb{F}$ present in $\mathscr{L}$ but not $\mathscr{L}'$ can be translated by a *syntactic abstraction* A into $\mathscr{L}'$, such that a program in $\mathscr{L}$ halts exactly when its translation in $\mathscr{L}'$ halts. A syntactic abstraction is a syntactic context, or (in Scheme) a macro that does not analyze its arguments. We weaken the notion of macro-expressibility to *top-macro-expressibility*: $\mathscr{L}'$ can top-macro-express $\mathscr{L}$ if and only if each facility $\mathbb{F}$ in $\mathscr{L}$ but not $\mathscr{L}'$ can be translated by a syntactic abstraction A into $\mathscr{L}'$, *and* there exists a unary syntactic abstraction A_0, such that a program in $\mathscr{L}$ halts exactly when A_0 applied to its translation in $\mathscr{L}'$ halts. That is, top-macro-expressibility allows the translation to specify a syntactic abstraction A_0 for the "top level" of programs of $\mathscr{L}$. For short, when the λ-calculus with one set of control operators top-macro-expresses the λ-calculus with another set of control operators, we say that the former set *simulates* the latter set. This relation is obviously a preorder: below we use the fact that we can compose simulations.

We show below that `shift` and `reset` simulate `control` and `prompt`, `shift0` and `reset0`, and `control0` and `prompt0`. On the way, we provide the latter calculi with CPS transformations that extend a standard one.

2.1 Expressing control operators

Merely that one set of control operators simulates another is not surprising. In particular, using mutable state (`set!`) and undelimited control (`call/cc`):

1. Sitaram and Felleisen [77] simulate `control` and `prompt`;
2. Filinski [37] simulates `shift` and `reset`;[1] and

[1] This simulation uses a single mutable storage cell containing an undelimited continuation. As such, it underlies the connection that Ariola et al. [5] made between delimited control and subtractive logic.

3. Gunter et al. [47, 48] simulate `cupto` and `set`, which subsume `control0` and `prompt0`.

Moreover, we can simulate `set!` and `call/cc` using `shift` and `reset`, in two steps: First, we can simulate an unbounded number of storage cells using a single storage cell by implementing our own memory allocation inside a vector or list. Second, because a single storage cell and undelimited control together form a monadic effect (namely the continuation monad transformer applied to the state monad), we can in turn simulate them using `shift` and `reset` [37–39]. Thus we already know that `shift` and `reset` simulate the other delimited-control operators. The CPS transformations for `shift` and `reset` [22–24] then pull back to the other operators.

What may be surprising is that we need not resort to undelimited control in order to simulate the dynamic operators. We use the word "resort" because translation through `call/cc` is undesirable: `call/cc` captures the context even beyond the delimiter.

1. In practical terms, this context junk is duplicated and discarded along with the delimited continuation itself, which is inefficient. For example, Queinnec [70] built a Web application using `call/cc` in which each serialized continuation contains 100K of overhead. Recently, Gasbichler and Sperber [44] showed that it is much more efficient to implement `shift` and `reset` without capturing undelimited continuations. (Gasbichler and Sperber actually implement `shift` and `reset` by way of `control` and `prompt`, but not `shift0` and `reset0` or `control0` and `prompt0`.)
2. In theoretical terms, this context junk not only hampers the efficiency of an implementation but also complicates reasoning about programs and their observational equivalence [41].

The program below demonstrates the space overhead that can be incurred when translating delimited-control operators through undelimited ones.

```
(let loop ((junk-identity
            (let ((junk (list 'junk)))
              (cons junk (reset (shift f f))))))
  (set-cdr! (car junk-identity)
            (list (cdr junk-identity)))
  (loop (cons (cdr (car junk-identity))
              (cdr junk-identity))))
```

The expression `(shift f f)` above captures the empty context, that is, the identity function as a delimited continuation. However, its undelimited continuation contains `junk`, to which the rest of `loop` appends. It does not matter if we replace `(reset (shift f f))` above by `(prompt (control f f))`, `(reset0 (shift0 f f))`, or `(prompt0 (control0 f f))`: whereas a native implementation of delimited control enters an infinite loop with bounded memory usage, an implementation of delimited control in terms of undelimited control, such as the three previous ones named above, fails to garbage-collect `junk` and thus expends an arbitrary amount of memory. In contrast, our simulation of `control` and `prompt` uses bounded space, according to a simple definition of space consumption [16] for Biernacka et al.'s abstract machine for `shift` and `reset` [9].

2.2 Extending the CPS transformation

Concomitant with the difficulty of using `shift` and `reset` to simulate the other control operators is the difficulty of devising denotational semantics for these operators that extend a

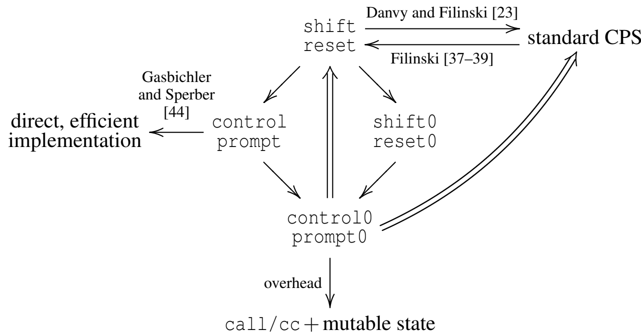

Fig. 4 Control operators and translations among them

standard CPS transformation. To be more precise, unlike with `shift` and `reset`, it is unclear how to translate the other operators using a transformation that coincides on pure λ-terms with Figure 1. Instead, semantics in the literature for dynamic delimited-control operators either rely on complex mutable data structures containing undelimited continuations (in essence implementing the operators in Scheme) or specify an algebra of contexts termed *abstract continuations* [35, 36, 69] (in essence operating on sequences of activation frames).

The use of these alternative reasoning tools for dynamic delimited control led to the natural question whether standard continuation semantics can be used at all. Ordinary continuation semantics was declared “inadequate” [35], “insufficient”, and a “failure” [36], as it was said that `control` expressions “in general have no CPS counterpart” [24]. Since these claims were made, however, monads have become more popular in the denotational semantics of computational effects [65]. Monads are so general that we can hope for them to cover even a language with `control0` and `prompt0`. Furthermore, Filinski showed how to represent monads using continuations and in terms of `shift` and `reset` [37–39], so we can further hope that ordinary continuations and `shift` and `reset` suffice to treat dynamic delimited-control operators. We show here that indeed they do. What distinguishes dynamic control operators is that the continuation is *recursive*.

Figure 4 depicts the situation: the single arrows represent previous translation results, and the two double arrows represent our contributions here.

