# Functional Pearl: The Great Escape

## Or, How to jump the border without getting caught

David Herman

Northeastern University

dherman@ccs.neu.edu

## Abstract

Filinski showed that `callcc` and a single mutable reference cell are sufficient to express the delimited control operators `shift` and `reset`. However, this implementation interacts poorly with dynamic bindings like exception handlers. We present a variation on Filinski's encoding of delimited continuations that behaves appropriately in the presence of exceptions and give an implementation in Standard ML of New Jersey. We prove the encoding correct with respect to the semantics of delimited dynamic binding.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

***General Terms*** Languages

***Keywords*** continuations, delimited control, dynamic binding

## 1. Introduction

First-class continuations are prevalent in mature implementations of functional programming languages such as Scheme (Kelsey et al. 1998) and Standard ML (Milner et al. 1990). They often appear in *undelimited* form: `call-with-current-continuation` in Scheme (`call/cc` for short) and the library function `callcc` of Standard ML of New Jersey both capture the entire program continuation. But many applications call for *delimited continuations*, which are characterized by the ability to evaluate an expression as if in a new, empty evaluation context (Felleisen 1988; Danvy and Filinski 1989). Restricting the scope of control effects protects the computational abstractions of idioms like threads, read-eval-print loops, and reflective towers of metacircular interpreters (Felleisen 1988; Sitaram and Felleisen 1990; Wand and Friedman 1986).

The problem we solve here is one that arose from a practical need: could we implement the delimited control operators `shift` and `reset` in Standard ML of New Jersey without modifying the compiler? The answer would seem to be an obvious "yes"—Sitaram and Felleisen (1990) showed that delimited control can be expressed in terms of undelimited control, and Andrzej Filinski published a well-known translation of `shift` and `reset` for languages with `callcc` and mutable reference cells (Filinski 1994). However, recent work by Kiselyov et al. (2006) showed that Filinski's implementation does not work for languages with exceptions.



*ICFP'07* October 1–3, 2007, Freiburg, Germany.



Filinski's encoding is correct in a simple $\lambda$-calculus. But a naïve translation of this encoding into ML results in a subtle bug: a reified continuation closes over *all* of the dynamically bound exception handlers. By contrast, the semantics of delimited dynamic binding (Kiselyov et al. 2006) prescribes that delimited continuations only close over a *part* of the dynamic environment.

Perhaps it should not be surprising that `callcc` and exceptions interact in non-trivial ways. Nevertheless, Filinski's SML implementation of `shift` and `reset` was considered standard for a decade before the problem of exception handlers was identified. Kiselyov et al. (2006) describe the problem of dynamic binding in the presence of delimited control and offer solutions using either modified semantics for dynamic variables or `callcc` or low-level primitives for concatenating and splitting dynamic environments.

This paper demonstrates that, for the particular problem of dynamically binding exception handlers, no changes to the underlying semantics are necessary. We present an implementation of `shift` and `reset` in the presence of exceptions and dynamically bound exception handlers using the same tools as the original Filinski encoding—`callcc` and a single, mutable reference cell—using a technique we call The Great Escape.

The next section reviews the semantics of `shift` and `reset`. Section 3 describes the original Filinski encoding, spells out the key invariant for the simulation, and proves the simulation correct with respect to a semantics without exceptions. Section 4 introduces exceptions and dynamically bound exception handlers to the specification semantics and illustrates the problems with the Filinski encoding. Section 5 presents our solution and proves it correct with respect to the updated semantics. Section 6 concludes.

## 2. Delimited continuations

Delimited control operators generalize first-class continuations by allowing programs to perform subcomputations as if in a new, empty evaluation context. This creates a kind of computational sandbox for the evaluation of a subexpression, which is useful for separating multiple stages of computation. Control delimiters serve as computational abstraction boundaries, preventing child stages from capturing or returning to parent stages.

For example, a simple read-eval-print loop implemented in Scheme might track the expression count in a local accumulator `n`. A naïve implementation directly evaluates the input with `eval`:

```
(define (repl)
  (define (loop n)
    (display n)
    (display " - ")
    (display (eval (read)))
    (newline)
    (loop (+ n 1)))
  (loop 0))
```

The accumulator appears to be strictly increasing, but an interaction can capture the current continuation and return to it later:

```
0 - (define k #f)
1 - (call/cc (lambda (x) (set! k x)))
2 - (k 17)
17
2 - (k 42)
42
2 -
```

Because `call/cc` captures the entire program continuation, the user interaction is able to escape the current context and return to an earlier point in the program history with a previous value for $n$.

A better implementation of `repl` could use a *control delimiter*. The `reset` operator evaluates its argument expression as if in an empty context, which prevents code from capturing or re-entering the continuation of the delimiter. In the case of `repl`, wrapping the call to `eval` protects the main loop from being captured by user code:

```
(display (reset (eval (read))))
```

The corresponding operator `shift` is similar to `call/cc`, but it can only capture the local portion of the continuation up to the most recent delimiter.

With `shift` and the new `repl`, the interaction sequence behaves as expected:

```
0 - (define k #f)
1 - (shift (lambda (x) (set! k x)))
2 - (k 17)
17
3 - (k 42)
42
4 -
```

There is another important difference between `call/cc` and `shift`: the jumps happen at different points. With `call/cc`, capturing the current continuation has no observable effect, but invoking a continuation aborts the current continuation. The reverse is true of `shift`; capturing a continuation immediately aborts it, but invoking a continuation does not abort.

For example, the result of

```
(+ 1 (call/cc (lambda (k) 42)))
```

is 43, whereas the result of

```
(reset (+ 1 (shift (lambda (k) 42))))
```

is 42. That is, `call/cc` does not abandon the current context when `k` is not invoked, but `shift` abandons the context immediately.

Because they do not jump, continuations captured by `shift` can often be composed like ordinary functions. For example, the result of

```
(reset (+ 1 (shift (lambda (k) (k (k 17))))))
```

is 19, because after `shift` aborts the local context, the two nested applications of `k` evaluate normally, without subsequent aborts.

### 2.1 Specification

Filinski defines the behavior of `shift` and `reset` via a "double-barrelled" (Thielecke 2002) continuation-passing interpretation in the $\lambda$-calculus. Each function in the interpretation receives two extra arguments: a local continuation, representing the immediate control context up to the dynamically nearest delimiter, and a *meta-continuation*, representing the rest of the control context after the delimiter (Wand and Friedman 1986).

Biernacki et al. (2006) instead use an elegant abstract machine semantics with two evaluation contexts to represent the two continuations. We use a simplified version of this abstract machine, with only a control expression $e$, a local continuation represented as an evaluation context $E$, and a metacontinuation $K$, represented as a stack of local contexts $E$.

$$\begin{array}{lcl} e & ::= & x \mid v \mid e\ e \mid \mathcal{S}x.e \mid \#e \\ v & ::= & \lambda x.e \mid \langle E \rangle \\ E & ::= & [\,] \mid E\ e \mid v\ E \\ K & ::= & \texttt{halt} \mid E :: K \\ C & ::= & E[e],\ K \end{array}$$

The syntax of this language is the untyped $\lambda$-calculus with additional expression forms $S$ for `shift` and $\#$ for `reset`, as well as reified continuations $\langle E \rangle$. A program state $C$ consists of a control expression $e$ and its two continuations $E$ and $K$.

The rules for evaluating this language are mostly straightforward:

$$\begin{array}{lcl} E[\lambda x.e\ v],\ K & \xrightarrow{\mathcal{S}} & E[e[v/x]],\ K \\ E[\mathcal{S}x.e],\ K & \xrightarrow{\mathcal{S}} & e[\langle E \rangle / x],\ K \\ E[\#e],\ K & \xrightarrow{\mathcal{S}} & e,\ E :: K \\ E[\langle E' \rangle\ v],\ K & \xrightarrow{\mathcal{S}} & E[\#E'[v]],\ K \\ v,\ E :: K & \xrightarrow{\mathcal{S}} & E[v],\ K \end{array}$$

The first rule is standard $\beta_v$ reduction. Evaluation of a `shift` expression $\mathcal{S}x.e$ captures and aborts the current local continuation, binding it to $x$. The `reset` form $\#e$ pushes the current local continuation onto the metacontinuation, protecting it from capture by subsequent calls to `shift`. Invoking a continuation $\langle E' \rangle$ is more involved: the value $v$ is plugged into the continuation $E'$, and evaluated within the current evaluation context $E$, but with the addition of a new delimiter. This prevents subsequent control effects in the body of $E'$ from capturing $E$. It is this extra delimiter that distinguishes the behavior of $\mathcal{S}$ from the control operator $\mathcal{F}$ (Felleisen 1987; Danvy and Filinski 1989; Sitaram 1994; Shan 2004; Kiselyov 2005; Biernacki et al. 2006). Finally, returning a value locally pops the next local continuation off of the global stack and continues returning.

## 3. Filinski encoding

Armed with a specification, let us now take a look at the Filinski encoding, guided by a concrete implementation in SML/NJ.

### 3.1 Implementation

In SML/NJ, a continuation of argument type `'a` is represented as a special value of type `'a cont`, and can be invoked with a library function

```
val throw : 'a cont -> 'a -> 'b
```

But the Scheme-style representation of first-class continuations as functions is closer to the Filinski encoding, so Filinski's SML implementation provides a functional abstraction for continuations. To distinguish it from the built-in `callcc`, Filinski names this function `escape` after the similar construct of Reynolds (1972).

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

Here the type `void` is intended to be uninhabited, so it can be used for functions that never return. With this in mind, the type for `escape` can be read as a Scheme-like functional `callcc`: its

argument is a function that consumes a continuation function of type `'a -> void`. The `coerce` function is analogous to `throw` in that it informs the ML type checker that an expression that never returns can have any type. This much of the implementation is simple:

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

The main library defines `shift` and `reset` and is parameterized over the type `ans` of *intermediate answers*, the values returned by local continuations.

```
signature CONTROL =
sig
    type ans
    val shift : (('a -> ans) -> ans) -> 'a
    val reset : (unit -> ans) -> ans
end;
```

Figure 1 shows the implementation of the Filinski encoding in SML/NJ. The reference cell `mk` serves as the representation of the metacontinuation, and the auxiliary function `return` simulates returning a value to the current metacontinuation by applying the contents of `mk`.

The implementation of `shift` calls `escape` to capture and bind the ML continuation to `k`, which serves as the representation of the local continuation $E$. The argument to `f` wraps the call to `k` with `reset` to implement the specified behavior of applying $E$. The `shift` function relies on `return` to return its result directly to the current metacontinuation, rather than to its local continuation. Of course, in the specification, this happens immediately by discarding the local context $E$ before evaluating the body, whereas in the implementation, this control effect happens after `f` returns.