3 Extending the CPS transformation using recursive continuations

In this central section of the paper, we translate dynamic control operators by extending the standard CPS transformation, then simulate them using `shift` and `reset`. We visit the operators in the order in which we introduced them in Section 1: first the static operators `shift` and `reset`, then the dynamic operators `control` and `prompt`, `shift0` and `reset0`, and finally `control0` and `prompt0`.

The key to our treatment is to represent delimited contexts as recursive functions: When a delimited context captured by a dynamic control operator is invoked, it may take control over the context of invocation. Hence, the captured context must take the invocation context as an argument in our CPS transformation, then apply the invocation context (except when discarding the invocation context abortively). Because the captured context and the invocation context may be the same, it is recursive for the former to apply the latter. We can think of the invocation context as an accumulator argument to the captured context [84].

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

$$\mathsf{List}\ \alpha \ = \ 1 + \alpha \times \mathsf{List}\ \alpha, \tag{13}$$

where 1 is the unit type (inhabited by the empty list `()`) and $\times$ constructs product types (inhabited by `cons` cells). For brevity, we take the unfolding of a recursive type to give not just isomorphic but in fact equivalent types [17, 43]. For example, (13) states an equation between types, not just an isomorphism; we do not explicitly convert a pair to a list.

3.1 Context types and answer types for `shift` and `reset`

We first review the types of delimited contexts captured by `shift` and `reset`, then turn to the types of delimited contexts captured by other operators. The CPS transformation 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 transformation 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 mapped to a type of the form $(\tau \to \omega_1) \to \omega_2$, where ω_1 and ω_2 are *answer types* [22, 64, 67, 83]. The delimited continuation is a function from the type τ, of the hole in the delimited context, to the type ω_1, of the context once plugged. 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 produces an integer as the final answer when plugged into a delimited context of type $\mathsf{Sym} \to \mathsf{Bool}$. One such delimited context is `(eq? [ ] 'b)` captured by `shift`: it produces a boolean when plugged with a symbol. In other words, the function `(lambda (x) (reset (eq? x 'b)))` maps symbols to booleans.

For comparison with other control operators below, we define the types

$$\mathsf{Context}\ \tau\ \omega \ = \ \tau \to \omega, \tag{14}$$

$$\mathsf{Answer}\ \omega \ = \ \omega, \tag{15}$$

such that

$$\mathsf{Context}\ \tau\ \omega \ = \ \tau \to \mathsf{Answer}\ \omega. \tag{16}$$

3.2 Translating `control` and `prompt`

We now treat the dynamic operators `control` and `prompt`.

3.2.1 Concatenating contexts by combining recursive continuations

In general, when a context captured by `control` is invoked, it may further capture the surrounding context (up to the nearest dynamically enclosing `reset`) at the point of invocation. Thus a context captured by `control`, unlike one captured by `shift`, does not denote a map from an intermediate result (that fills a hole) to a final answer, but rather denotes a map from an intermediate result *and any surrounding context at the point of invocation* to a final answer. In other words, because a `control`-captured context may take control over a context where it is subsequently invoked, we need to pass it the invocation context as a semantic argument. There are two cases: either the invocation context is the empty context [ ] (if the captured context is invoked right under a delimiter) or it is not empty. Accordingly, we let a context captured by `control` whose hole is of type τ and answer is of type ω take the type $\mathsf{Context}'\ \tau\ \omega$ defined by

$$\mathsf{Context}'\ \tau\ \omega \;=\; \tau \to (1 + \mathsf{Context}'\ \omega\ \omega) \to \omega. \tag{17}$$

In this recursive type definition, $1+\alpha$ means either the special token `#f` or a value of type α. We use `#f` (of type 1) to represent the empty invocation context.

The empty context captured by `shift` is the identity function of type $\mathsf{Context}\ \omega\ \omega$. Analogously, the empty context captured by `control` has the type $\mathsf{Context}'\ \omega\ \omega$, but it cannot just be the identity function because there is an additional argument in (17) of type $1+\mathsf{Context}'\ \omega\ \omega$. Rather, it is the recursive function `send` defined below.

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

This function implements the empty context as follows. Plugging an intermediate result `v` (of type ω) into the empty context in an invocation context `mc` (of type $1+\mathsf{Context}'\ \omega\ \omega$) yields the final answer of `mc`[`v`] (that is, the final answer of filling the hole in `mc` with `v`). If `mc` is empty (that is, `#f`), then the final answer is just `v`. If `mc` is not empty, then the final answer is that of plugging `v` into `mc` in the empty invocation context `#f`. This code thus relies on the fact that contexts and their concatenation form a monoid. In particular, the empty context is a left and right identity for concatenation.

To concatenate two contexts captured by `shift` is to compose two continuation functions. To concatenate two contexts captured by `control`, one of type $\mathsf{Context}'\ \tau\ \omega$ inside another of type $1+\mathsf{Context}'\ \omega\ \omega$, we define a recursive function `compose`, of type $\big(\mathsf{Context}'\ \tau\ \omega \times (1+\mathsf{Context}'\ \omega\ \omega)\big) \to \mathsf{Context}'\ \tau\ \omega$ (among other types):

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

This function concatenates two contexts `c` and `mc1` as follows. Plugging an intermediate result `v` into `mc1`[`c`[ ]] in an invocation context `mc2` yields the final answer of `mc2`[`mc1`[`c`[`v`]]], which by the associativity of concatenation is the same as plugging `v` into `c` in the invocation context `mc2`[`mc1`[ ]]. If `mc1` is empty (that is, `#f`), then the concatenation `mc1`[`c`[ ]] is just `c`[ ].

The special token `#f`, of type $1+\mathsf{Context}'\ \omega\ \omega$ above, distinguishes the empty invocation context from other invocation contexts, so as to ground the recursion in `send` and `compose`. Both `#f` and `send` represent the empty context: If our target language lets us compare values against `send` intensionally, then we can eliminate `#f` and use `send` as the special token. That is, we could implement `send` in Scheme as

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

and `compose` as

```
(define (compose c mc1)
  (if (eq? send mc1)
      c
      (lambda (v)
        (lambda (mc2)
          ((c v) (compose mc1 mc2))))))
```

but do not, for clarity. As a reviewer points out, another way to avoid the sum type and the special token `#f` is the Church encoding.

3.2.2 Extending the standard CPS transformation with `control` and `prompt`

According to (17), the type $\mathsf{Context}'\ \tau\ \omega$ is a function type, and τ only appears in its domain, not codomain. Thus a context captured by `control` has the function type of a continuation, just like a context captured by `shift`, except for the recursive answer type $\mathsf{Answer}'\ \omega$ defined by

$$\mathsf{Answer}'\ \omega\ =\ (1+\mathsf{Context}'\ \omega\ \omega)\to\omega\ =\ (1+\omega\to\mathsf{Answer}'\ \omega)\to\omega, \tag{18}$$

such that

$$\mathsf{Context}'\ \tau\ \omega\ =\ \tau\to\mathsf{Answer}'\ \omega\ =\ \mathsf{Context}\ \tau\ (\mathsf{Answer}'\ \omega). \tag{19}$$

Thus we can write $\mathsf{Context}'$ in terms of $\mathsf{Context}$! That is, we can treat a delimited context captured by `control` as an ordinary, if recursive, continuation. The equations below extend Figure 1 to `control` and `prompt`. It maps every source type τ, at the top level or to the right of a function arrow, to a type of the form $(\tau\to\mathsf{Answer}'\ \omega)\to\mathsf{Answer}'\ \omega$.

$$\overline{\texttt{(prompt }E\texttt{)}} = \texttt{(lambda (c) (c ((}\overline{E}\texttt{ send) \#f)))} \tag{20}$$

$$\overline{\texttt{(control f }E\texttt{)}} = \tag{21}$$

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

The final result of a complete program E, surrounded by a `prompt`, is `((`$\overline{E}$` send) #f)` (of type ω). This explains the translation for `prompt` in (20): it plugs the result of E into the current continuation `c`. To understand the translation for `control` in (21), suppose we plug `(control f `E`)` into a context `c1` inside an invocation context `mc1`. Then our program is of the form

$$M[\texttt{(prompt mc1}[\texttt{c1}[\texttt{(control f }E\texttt{)}]]\texttt{)}], \tag{22}$$

where $M[\ ]$ is a metacontext. This program reduces to

$$M[\texttt{(prompt }E\{\texttt{f}\mapsto\cdots\}\texttt{)}]. \tag{23}$$

When we later plug `(f x)`, where `x` is some value, into a context `c2` inside an invocation context `mc2`, our program is of the form

$$M[\texttt{(prompt mc2}[\texttt{c2}[\texttt{(f } V\texttt{)}]]\texttt{)}], \tag{24}$$

and should reduce to

$$M[\texttt{(prompt mc2}[\texttt{c2}[\texttt{mc1}[\texttt{c1}[\texttt{x}]]]]\texttt{)}]. \tag{25}$$

The body of `f` in (21) is `(((compose c1 mc1) x) (compose c2 mc2))`, which is the result of `(prompt mc2[c2[mc1[c1[x]]]])`.

3.2.3 Simulating `control` and `prompt` in terms of `shift` and `reset`

Because this transformation extends a standard CPS transformation, 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` express 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 ()
    ((prompt e)
     (shift c (c ((reset (send e)) #f))))))

(define-syntax control
  (syntax-rules ()
    ((control 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)))))))
```

To obtain the final result of a complete program (surrounded by a `prompt`), we enclose the program in `prompt-top-level`.

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

The definition of `prompt` above happens to simplify to this macro.

These macros correspond directly to the CPS equations in the previous paragraph, except:

1. Wherever the CPS equations abstract over a continuation argument, the macros use `shift` rather than `lambda`.
2. Wherever the equations pass the continuation C to $\overline{E}$, the macros say `(reset (`$C\ E$`))`, to plug E into the delimited context `(`C `[ ])`. The only C above is `send`.

This simulation of `control` and `prompt` in terms of `shift` and `reset` does not use `call/cc` directly: it does not capture any continuation beyond the outermost delimiting `prompt`. It also does not keep state, though `mc` above (of type $1+\mathsf{Context}'\ \omega\ \omega$ in (18)) is effectively threaded along as a single storage cell.

3.3 Translating shift0 and reset0

Appendix C of Danvy and Filinski's technical report [22] considers `shift0` and `reset0` briefly. It shows a denotational model that passes around a list of delimited contexts, which can be thought of as a sequence of activation frames, except each frame corresponds to a `reset0` rather than a function call.[2] 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

$$\mathsf{Context}_0\ \tau\ \omega \ = \ \tau \rightarrow \mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega) \rightarrow \omega. \tag{26}$$

A value of type $\mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega)$ lists successive delimited contexts from innermost to outermost.

The function `propagate` defined 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 `lc` 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 3.2, $\mathsf{Context}_0\ \tau\ \omega$ is a function type in which τ appears only 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

$$\mathsf{Answer}_0\ \omega \ = \ \mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega) \rightarrow \omega \ = \ \mathsf{List}\,(\omega \rightarrow \mathsf{Answer}_0\ \omega) \rightarrow \omega, \tag{27}$$

such that

$$\mathsf{Context}_0\ \tau\ \omega \ = \ \tau \rightarrow \mathsf{Answer}_0\ \omega \ = \ \mathsf{Context}\ \tau\ (\mathsf{Answer}_0\ \omega). \tag{28}$$

Thus we can write $\mathsf{Context}_0$ in terms of $\mathsf{Context}$. Therefore, just as with `control`, we can treat a delimited context captured by `shift0` as an ordinary continuation. Following the Appendix C mentioned above, the equations below extend Figure 1 to a CPS transformation 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 $(\tau \rightarrow \mathsf{Answer}_0\ \omega) \rightarrow \mathsf{Answer}_0\ \omega$.

```
‾(reset0 E)‾ = (lambda (c)                                  (29)
                 (lambda (lc)
                  ((Ē propagate) (cons c lc))))
‾(shift0 f E)‾ = (lambda (c1)                               (30)
                 (lambda (lc)
                  (let ((f (lambda (x)
                             (lambda (c2)
                              (lambda (lc)
                               ((c1 x) (cons c2 lc)))))))
                    ((Ē (car lc)) (cdr lc)))))
```

[2] Johnson and Duggan [52] add control facilities to the programming language GL that are similar in power to `shift0` and `reset0`, but they make each function call delimit the context (like in Landin's SECD machine [21, 62]), so their frames do correspond to function calls.

As promised at the end of Section 1.2, these equations are in CPS: the apparently non-tail call in expressions like `((`$\overline{E}$` propagate) (cons c lc))` begins a curried call with two arguments. The final result of a complete program E is `((`$\overline{E}$` propagate) '())`.

As in Section 3.2.3, we turn these equations into a simulation of `shift0` and `reset0` in terms of `shift` and `reset` that neither captures undelimited continuations nor keeps mutable state.