The `reset` function implements the stack discipline of the metacontinuation. The function first captures the current ML continuation and binds it to `k`, representing the local continuation $E$. The metacontinuation cell is updated to a new function that, when applied, first simulates popping $E$ from the stack by restoring the cell to its previous state, before returning the result to `k`. Again, the result of the `reset` expression returns to the current metacontinuation via a final call to `return`.

The bewildered reader may at this point be wondering how we have managed to represent a delimited continuation $E$ with an undelimited continuation `k` throughout this implementation. To understand why this works, we must consider the invariant maintained by this encoding.

### 3.2 Borders

The challenge in representing delimited continuations with `callcc` is preventing the extra information captured in an undelimited continuation from affecting the program behavior. The encoding relies on an invariant that every captured continuation contains a *border* that forces control to jump out of the captured continuation and return to the metacontinuation, effectively discarding the extraneous portion of the undelimited continuation.

To make this notion precise, consider a little model of our implementation language with undelimited first-class continuations and a single mutable reference cell:

$$\begin{array}{lcl} \mathsf{e} & ::= & \mathsf{x} \mid \mathsf{v} \mid \uparrow \mid \downarrow \mathsf{e} \mid \mathcal{C}\mathsf{x}.\mathsf{e} \\ \mathsf{v} & ::= & \lambda \mathsf{x}.\mathsf{e} \mid \langle \mathsf{E} \rangle \\ \mathsf{E} & ::= & [\,] \mid \mathsf{E}\,\mathsf{e} \mid \mathsf{v}\,\mathsf{E} \mid \downarrow \mathsf{E} \\ \mathsf{C} & ::= & \mathsf{E}[\mathsf{e}],\ \mathsf{v} \end{array}$$

We use a sans serif font to distinguish terms in the model of the implementation language from the model of the specification language. The up-arrow expression form $\uparrow$ denotes a dereference of the single cell and the down-arrow form $\downarrow e$ denotes assignment to the cell. The expression form $\mathcal{C}$x.e captures a continuation and binds it to x in the scope of e. A program state C consists of a continuation E, a control expression e, and the current value v of the single reference cell. We can define additional conveniences as syntactic sugar, such as

$$\begin{array}{rcl} \mathsf{let}\ \mathsf{x} \leftarrow \mathsf{e}\ \mathsf{in}\ \mathsf{e}' & = & (\lambda \mathsf{x}.\mathsf{e}')\ \mathsf{e} \\ (\mathsf{e}; \mathsf{e}') & = & \mathsf{let}\ \_ \leftarrow \mathsf{e}\ \mathsf{in}\ \mathsf{e}' \\ & = & (\lambda \_.\mathsf{e}')\ \mathsf{e} \end{array}$$

where _ represents any variable not free in $\mathsf{e}'$.

The operational semantics for this tiny calculus is an untyped, call-by value $\lambda$-calculus, much like Scheme or an untyped ML:

$$\begin{array}{lcl} \mathsf{E}[\lambda \mathsf{x}.\mathsf{e}\ \mathsf{v}],\ \mathsf{v}' & \xrightarrow{\mathcal{C}} & \mathsf{E}[\mathsf{e}[\mathsf{v}/\mathsf{x}]],\ \mathsf{v}' \\ \mathsf{E}[\mathcal{C}\mathsf{x}.\mathsf{e}],\ \mathsf{v} & \xrightarrow{\mathcal{C}} & \mathsf{E}[\mathsf{e}[\langle \mathsf{E} \rangle / \mathsf{x}]],\ \mathsf{v} \\ \mathsf{E}[\langle \mathsf{E}' \rangle\ \mathsf{v}],\ \mathsf{v}' & \xrightarrow{\mathcal{C}} & \mathsf{E}'[\mathsf{v}],\ \mathsf{v}' \\ \mathsf{E}[\uparrow],\ \mathsf{v} & \xrightarrow{\mathcal{C}} & \mathsf{E}[\mathsf{v}],\ \mathsf{v} \\ \mathsf{E}[\downarrow \mathsf{v}],\ \mathsf{v}' & \xrightarrow{\mathcal{C}} & \mathsf{E}[\mathsf{v}],\ \mathsf{v} \end{array}$$

As before, application follows $\beta_v$. The $\mathcal{C}$x.e form reifies the entire program continuation E and binds it to x to evaluate e. Application of a captured continuation $\mathsf{E}'$ replaces the current continuation. The forms $\uparrow$ and $\downarrow$ v retrieve and update the value of the mutable cell, respectively.

The simulation invariant, shown in Figure 3, formalizes the important components of the simulation. The judgment $\mathsf{C} \sim C$ indicates that a configuration C in the implementation machine simulates a configuration $C$ in the specification machine if it evaluates related terms in related continuations or if it simulates returning to the metacontinuation by applying the current value of the cell to a related value.