```
(define-syntax reset0
  (syntax-rules ()
    ((reset0 e)
     (shift c
      (lambda (lc)
       ((reset (propagate e)) (cons c lc)))))))

(define-syntax shift0
  (syntax-rules ()
    ((shift0 f e)
     (shift c1
      (lambda (lc)
       (let ((f (lambda (x)
                  (shift c2
                   (lambda (lc)
                    ((c1 x) (cons c2 lc)))))))
         ((reset ((car lc) e)) (cdr lc)))))))))

(define-syntax reset0-top-level
  (syntax-rules ()
    ((reset0-top-level e)
     ((reset (propagate e)) '()))))
```

The last macro `reset0-top-level` encloses a complete program to yield its final result.

3.4 Translating `control0` and `prompt0`

The `control0` operator removes both occurrences of `prompt0` on the right-hand side of (12); it combines the dynamic nature of `control` and `shift0`. It is thus not surprising that we can treat `control0` with recursive continuations and the CPS transformation by combining the ideas from Sections 3.2 and 3.3.

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

$$\mathsf{Context}_0'\ \tau\ \omega \ = \ \tau \to (1 + \mathsf{Context}_0'\ \omega\ \omega) \to \mathsf{List}\,(\mathsf{Context}_0'\ \omega\ \omega) \to \omega, \tag{31}$$

in which τ appears only in the domain. A delimited context captured by `control0` is thus just like one captured by `shift` with the recursive answer type $\mathsf{Answer}_0'\ \omega$ defined by

$$\begin{aligned}\mathsf{Answer}_0'\ \omega \ &= \ (1 + \mathsf{Context}_0'\ \omega\ \omega) \to \mathsf{List}\,(\mathsf{Context}_0'\ \omega\ \omega) \to \omega \\ &= \ \big(1 + (\omega \to \mathsf{Answer}_0'\ \omega)\big) \to \mathsf{List}\,(\omega \to \mathsf{Answer}_0'\ \omega) \to \omega,\end{aligned} \tag{32}$$

such that

$$\mathsf{Context}_0'\ \tau\ \omega \ = \ \tau \to \mathsf{Answer}_0'\ \omega \ = \ \mathsf{Context}\ \tau\ (\mathsf{Answer}_0'\ \omega). \tag{33}$$

Thus we can write $\mathsf{Context}'_0$ in terms of $\mathsf{Context}$. Informally speaking, the $1+$ part of the types above keeps track of the delimited context within the nearest dynamically enclosing `prompt0`, and the List part keeps track of the delimited contexts beyond that `prompt0`.

The empty delimited context captured by `control0`, 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 simply use the code for `compose` above, because—although it is created for `control`—it also has the type

$$\big(\mathsf{Context}'_0\ \tau\ \omega \times (1+\mathsf{Context}'_0\ \omega\ \omega)\big) \to \mathsf{Context}'_0\ \tau\ \omega. \tag{34}$$

Finally, we use `send-propagate` and `compose` to define an ordinary CPS transformation for `control0`. It maps every source type τ, at the top level or to the right of a function arrow, to a type of the form $(\tau \to \mathsf{Answer}'_0\ \omega) \to \mathsf{Answer}'_0\ \omega$.

$$\overline{\texttt{(prompt0 }E\texttt{)}} = \texttt{(lambda (c) (lambda (mc) (lambda (lc) ((}(\overline{E}\texttt{ send-propagate) #f) (cons (compose c mc) lc)))))} \tag{35}$$

```
(prompt0 E) = (lambda (c)
                (lambda (mc)
                 (lambda (lc)
                  (((E send-propagate) #f)
                   (cons (compose c mc) lc)))))
```

$$\overline{\texttt{(control0 f }E\texttt{)}} = \ldots \tag{36}$$

```
(control0 f E) = (lambda (c1)
                   (lambda (mc1)
                    (lambda (lc)
                     (let ((f (lambda (x)
                                (lambda (c2)
                                 (lambda (mc2)
                                  (((compose c1 mc1) x)
                                   (compose c2 mc2)))))))
                       (((E (car lc)) #f) (cdr lc))))))
```

(In both equations above, the left-hand side is overlined and E on the right-hand side is printed as $\overline{E}$.)

The final result of a complete program E is `(((`$\overline{E}$` propagate) #f) '())`.

Again, we turn these CPS equations into a simulation of `control0` and `prompt0` in terms of `shift` and `reset` that neither captures undelimited continuations nor keeps mutable state.

```
(define-syntax prompt0
  (syntax-rules ()
    ((prompt0 e)
     (shift c
      (lambda (mc)
       (lambda (lc)
        (((reset (send-propagate e)) #f)
         (cons (compose c mc) lc))))))))
```

```
(define-syntax control0
  (syntax-rules ()
    ((control0 f e)
     (shift c1
      (lambda (mc1)
       (lambda (lc)
        (let ((f (lambda (x)
                   (shift c2
                    (lambda (mc2)
                     (((compose c1 mc1) x)
                      (compose c2 mc2)))))))
          (((reset ((car lc) e)) #f)
           (cdr lc))))))))))

(define-syntax prompt0-top-level
  (syntax-rules ()
    ((prompt0-top-level e)
     (((reset (send-propagate e)) #f) '()))))
```

The last macro `prompt0-top-level` encloses a complete program to yield its final result.

4 Relation to other accounts of `control` and `prompt`

Section 3.2.2 above gives a CPS transformation for `control` and `prompt`. Another way to view the same definitions in hindsight is to recognize that a denotational semantics given by Felleisen et al. [35, Section 4] encodes `control` and `prompt` in the monad Control given by

$$\mathsf{Control}\ \tau = \mathsf{Context}'\ \tau\ \omega \to \omega, \tag{37}$$

$$\texttt{(unit x)} = \texttt{(send x)}, \tag{38}$$

$$\texttt{(bind m c)} = \texttt{(lambda (mc) (m (compose c mc)))}. \tag{39}$$

(In Felleisen et al.'s notation, `send` is $\hat{\mathbf{I}}$ and `compose` is $\lambda gf.\, \mathbf{B}_{val} f g$.) Because the answer types $\mathsf{Answer}'\ \omega$ and ω are different, this monad is not the continuation monad, so these denotational equations do not give a standard CPS transformation. (In contrast, Wadler [83] shows how Murthy's types for `shift` and `reset` [67] unify the two answer types and thus are compatible with the continuation monad.) Yet we can use Filinski's representation of monads in terms of `shift` and `reset` [37–39] to represent `control` and `prompt`—essentially as in (20) and (21). In fact, it recently came to light that Filinski, in personal communication to Danvy in 1994, had already represented the Control monad and thus implemented `control` and `prompt` in terms of `shift` and `reset`, though instead of the `send` and `compose` we define in Section 3.2.1, Filinski used Felleisen et al.'s initial-algebra representation of contexts [36]. As a reviewer points out, the monadic view is one way to prove that our definitions implement `control` and `prompt`.

A more direct proof that our translation is faithful lies in Ager et al.'s functional correspondence between evaluators and abstract machines [2–4] using defunctionalization [26, 72] and the CPS transformation. In one direction, this correspondence turns an evaluator into an abstract machine by transforming the evaluator into CPS and defunctionalizing continuations (represented by λ-abstractions) into contexts. For example, Ager et al. [2] turned a call-by-value evaluator into the CEK machine [34] and a call-by-name evaluator into the

Krivine machine [61]. In the other direction, the correspondence turns an abstract machine into an evaluator by refunctionalizing contexts into continuations and transforming the result into direct style. For example, Danvy [21] revealed the evaluator underlying Landin's SECD machine [62]; more recently, Biernacki et al. [13] derived another CPS transformation for `control` and `prompt` from their new abstract machine.

In our case, we follow the latter direction to turn our equations into an abstract machine, as detailed in the appendix: First, we convert the right-hand-sides of our equations in Figure 1 and (20) and (21) into an evaluator that passes environments and closures explicitly. Second, we CPS-transform the evaluator to eliminate the non-tail calls in (20) and (21). Finally, we defunctionalize continuations and metacontinuations (both represented by λ-abstractions) to get an abstract machine corresponding to the evaluator [9, 20, 26]. Indeed, Biernacki et al. have already constructed this machine [13, Section 12]. The machine extends the data type of contexts in (5) with a new constructor corresponding to the `compose` function in Section 3.2.1. A straightforward simulation argument between this machine and Biernacka et al.'s definitional machine shows that our machine correctly implements `control` and `prompt`, so our CPS transformation does as well.

Sitaram and Felleisen [77] simulate `control` and `prompt` in Scheme using `call/cc` and multiple mutable storage cells. Our simulation of `control` and `prompt` using `shift` and `reset` composes with Filinski's simulation of `shift` and `reset` using `call/cc` and a single mutable storage cell [37] to yield a more modular simulation of `control` and `prompt` using `call/cc` and state. Sitaram and Felleisen's simulation maintains a global, mutable *run-stack*, 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 correlate with our simulation: The run-stack is a sequence of "`mc`" functions (of type $1 + \mathsf{Context}'\ \omega\ \omega$), one for each dynamically active `prompt`. Each `mc` function is a sub-stack, the result of concatenating `control`-captured contexts using `compose`.

5 Conclusion and related work

We have presented the first CPS transformation for dynamic delimited-control operators, including `control` and `prompt`, that extends a standard CPS transformation. Based on this new CPS transformation, we have shown how `shift` and `reset` simulate dynamic operators and so are just as expressive.

Now that we know that `shift` and `reset` simulate dynamic operators, an implementation of `shift` and `reset` that does not capture undelimited continuations, like Gasbichler and Sperber's [44], gives rise to an implementation of dynamic operators that does not capture undelimited continuations. Given how many delimited-control operators have been (and will be?) proposed, simulations like the ones presented here are attractive because they do not require changing the underlying implementation at all before a new operator can be introduced. More generally, because our CPS transformation extends a standard one, it can be incorporated into CPS-based language implementations.

Like us, Dybvig et al. [30] recently extended the standard CPS transformation to deal with delimited-control operators other than `shift` and `reset`. Also recently, Kiselyov [55] used `shift` and `reset` to implement our three pairs of dynamic operators without capturing undelimited continuations or keeping mutable state, as we do here—though his solution is more uniform, and his answer type is a union rather than a function type. As mentioned in Section 4, Filinski early on represented a monad that supports `control` and `prompt`. A decade later, Biernacki et al. [13] propose a *dynamic CPS* to account for `control` and

`prompt`. These two pieces of work each lead to yet another closely-related implementation of `control` and `prompt` in terms of `shift` and `reset`. We stress the connection between these four results: Danvy and Filinski designed `shift` and `reset` precisely to expose delimited continuations in direct style. In part because `shift` and `reset` enjoy a simple CPS transformation as a reasoning tool, they are especially well-studied and widely-applied among delimited-control operators, for instance to represent monads. By equipping dynamic operators also with CPS transformations, we hope to make them easier to use and implement in a way that generalizes to future operators.

5.1 Applications in practical programming

Besides accounting for 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 [77, Section 6], specifically its recursive answer type [39, Section 4.2], which can be depicted graphically as a flowchart. Three special cases of such interactions are

1. the interaction between a Web server and user agents [28, 45, 70];
2. the interaction between a cursor iterating over a collection and its client (as epitomized in the classic same-fringe problem) [54]; and
3. the interaction between a zipper [1, 50, 51] and its client [56, 57].

Whereas many practical programming examples in the literature call for delimited continuations to be recursive, very few call for them to be dynamic: usually a dynamically captured delimited context is invoked right under a delimiter anyway, so it could as well have been captured statically. Recently, though, several programming examples have been proposed to contrast static and dynamic delimited continuations:

1. copying versus reversing a list [9, Section 4.6] [12, Section 2.3] [13, Section 9];
2. generating Pythagorean triples in increasing versus decreasing order [9, Section 5.6];
3. depth-first versus breadth-first same-fringe [14, Section 3] [15, Section 4]; and
4. depth-first versus breadth-first tree labeling [15, Section 5].

We hope that our and other [13, 30, 55] CPS transformations and simulations in terms of `shift` and `reset` will prompt more examples in the future by making dynamic delimited control easier to reason about, use, and implement.

5.2 Multiple delimiter labels

We have not analyzed Gunter et al.'s control operators `cupto` and `set` [47, 48], which generalize `control0` and `prompt0` to multiple delimiter labels, or *prompts*. Briefly, `set` adds to the context a delimiter with a specified label; `cupto` captures the context up to the nearest dynamically enclosing delimiter with the specified label. This additional flexibility may enable control operators from different families, such as `control` and `reset`, to interact harmoniously; we have not tackled this issue here.

In general, labeling delimiters helps the programmer separate concerns and maintain modularity, but makes it harder to reason about programs. In these regards, it resembles multiple mutable storage cells and extensible data types like ML exceptions. In fact, Gunter et al. [48] use `cupto` and `set` to implement the latter language features and vice versa. It would be illuminating to relate the typing and semantics of these features, in contrast to existing work on undelimited continuations [81]. Moreover, it seems possible to use this paper's technique to implement `cupto` and `set` without the overhead of undelimited continuations. However, we need more machinery (a simulation relation between machine states, for example), and we leave it to future work.

One practical use of delimiter labels and recursive continuations is Kiselyov et al.'s translation from dynamic binding to delimited control [60]. They treat the evaluation context as a recursive data structure and map each dynamic variable to a delimiter label.

Another application of delimiter labels and potentially recursive continuations lies in Balat et al.'s type-directed partial evaluator for the λ-calculus with products and sums [7]. To efficiently normalize a λ-term that uses sums, Balat et al.'s algorithm uses Gunter et al.'s `cupto` operator [47, 48], rather than `shift` as in previous work by Balat and Danvy [6]. 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 delimiter labels for `cupto`. (In 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 Balat et al.'s list of delimiter labels would be pleasingly identified with the stack of delimiters on the evaluation context in Gunter et al.'s operational semantics. An implementation of `cupto` or `shift` that does not capture undelimited continuations [58] would also make the algorithm more efficient [44].

Acknowledgements This paper would not be written without Oleg Kiselyov's enlightening and encouraging comments. It would be much more confusing without Olivier Danvy's patient guidance. Thanks to Chris Barker, John Clements, Matthias Felleisen, Andrzej Filinski, Shriram Krishnamurthi, Kevin Millikin, Stuart Shieber, Sam Tobin-Hochstadt, and the reviewers for ICFP 2004, the 2004 Scheme workshop, and *HOSC*. Most of this work was done at Harvard University, supported by the United States National Science Foundation Grant BCS-0236592.

Appendix: Deriving an abstract machine for `control` and `prompt`

In this appendix, we derive an abstract machine for `control` and `prompt` from the CPS equations in Figure 1 and Section 3.2.2. The correctness of the equations then follows from that of our derivation and the resulting machine.

We begin by turning the equations into an evaluator for `control` and `prompt` that passes environments explicitly. To express inductive data types in Scheme, we use Friedman et al.'s `define-datatype` and `cases` macros [42]. The definition of `expression` below illustrates their usage.