The judgment $\mathsf{e} \sim e$ relates expressions of the specification and implementation machines. Delimited continuations are encoded as functions that apply related undelimited continuations under a `reset`. Plugging a value into a local continuation is encoded as application of that continuation's encoding. A `shift` expression is simulated by capturing the current continuation and binding k to a function that applies the captured continuation under a fresh delimiter; to simulate the control effect of aborting the local continuation, the body expression is evaluated in the context of a border which gives the result to the metacontinuation. The encoding of `reset` captures the current continuation and binds it to a fresh k in order

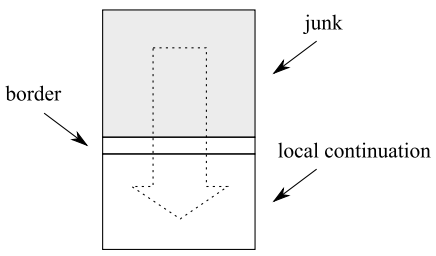


**Figure 2.** A continuation in the simulation.

```
functor Control (type ans) : CONTROL =
struct
    open Escape
    exception MissingReset
    type ans = ans

    val mk : (ans -> void) ref = ref (fn _ => raise MissingReset)

    fun return x = coerce (!mk x)

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

    fun shift f =
        escape (fn k => return (f (fn v => reset (fn () => coerce (k v)))))
end;
```

**Figure 1.** The Filinski encoding in Standard ML of New Jersey.

to push the current control context onto the metacontinuation; when the metacontinuation is invoked, it pops this extra frame by reverting to its previous value. Again, the body of the `reset` is evaluated under a border that returns its result to the metacontinuation.

Judgment $\mathsf{E} \sim E$ states that a continuation in the implementation simulates a local continuation of the specification even if it contains extra "junk" at the beginning, so long as the relevant local portion is delimited with a border. The judgment $\mathsf{E} \approx E$ relates just that local portion to the specified local continuation.

Finally, the judgment $\mathsf{v} \sim K$ relates the reference cell of the implementation machine to the metacontinuation. For simplicity, we fix an initial continuation $\mathsf{E}_0$ and an initial value $\mathsf{v}_0$ in the reference cell; if a computation terminates, the final value is returned to $\mathsf{E}_0$ and the reference cell is reverted to $\mathsf{v}_0$.

An implementation in the model ML can be directly derived from the invariant.

$$
\begin{array}{rcl}
[\![x]\!] & = & \mathsf{x} \\
[\![\lambda x.e]\!] & = & \lambda\mathsf{x}.[\![e]\!] \\
[\![e_1\ e_2]\!] & = & [\![e_1]\!]\ [\![e_2]\!] \\
[\![\mathcal{S}k.e]\!] & = & \mathcal{C}\mathsf{k}'.\mathsf{let\ x} \leftarrow (\mathsf{let\ k} \leftarrow \lambda\mathsf{x}.[\![\#(k'\ x)]\!]\ \mathsf{in}\ [\![e]\!])\ \mathsf{in} \\
 & & \qquad ((\uparrow)\ \mathsf{x}) \\
 & & \mathsf{k}' \notin FV([\![e]\!]) \\
[\![\#e]\!] & = & \mathcal{C}\mathsf{k}.(\mathsf{let\ m} \leftarrow (\uparrow)\ \mathsf{in} \\
 & & \qquad \downarrow\lambda\mathsf{x}.(\downarrow\mathsf{m}; \mathsf{k\ x}) \\
 & & \quad \mathsf{let\ x} \leftarrow [\![e]\!]\ \mathsf{in}\ ((\uparrow)\ \mathsf{x})) \\
 & & \mathsf{k} \notin FV([\![e]\!])
\end{array}
$$

Translating a top-level program configuration $e$, `halt` to the configuration $\mathsf{E}_0[[\![\#e]\!]]$, $\mathsf{v}_0$ yields a valid initial configuration for the simulation. The simulation theorem shows that Filinski's encoding is a faithful implementation of the specification.

**Lemma 1 (substitution preserves invariant)**
*If* $\mathsf{e} \sim e$ *and* $\mathsf{v} \sim v$ *then* $\mathsf{e}[\mathsf{v}/\mathsf{x}] \sim e[v/x]$.

*Proof* Structural induction on $\mathsf{e}$, $e$, $\mathsf{v}$ and $v$. □

**Theorem 1 (simulation, Filinski encoding)**
*With the languages and simulation invariant of Sections 2.1 and 3.2, if* $\mathsf{C} \sim C$ *and* $C \xrightarrow{\mathcal{S}} C'$*, then there exists a term* $\mathsf{C}'$ *such that* $\mathsf{C} \xrightarrow{\mathcal{C}} \mathsf{C}'$ *and* $C' \sim \mathsf{C}'$.

$$\text{[I-EVAL]}\quad \frac{E[e] \neq v' \qquad \mathsf{E} \sim E \qquad \mathsf{e} \sim e \qquad \mathsf{v} \sim K}{\mathsf{E}[\mathsf{e}],\ \mathsf{v} \sim E[e],\ K}$$

$$\text{[I-RETURN]}\quad \frac{\mathsf{v} \sim v \qquad \mathsf{v}' \sim K \qquad \text{any } \mathsf{E}}{\mathsf{E}[(\mathsf{v}'\ \mathsf{v})],\ \mathsf{v}' \sim v,\ K}$$

$$\text{[I-VAR]}\quad \frac{}{\mathsf{x} \sim x} \qquad \text{[I-ABS]}\quad \frac{\mathsf{e} \sim e}{\lambda\mathsf{x}.\mathsf{e} \sim \lambda x.e} \qquad \text{[I-APP]}\quad \frac{\mathsf{e} \sim e \qquad \mathsf{e}' \sim e'}{\mathsf{e}\ \mathsf{e}' \sim e\ e'}$$

$$\text{[I-REIFY]}\quad \frac{\mathsf{e} \sim \#(y\ x) \qquad \mathsf{E} \sim E}{\lambda\mathsf{x}.\mathsf{e}[\langle\mathsf{E}\rangle/\mathsf{y}] \sim \langle E\rangle} \qquad \text{[I-REFLECT]}\quad \frac{\mathsf{E} \sim E \qquad \mathsf{v} \sim v}{(\langle\mathsf{E}\rangle\ \mathsf{v}) \sim E[v]}$$

$$\text{[I-SHIFT]}\quad \frac{\mathsf{k}' \notin FV(\mathsf{e}) \qquad \mathsf{e} \sim e \qquad \mathsf{e}' \sim \#(k'\ x)}{\mathcal{C}\mathsf{k}'.\mathsf{let\ x} \leftarrow (\mathsf{let\ k} \leftarrow \lambda\mathsf{x}.\mathsf{e}'\ \mathsf{in\ e})\ \mathsf{in} \sim \mathcal{S}k.e\ ((\uparrow)\ \mathsf{x})}$$

$$\text{[I-RESET]}\quad \frac{\mathsf{k} \notin FV(\mathsf{e}) \qquad \mathsf{e} \sim e}{\mathcal{C}\mathsf{k}.(\mathsf{let\ m} \leftarrow (\uparrow)\ \mathsf{in}\ \downarrow\lambda\mathsf{x}.(\downarrow\mathsf{m}; \mathsf{k\ x});\ \mathsf{let\ x} \leftarrow \mathsf{e}\ \mathsf{in}\ ((\uparrow)\ \mathsf{x})) \sim \#e}$$

$$\text{[I-BORDER]}\quad \frac{\mathsf{E} \approx E \qquad \text{any } \mathsf{E}'}{\mathsf{E}'[\mathsf{let\ x} \leftarrow \mathsf{E}\ \mathsf{in}\ ((\uparrow)\ \mathsf{x})] \sim E}$$

$$\text{[I-HOLE]}\quad \frac{}{[\,] \approx [\,]} \qquad \text{[I-OPERATOR]}\quad \frac{\mathsf{E} \approx E \qquad \mathsf{e} \sim e}{\mathsf{E}\ \mathsf{e} \approx E\ e} \qquad \text{[I-OPERAND]}\quad \frac{\mathsf{v} \sim v \qquad \mathsf{E} \approx E}{\mathsf{v}\ \mathsf{E} \approx v\ E}$$

$$\text{[I-EMPTY]}\quad \frac{}{\lambda\mathsf{x}.(\downarrow\mathsf{v}_0; \mathsf{E}_0\ \mathsf{x}) \sim \mathtt{halt}} \qquad \text{[I-PUSH]}\quad \frac{\mathsf{v} \sim K \qquad \mathsf{E} \sim E}{\lambda\mathsf{x}.(\downarrow\mathsf{v}; \langle\mathsf{E}\rangle\ \mathsf{x}) \sim E :: K}$$

**Figure 3.** The simulation invariant.

*Proof* By cases on the reduction rules of the specification machine and the definition of the invariant relations, using Lemma 1. □

**Corollary** *If* $\mathsf{C} \sim C$ *and* $C \xrightarrow{\mathcal{S}}{}^{*} v$, `halt` *then there exists a value* $\mathsf{v}$ *such that* $\mathsf{C} \xrightarrow{\mathcal{C}}{}^{*} \mathsf{E}_0[\mathsf{v}]$, $\mathsf{v}_0$ *and* $\mathsf{v} \sim v$.

## 4. Exceptions

Theorem 1 assures us that the implementation is correct—assuming, of course, that the model is a realistic reflection of the implementation language. But what happens when the implementation language contains exceptions?

### 4.1 SML Exceptions

Let us see what happens if we extend the model of the implementation language with a fixed set of exception constants $\varepsilon$ and exception handlers $\mathsf{h}$:

$$\begin{array}{lcl} \mathsf{e} & ::= & \ldots \mid \mathsf{handle\ h} \Rightarrow \mathsf{e} \\ \mathsf{v} & ::= & \ldots \mid \mathsf{raise} \mid \varepsilon \\ \mathsf{h} & ::= & \mathsf{x} \mid \varepsilon \end{array}$$

Furthermore, we must be able to install exception handlers during evaluation. In SML/NJ, a captured continuation closes over its installed exception handlers so it suffices to add handler frames to the definition of evaluation contexts $\mathsf{E}$:

$$\mathsf{E} \quad ::= \quad \ldots \mid \mathsf{E\ handle\ h} \Rightarrow \mathsf{e}$$

Now we can extend the implementation semantics with rules for raising and handling exceptions:

$$\begin{array}{l} \mathsf{E}[\mathsf{E}'[\mathsf{raise}\ \varepsilon]\ \mathsf{handle}\ \varepsilon \Rightarrow \mathsf{e}],\ \mathsf{v} \quad \xrightarrow{\mathcal{C}} \quad \mathsf{E}[\mathsf{e}],\ \mathsf{v} \\ \qquad \text{if } \mathsf{E}' \neq \mathsf{E}_1[\mathsf{E}_2\ \mathsf{handle\ h} \Rightarrow \mathsf{e}'] \\ \mathsf{E}[\mathsf{E}'[\mathsf{raise}\ \varepsilon]\ \mathsf{handle}\ \mathsf{x} \Rightarrow \mathsf{e}],\ \mathsf{v} \quad \xrightarrow{\mathcal{C}} \quad \mathsf{E}[\mathsf{e}[\varepsilon/\mathsf{x}]],\ \mathsf{v} \\ \qquad \text{if } \mathsf{E}' \neq \mathsf{E}_1[\mathsf{E}_2\ \mathsf{handle\ h} \Rightarrow \mathsf{e}'] \\ \mathsf{E}[\mathsf{E}'[\mathsf{raise}\ \varepsilon]\ \mathsf{handle}\ \varepsilon' \Rightarrow \mathsf{e}],\ \mathsf{v} \quad \xrightarrow{\mathcal{C}} \quad \mathsf{E}[\mathsf{raise}\ \varepsilon],\ \mathsf{v} \\ \qquad \text{if } \mathsf{E}' \neq \mathsf{E}_1[\mathsf{E}_2\ \mathsf{handle\ h} \Rightarrow \mathsf{e}'] \text{ and } \varepsilon \neq \varepsilon' \end{array}$$