```
; type expression
(define-datatype expression expression?
  (var (id symbol?))
  (lam (id symbol?) (body expression?))
  (app (rator expression?) (rand expression?))
  (prompt (body expression?))
  (control (id symbol?) (body expression?)))

; type value = value -> context -> 1+context -> value
(define value? procedure?)
```

```
; type environment = list (symbol * value)
(define (environment? env)
  (or (null? env)
      (and (pair? env)
           (pair? (car env))
           (symbol? (caar env))
           (value? (cdar env))
           (environment? (cdr env)))))
```

```
; type context = value -> 1+context -> value
(define context? procedure?)
(define (maybe ?)
  (lambda (mc) (or (not mc) (? mc))))
```

```
; eval: expression environment -> context -> 1+context -> value
(define (eval exp env) (lambda (c) (lambda (mc)
  (cases expression exp
    (var (id)
     ((c (cdr (assoc id env))) mc))
    (lam (id body)
     (let ((f (lambda (x) (eval body (cons (cons id x) env)))))
       ((c f) mc)))
    (app (rator rand)
     (((eval rator env)
       (lambda (f) ((eval rand env) (lambda (x) ((f x) c)))))
      mc))
    (prompt (body)
     ((c (((eval body env) send) #f)) mc))
    (control (id body)
     (let ((f (lambda (x) (lambda (c2) (lambda (mc2)
                (((compose c mc) x) (compose c2 mc2)))))))
       (((eval body (cons (cons id f) env)) send) #f)))))))
```

```
; evaluate: expression -> value
(define (evaluate exp)
  (((eval exp '()) send) #f))
```

First, we defunctionalize functional values to yield closures of two kinds: λ-abstracted functions (`func`) and `control`-captured contexts (`ctxt`).

```
; type value
(define-datatype value value?
  (func (id symbol?) (body expression?) (env environment?))
  (ctxt (c context?) (mc (maybe context?))))
```

```
; eval: expression environment -> context -> 1+context -> value
(define (eval exp env) (lambda (c) (lambda (mc)
  (cases expression exp
    (var (id)
     ((c (cdr (assoc id env))) mc))
    (lam (id body)
     (let ((f (func id body env))) ; closure conversion
       ((c f) mc)))
    (app (rator rand)
     (((eval rator env)
       (lambda (f)
        ((eval rand env)
         (lambda (x)
          (cases value f           ; closure conversion
            (func (id body env)
             ((eval body (cons (cons id x) env)) c))
            (ctxt (c1 mc1)
             (lambda (mc2)
               (((compose c1 mc1) x) (compose c mc2)))))))))
      mc))
    (prompt (body)
     ((c (((eval body env) send) #f)) mc))
    (control (id body)
     (let ((f (ctxt c mc)))        ; closure conversion
       (((eval body (cons (cons id f) env)) send) #f)))))))
```

Second, we introduce metacontinuations (type `metacont`) and transform the evaluator to CPS. For clarity, we uncurry `context` functions and the `eval` function, so we redefine the `send` and `compose` functions from Section 3.2.1.

```
; type context = value 1+context metacont -> value
(define context? procedure?)
; type metacont = value -> value
(define metacont? procedure?)
; eval: expression environment context 1+context metacont -> value
(define (eval exp env c mc d)
  (cases expression exp
    (var (id)
     (c (cdr (assoc id env)) mc d))
    (lam (id body)
     (c (func id body env) mc d))
    (app (rator rand)
     (eval rator env
       (lambda (f mc d)
         (eval rand env
           (lambda (x mc d)
             (cases value f
               (func (id body env)
                (eval body (cons (cons id x) env) c mc d))
               (ctxt (c1 mc1)
                ((compose c1 mc1) x (compose c mc) d))))
           mc d))
       mc d))
    (prompt (body)
     (eval body env send #f (lambda (v) (c v mc d))))
    (control (id body)
     (eval body (cons (cons id (ctxt c mc)) env) send #f d))))
; send: context
(define (send v mc d)
  (if mc (mc v #f d) (d v)))
; compose: context 1+context -> context
(define (compose c mc1)
  (if mc1 (lambda (v mc2 d) (c v (compose mc1 mc2) d)) c))
; evaluate: expression -> value
(define (evaluate exp)
  (eval exp '() send #f (lambda (v) v)))
```

Finally, we defunctionalize contexts (of type `context`) and metacontinuations (of type `metacont`) in the CPS evaluator to yield an abstract machine, expressed as first-order, tail-recursive Scheme code. The intermediate states of this abstract machine are `eval`, `cont`, `meta`, and the start state is `evaluate`.