### 4.2 Delimited dynamic binding

To understand how exceptions interact with delimited control operators, let us also add similar syntax to the specification language:

$$\begin{array}{lcl} e & ::= & \ldots \mid e\ \mathbf{handle}\ h \Rightarrow e \\ v & ::= & \ldots \mid \mathbf{raise} \mid \varepsilon \\ h & ::= & x \mid \varepsilon \end{array}$$

Now we are faced with a design decision, namely how delimited evaluation contexts interact with exception handlers. Specifically, the question is what exception handlers captured continuation should close over:

- none of the current exception handlers;
- all exception handlers in the current continuation; or
- all exception handlers in the current continuation, up to the nearest delimiter.

Kiselyov et al. (2006) argue that the third option is the most sensible semantics. Indeed, if we use a representation of evaluation contexts analogous to Section 4.1, then reified continuations capture exactly the bindings installed since the last delimiter:

$$E \quad ::= \quad \ldots \mid E\ \mathbf{handle}\ h \Rightarrow e$$

Exception handlers are an example of *dynamic bindings*, also known as fluids or implicit parameters (Haynes and Friedman 1987; Hanson 1991; Moreau 1997; Lewis et al. 2000). Because dynamic bindings are associated with their control context, context delimiters should also delimit the scope of dynamic bindings. To wit: capturing a portion of that context should accordingly capture only the relevant portion of current dynamic bindings. Furthermore, applying a captured delimited continuation should install its exception handlers in the context of the current handlers. In other words, installing a delimited continuation has the effect of "splicing" together two sets of handlers.

The new rules of the specification semantics are mostly analogous to those of Section 4.1:

$$\begin{array}{l} E[E'[\mathbf{raise}\ \varepsilon]\ \mathbf{handle}\ \varepsilon \Rightarrow e],\ K \quad \xrightarrow{\mathcal{S}} \quad E[e],\ K \\ \qquad \text{if } E' \neq E_1[E_2\ \mathbf{handle}\ h \Rightarrow e'] \\ E[E'[\mathbf{raise}\ \varepsilon]\ \mathbf{handle}\ x \Rightarrow e],\ K \quad \xrightarrow{\mathcal{S}} \quad E[e[\varepsilon/x]],\ K \\ \qquad \text{if } E' \neq E_1[E_2\ \mathbf{handle}\ h \Rightarrow e'] \\ E[E'[\mathbf{raise}\ \varepsilon]\ \mathbf{handle}\ \varepsilon' \Rightarrow e],\ K \quad \xrightarrow{\mathcal{S}} \quad E[\mathbf{raise}\ \varepsilon],\ K \\ \qquad \text{if } E' \neq E_1[E_2\ \mathbf{handle}\ h \Rightarrow e'] \text{ and } \varepsilon \neq \varepsilon' \\ E[\mathbf{raise}\ \varepsilon],\ E' :: K \quad \xrightarrow{\mathcal{S}} \quad E'[\mathbf{raise}\ \varepsilon] \\ \qquad \text{if } E \neq E_1[E_2\ \mathbf{handle}\ h \Rightarrow e] \end{array}$$

The first three rules correspond to the rules of the implementation semantics. The last rule specifies that uncaught exceptions propagate from child computations to parent computations.

### 4.3 Bug

Unfortunately, the above semantics is not the one implemented by the code in Section 3.1.

For example, according to the semantics of delimited dynamic binding, the following code fragment should evaluate to 1:

```
reset
  (fn _ =>
      (shift (fn k => (k 0)
                      handle Fail _ => 1))
    + (raise Fail "uncaught"))
handle Fail _ => 2
```

Instead, the Filinski implementation returns 2. The problem occurs when the `Fail` exception is raised in the original context of the `reset` expression; the raised exception blunders right past the border and is consequently caught by the `handle Fail _ => 2` guard.

So much for an escape route.

## 5. The Great Escape

The problem with the invariant of Section 3.2 is that it does not prevent exceptions from crossing a border; it only takes ordinary returns into account. The simulation invariant should guarantee that *control never crosses the border*, whether by returning or raising an exception. Instead, when control reaches the border, it should always be redirected to the metacontinuation.

More generally, the problem is that we need some way of splicing together the dynamic bindings of two different continuations. Invoking a continuation installs the local exception handlers from the captured continuation in front of the global exception handlers of the metacontinuation. But Standard ML does not provide primitives for cutting and pasting dynamically bound exception handlers.

The solution is to install a dynamic barrier, in the form of a universal exception handler $\mathsf{handle\ x}$, that effectively blocks any handlers beyond the border. When the barrier catches an exception, it redirects it to the metacontinuation to raise it again. This creates the "tunnel" through which both returned values and raised exceptions alike can cross the border without getting caught by the wrong handlers.