```
; type context
(define-datatype context context?
  (send)
  (arg (rand expression?) (env environment?) (c context?))
  (fun (f value?) (c context?))
  (compose (c context?) (mc (maybe context?))))
; type metacont
(define-datatype metacont metacont?
  (done)
  (delim (c context?) (mc (maybe context?)) (d metacont?)))
; eval: expression environment context 1+context metacont -> value
(define (eval exp env c mc d)
  (cases expression exp
    (var (id)
     (cont c (cdr (assoc id env)) mc d))
    (lam (id body)
     (cont c (func id body env) mc d))
    (app (rator rand)
```

```
     (eval rator env (arg rand env c) mc d))
    (prompt (body)
     (eval body env (send) #f (delim c mc d)))
    (control (id body)
     (eval body (cons (cons id (ctxt c mc)) env) (send) #f d))))
; cont: context value 1+context metacont -> value
(define (cont c v mc d)
  (cases context c
    (send ()
     (if mc (cont mc v #f d)
            (meta d v)))
    (arg (rand env c)
     (eval rand env (fun v c) mc d))
    (fun (f c)
     (cases value f
       (func (id body env)
        (eval body (cons (cons id v) env) c mc d))
       (ctxt (c1 mc1)
        (cont (compose c1 mc1) v (compose c mc) d))))
    (compose (c mc1)
     (if mc1 (cont c v (compose mc1 mc) d)
             (cont c v mc d)))))
; meta: metacont value -> value
(define (meta d v)
  (cases metacont d
    (done ()
     v)
    (delim (c mc d)
     (cont c v mc d))))
; evaluate: expression -> value
(define (evaluate exp)
  (eval exp '() (send) #f (done)))
```

Biernacki et al. [13, Section 12] also mention this machine. To verify it, we relate it to Biernacka et al.'s definitional machine for `control` and `prompt` [9, 12–15].

1. Our special token `#f`, of type `1+context`, corresponds to their empty context.
2. Our contexts correspond to their contexts. The only interesting case is that our binary context constructor `compose` corresponds to their binary function (not a constructor) $\star$ for concatenating contexts. That is, our `(compose c mc)` corresponds to their $\mathtt{c} \star \mathtt{mc}$.
3. Our metacontinuations correspond to their metacontexts: `(done)` corresponds to nil, and `(delim c mc d)` corresponds to $(\mathtt{c} \star \mathtt{mc}) :: \mathtt{d}$.
4. Our intermediate states `eval`, `cont`, and `meta` correspond to their intermediate states *eval*, $cont_1$, and $cont_2$, respectively.

Each transition in the definitional machine corresponds to one or more transitions in our derived machine: the only interesting case is the `compose` case at the bottom of `cont`, which does not loop infinitely because `c` is finite. Conversely, each transition in our machine is followed by zero or more further transitions, such that these transitions together correspond to one transition in the definitional machine. Hence the two machines terminate on corresponding input programs.

References

1. Abbott, Michael, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2003. Derivatives of containers. In *TLCA 2003: Proceedings of the 6th international conference on typed lambda calculi and applications*, ed. Martin Hofmann, 16–30. Lecture Notes in Computer Science 2701, Berlin: Springer-Verlag.

2. Ager, Mads Sig, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th international conference on principles and practice of declarative programming*, 8–19. New York: ACM Press.
3. Ager, Mads Sig, Olivier Danvy, and Jan Midtgaard. 2004. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters* 90(5):223–232. Extended version available as BRICS Report RS-04-3.
4. ———. 2005. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science* 342(1):149–172. Extended version available as BRICS Report RS-04-28.
5. Ariola, Zena M., Hugo Herbelin, and Amr Sabry. 2004. A type-theoretic foundation of continuations and prompts. In *ICFP '04: Proceedings of the ACM international conference on functional programming*, 40–53. New York: ACM Press.
6. 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.
7. 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.
8. Barker, Chris. 2004. Continuations in natural language. In *CW'04: Proceedings of the 4th ACM SIGPLAN continuations workshop*, ed. Hayo Thielecke, 1–11. Tech. Rep. CSR-04-1, School of Computer Science, University of Birmingham.
9. Biernacka, Małgorzata, Dariusz Biernacki, and Olivier Danvy. 2005. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science* 1(2:5).
10. Biernacka, Małgorzata, and Olivier Danvy. 2007. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science* 375(1–3):76–108.
11. Biernacki, Dariusz, and Olivier Danvy. 2004. From interpreter to logic engine by defunctionalization. In *LOPSTR 2003: 13th international symposium on logic based program synthesis and transformation*, ed. Maurice Bruynooghe, 143–159. Lecture Notes in Computer Science 3018, Berlin: Springer-Verlag.
12. ———. 2006. A simple proof of a folklore theorem about delimited control. *Journal of Functional Programming* 16(3):269–280.
13. Biernacki, Dariusz, Olivier Danvy, and Kevin Millikin. 2005. A dynamic continuation-passing style for dynamic delimited continuations. Report RS-05-16, BRICS, Denmark.
14. Biernacki, Dariusz, Olivier Danvy, and Chung-chieh Shan. 2005. On the dynamic extent of delimited continuations. *Information Processing Letters* 96(1):7–17.
15. ———. 2006. On the static and dynamic extents of delimited continuations. *Science of Computer Programming* 60(3):274–297.
16. 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.
17. 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.
18. Danvy, Olivier. 1994. Back to direct style. *Science of Computer Programming* 22(3):183–195.
19. ———. 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.
20. ———. 2004. On evaluation contexts, continuations, and the rest of the computation. In *CW'04: Proceedings of the 4th ACM SIGPLAN continuations workshop*, ed. Hayo Thielecke. Tech. Rep. CSR-04-1, School of Computer Science, University of Birmingham.
21. ———. 2005. A rational deconstruction of Landin's SECD machine. In *Implementation and application of functional languages: 16th international workshop, IFL 2004*, ed. Clemens Grelck, Frank Huch, Greg Michaelson, and Philip W. Trinder, 52–71. Lecture Notes in Computer Science 3474, Berlin: Springer-Verlag.
22. 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.
23. ———. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on Lisp and functional programming*, 151–160. New York: ACM Press.
24. ———. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2(4):361–391.

25. 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.
26. Danvy, Olivier, and Lasse R. Nielsen. 2001. Defunctionalization at work. In *Proceedings of the 3rd international conference on principles and practice of declarative programming*, 162–174. New York: ACM Press.
27. Danvy, Olivier, and Zhe Yang. 1999. An operational investigation of the CPS hierarchy. In *Programming languages and systems: Proceedings of ESOP'99, 8th European symposium on programming*, ed. S. Doaitse Swierstra, 224–242. Lecture Notes in Computer Science 1576, Berlin: Springer-Verlag.
28. Double, Chris. 2004. Partial continuations. `http://www.double.co.nz/scheme/partial-continuations/partial-continuations.html`.
29. 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.
30. Dybvig, R. Kent, Simon L. Peyton Jones, and Amr Sabry. 2005. A monadic framework for delimited continuations. Tech. Rep. 615, Computer Science Department, Indiana University.