The metacontinuation needs to be able to distinguish ordinary returns from raised exceptions, so every tunneled value is tagged as either a return or an exception:

```
datatype tunneled = SUCCESS of ans
                  | FAILURE of exn
```

Any answer sent to the metacontinuation must be wrapped as a `SUCCESS`, and any exception redirected to the metacontinuation is wrapped as a `FAILURE`. To complete the protocol, the metacontinuation must always unwrap the tunneled value and either return or re-raise it.

We call this trick The Great Escape.

### 5.1 The new invariant

To keep things simple, we can model the `tunneled` datatype with a standard functional representation rather than adding sum types to the model. A `tunneled` value in the model is a higher-order function that consumes two function arguments `s` and `f`. A `SUCCESS` always applies the `s` argument to its answer value, and a `FAILURE` applies the `f` argument to its exception.

The new encoding in the model ML wraps both locally returned values and locally uncaught exceptions as tunneled data, and the metacontinuation unwraps the tunneled data by returning values and re-raising exceptions.

$$
\begin{array}{rcl}
[\![\mathcal{S}k.e]\!] & = & \mathcal{C}\mathsf{k}'.\mathsf{let\ x} \leftarrow (\mathsf{let\ k} \leftarrow \lambda\mathsf{x}.[\![\#(k'\ x)]\!]\ \mathsf{in}\ [\![e]\!])\ \mathsf{in} \\
& & \qquad ((\uparrow)\ \lambda\mathsf{s}.\lambda\mathsf{f}.(\mathsf{s\ x})) \\
& & \quad \mathsf{handle\ x} \Rightarrow ((\uparrow)\ \lambda\mathsf{s}.\lambda\mathsf{f}.(\mathsf{f\ x})) \\
& & \mathsf{k}' \notin FV([\![e]\!]) \\
[\![\#e]\!] & = & (((\mathcal{C}\mathsf{k}.(\mathsf{let\ m} \leftarrow (\uparrow)\ \mathsf{in} \\
& & \qquad \downarrow\lambda x.(\downarrow\mathsf{m};\mathsf{k\ x}); \\
& & \quad \mathsf{let\ x} \leftarrow [\![e]\!]\ \mathsf{in}\ ((\uparrow)\ \lambda\mathsf{s}.\lambda\mathsf{f}.(\mathsf{s\ x})) \\
& & \quad \mathsf{handle\ x} \Rightarrow ((\uparrow)\ \lambda\mathsf{s}.\lambda\mathsf{f}.(\mathsf{f\ x})))) \\
& & \ \lambda\mathsf{x}.\mathsf{x}) \\
& & \ \mathsf{raise}) \\
& & \mathsf{k} \notin FV([\![e]\!])
\end{array}
$$

Figure 4 shows the new and updated rules for the new simulation invariant. The simple new expression forms relate by components. The implementations of `shift` and `reset` now wrap as a tunneled value either the local result or any locally uncaught exception. A border separating the local portion of a continuation now guards both ordinary return and exceptional return.

**Theorem 2 (simulation, The Great Escape)**
*With the languages of Sections 4.1 and 4.2 and the simulation invariant of Section 5.1, if* $\mathsf{C} \sim C$ *and* $C \xrightarrow{\mathcal{S}} C'$ *and then there exists a term* $\mathsf{C}'$ *such that* $\mathsf{C} \xrightarrow{\mathcal{S}}{}^{*} \mathsf{C}'$ *and* $\mathsf{C}' \sim C'$.

*Proof* As before, using the new simulation invariant. □

### 5.2 Space efficiency

The new translation is operationally correct, but for practical purposes, it suffers terrible space efficiency. Users do not expect the continuation captured by `shift` to consume as much space as a full continuation captured by `callcc`.

In fact, every time the simulation captures a delimited continuation, the simulation invariant guarantees that control will never pass the border. Of course, the SML/NJ garbage collector cannot infer this information about our program invariant. The problem, then, is that continuation frames that should be dead remain live in memory for too long.

Happily, SML/NJ provides the following useful abstraction in the `SMLofNJ.Cont` library:

```
val isolate : ('a -> unit) -> 'a cont
```

The `isolate` function consumes a function that does not return and converts it into a first-class continuation. Invoking this continuation abandons the current continuation and replaces it with the isolated function.

$$\frac{}{\mathsf{raise} \sim \mathbf{raise}}\ [\text{I-RAISE}] \qquad \frac{}{\varepsilon \sim \varepsilon}\ [\text{I-EXCEPTION}]$$

$$\frac{\mathsf{e} \sim e \qquad \mathsf{e}' \sim e'}{\mathsf{e\ handle\ h} \Rightarrow \mathsf{e}' \sim e\ \mathbf{handle}\ h \Rightarrow e'}\ [\text{I-HANDLE}]$$

$$\frac{\mathsf{k}' \notin FV\mathsf{e} \qquad \mathsf{e} \sim e \qquad \mathsf{e}' \sim \#(k\ x)}{\begin{array}{l}\mathcal{C}\mathsf{k}'.\mathsf{let\ x} \leftarrow (\mathsf{let\ k} \leftarrow \lambda\mathsf{x}.\mathsf{e}'\ \mathsf{in\ e})\ \mathsf{in} \sim \mathcal{S}k.e \\ \qquad ((\uparrow)\ \lambda s.\lambda f.(s\ x)) \\ \quad \mathsf{handle\ x} \Rightarrow ((\uparrow)\ \lambda s.\lambda f.(f\ x))\end{array}}\ [\text{I-SHIFT*}]$$