31. 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, Computer Science Department, Indiana University. Also as Tech. Rep. 226.
32. ———. 1988. The theory and practice of first-class prompts. In *POPL '88: Conference record of the annual ACM symposium on principles of programming languages*, 180–190. New York: ACM Press.
33. ———. 1991. On the expressive power of programming languages. *Science of Computer Programming* 17(1–3):35–75.
34. Felleisen, Matthias, and Daniel P. Friedman. 1987. Control operators, the SECD machine, and the λ-calculus. In *Formal description of programming concepts III*, ed. Martin Wirsing, 193–217. Amsterdam: Elsevier Science.
35. Felleisen, Matthias, Daniel P. Friedman, Bruce F. Duba, and John Merrill. 1987. Beyond continuations. Tech. Rep. 216, Computer Science Department, Indiana University.
36. 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.
37. 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.
38. ———. 1996. Controlling effects. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Also as Tech. Rep. CMU-CS-96-119.
39. ———. 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.
40. ———. 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.
41. Friedman, Daniel P., and Christopher T. Haynes. 1985. Constraining control. In *POPL '85: Conference record of the annual ACM symposium on principles of programming languages*, 245–254. New York: ACM Press.
42. Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 2001. *Essentials of programming languages*. 2nd ed. Cambridge: MIT Press.
43. Gapeyev, Vladimir, Michael Y. Levin, and Benjamin C. Pierce. 2000. Recursive subtyping revealed. In *ICFP '00: Proceedings of the ACM international conference on functional programming*, vol. 35(9) of *ACM SIGPLAN Notices*, 221–231. New York: ACM Press.
44. 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.
45. Graunke, Paul Thorsen. 2003. Web interactions. Ph.D. thesis, College of Computer Science, Northeastern University.
46. Grobauer, Bernd, and Zhe Yang. 2001. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation* 14(2–3):173–219.
47. 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.

48. ———. 1998. Return types for functional continuations. http://pauillac.inria.fr/~remy/work/cupto/.
49. 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.
50. Hinze, Ralf, and Johan Jeuring. 2001. Weaving a web. *Journal of Functional Programming* 11(6): 681–689.
51. Huet, Gérard. 1997. The zipper. *Journal of Functional Programming* 7(5):549–554.
52. Johnson, Gregory F., and Dominic Duggan. 1988. Stores and partial continuations as first-class objects in a language and its environment. In *POPL '88: Conference record of the annual ACM symposium on principles of programming languages*, 158–168. New York: ACM Press.
53. 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, Gerald Jay 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.
54. Kiselyov, Oleg. 2004. General ways to traverse collections. http://okmij.org/ftp/Scheme/enumerators-callcc.html; http://okmij.org/ftp/Computation/Continuations.html.
55. ———. 2005. How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Tech. Rep. 611, Computer Science Department, Indiana University.
56. ———. 2005. Two-hole zippers and transactions of various isolation modes. Message to the Haskell mailing list; http://okmij.org/ftp/Haskell/Zipper2.lhs; http://okmij.org/ftp/Computation/Continuations.html.
57. ———. 2005. Zipper as a delimited continuation. Message to the Haskell mailing list; http://okmij.org/ftp/Haskell/Zipper1.lhs; http://okmij.org/ftp/Computation/Continuations.html.
58. ———. 2006. Native delimited continuations in (byte-code) OCaml. http://okmij.org/ftp/Computation/Continuations.html#caml-shift.
59. Kiselyov, Oleg, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *ICFP '05: Proceedings of the ACM international conference on functional programming*, 192–203. New York: ACM Press.
60. Kiselyov, Oleg, Chung-chieh Shan, and Amr Sabry. 2006. Delimited dynamic binding. In *ICFP '06: Proceedings of the ACM international conference on functional programming*, 26–37. New York: ACM Press.
61. Krivine, Jean-Louis. 1985. Un interpréteur du lambda-calcul. http://www.pps.jussieu.fr/~krivine/articles/interprt.pdf.
62. Landin, Peter J. 1964. The mechanical evaluation of expressions. *The Computer Journal* 6(4):308–320.
63. 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.
64. 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.
65. Moggi, Eugenio. 1991. Notions of computation and monads. *Information and Computation* 93(1):55–92.
66. Murphy, Tom, VII, Karl Crary, and Robert Harper. 2005. Distributed control flow with classical modal logic. In *Computer science logic: 19th international workshop, CSL 2005*, ed. C.-H. Luke Ong, 51–69. Lecture Notes in Computer Science 3634, Berlin: Springer-Verlag.
67. Murthy, Chetan R. 1992. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In *CW'92: Proceedings of the ACM SIGPLAN workshop on continuations*, ed. Olivier Danvy and Carolyn Talcott, 49–71. Tech. Rep. STAN-CS-92-1426, Department of Computer Science, Stanford University. ftp://cstr.stanford.edu/pub/cstr/reports/cs/tr/92/1426.
68. Plotkin, Gordon D. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1(2):125–159.
69. Queinnec, Christian. 1993. A library of high-level control operators. *Lisp Pointers* 6(4):11–26.
70. ———. 2004. Continuations and web servers. *Higher-Order and Symbolic Computation* 17(4):277–295.
71. 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.
72. Reynolds, John C. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM national conference*, vol. 2, 717–740. New York: ACM Press. Reprinted with a foreword in *Higher-Order and Symbolic Computation* 11(4):363–397.

73. Sabry, Amr, and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation* 6(3–4):289–360.
74. Sewell, Peter, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. 2005. Acute: High-level programming language design for distributed computation. In *ICFP '05: Proceedings of the ACM international conference on functional programming*, 15–26. New York: ACM Press.
75. Shan, Chung-chieh. 2004. Delimited continuations in natural language: Quantification and polarity sensitivity. In *CW'04: Proceedings of the 4th ACM SIGPLAN continuations workshop*, ed. Hayo Thielecke, 55–64. Tech. Rep. CSR-04-1, School of Computer Science, University of Birmingham.
76. 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.
77. Sitaram, Dorai, and Matthias Felleisen. 1990. Control delimiters and their hierarchies. *Lisp and Symbolic Computation* 3(1):67–99.
78. Steele, Guy Lewis, Jr. 1978. RABBIT: A compiler for SCHEME. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Also as Memo 474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.
79. Strachey, Christopher, and Christopher P. Wadsworth. 1974. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Programming Research Group, Oxford University Computing Laboratory. Reprinted with a foreword in *Higher-Order and Symbolic Computation* 13(1–2):135–152.
80. 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.
81. Thielecke, Hayo. 2001. Contrasting exceptions and continuations.
82. Thiemann, Peter. 1999. Combinators for program generation. *Journal of Functional Programming* 9(5): 483–525.
83. Wadler, Philip L. 1994. Monads and composable continuations. *Lisp and Symbolic Computation* 7(1): 39–56.
84. Wand, Mitchell. 1980. Continuation-based program transformation strategies. *Journal of the ACM* 27(1): 164–180.