$$\frac{\mathsf{k} \notin FV(\mathsf{e}) \qquad \mathsf{e} \sim e}{\begin{array}{l}(((\mathcal{C}\mathsf{k}.(\mathsf{let\ m} \leftarrow (\uparrow)\ \mathsf{in} \qquad \sim \#e \\ \qquad \downarrow\lambda x.(\downarrow\mathsf{m};\mathsf{k\ x}); \\ \quad \mathsf{let\ x} \leftarrow \mathsf{e\ in}\ ((\uparrow)\ \lambda s.\lambda f.(\mathsf{s\ x})) \\ \quad \mathsf{handle\ x} \Rightarrow ((\uparrow)\ \lambda s.\lambda f.(\mathsf{f\ x})))) \\ \ \lambda\mathsf{x}.\mathsf{x}) \\ \ \mathsf{raise})\end{array}}\ [\text{I-RESET*}]$$

$$\frac{\mathsf{E} \approx E \qquad \text{any } \mathsf{E}'}{\mathsf{E}'[\mathsf{let\ x} \leftarrow \mathsf{E\ in}\ ((\uparrow)\ \lambda s.\lambda f.(\mathsf{s\ x})) \sim E \\ \quad \mathsf{handle\ x} \Rightarrow ((\uparrow)\ \lambda s.\lambda f.(\mathsf{f\ x}))]}\ [\text{I-BORDER*}]$$

$$\frac{\mathsf{E} \approx E \qquad \mathsf{e} \sim e}{\mathsf{handle\ h} \Rightarrow \mathsf{e} \approx E\ \mathbf{handle}\ h \Rightarrow e}\ [\text{I-INSTALL}]$$

**Figure 4.** New and changed (*) rules for the simulation invariant.

This is just what we need to inform the runtime to abandon the useless portion of the current continuation whenever we capture a delimited continuation:

```
fun abort thunk = throw (isolate thunk) ()
```

This function effectively performs a computation in the empty continuation.

As it turns out, every use of `escape` in the simulation sends its results to the metacontinuation, so we *never* need the current continuation after capturing it. This means we can change the definition of `escape` to abandon the continuation with `abort` after capturing it. This corresponds to an alternative semantics for `escape`:

$$\mathsf{E}[\mathcal{C}\mathsf{x.e}],\ \mathsf{v} \quad \xrightarrow{\mathcal{C}} \quad \mathsf{e}[\langle\mathsf{E}\rangle/\mathsf{x}],\ \mathsf{v}$$

Unsurprisingly, the simulation invariant holds for this alternative semantics as well; the only difference is that there is never extra "junk" in a captured continuation.

**Theorem 3 (simulation, The Great Escape, optimized)**
*With the languages of Sections 4.2 and 5.2 and the simulation invariant of Section 5.1, if* $\mathsf{C} \sim C$ *and* $C \xrightarrow{\mathcal{S}} C'$*, then there exists a term* $\mathsf{C}'$ *such that* $\mathsf{C} \xrightarrow{\mathcal{C}}{}^{*} \mathsf{C}'$ *and* $\mathsf{C}' \sim C'$.

### 5.3 Implementation

Figure 5 shows the new implementation in SML/NJ.

## 6. Conclusion

We have shown that, with a proper exception-handling protocol, it is possible to express delimited continuations with undelimited continuations even in the presence of exceptions.

While the general problem of splicing together dynamic bindings from two contexts is hard to achieve without additional primitives in the implementation language, our encoding demonstrates that exception handlers can be "cut" with a universal exception handler and "pasted" back together with an appropriate return protocol. Pleasingly, this implementation fits the feature set of Standard ML of New Jersey well, giving us a non-native implementation of `shift` and `reset` with the correct semantics and appropriate space behavior.

## Acknowledgments

I thank Jacob Matthews and Ryan Culpepper for the brainstorming that led to this work. I thank Mitchell Wand for his guidance and Matthias Felleisen and the anonymous reviewers for their thorough and helpful comments.

## References

Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006. ISSN 0167-6423.

Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, 1989.

Matthias Felleisen. *The calculi of $\lambda_v$-CS conversion: A syntactic theory of control and state in imperative higher-order languages*. PhD thesis, 1987.

Matthias Felleisen. The theory and practice of first-class prompts. In *Symposium on Principles of Programming Languages*, 1988.

Andrzej Filinski. Representing monads. In *Principles of Programming Languages (POPL)*, pages 446–457, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-636-0.

Chris Hanson. MIT scheme reference manual. Technical Report AITR-1281, 1991.

Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):582–598, 1987. ISSN 0164-0925.

Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised$^5$ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 1998.

Oleg Kiselyov. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Indiana University, March 2005.

Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited dynamic binding. In *International Conference on Functional Programming*, 2006.

Jeffrey R. Lewis, Mark Shields, John Launchbury, and Erik Meijer. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages*, 2000.

Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, 1990. ISBN 0-262-63132-6.

Luc Moreau. A Syntactic Theory of Dynamic Binding. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT/FASE'97)*, volume 1214, pages 727–741, Lille, France, April 1997. Springer-Verlag.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, August 1972.

Chung-Chieh Shan. Shift to control. In *Scheme Workshop*, September 2004.

Dorai Sitaram. *Models of Control and Their Implications for Programming Language Design*. PhD thesis, 1994.

Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computing*, 3(1), 1990.

Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-Order and Symbolic Computation*, 14(2):141–160, 2002.

Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *LISP and functional programming*, 1986.

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

functor Control (type ans) : CONTROL =
struct
    open Escape
    exception MissingReset
    type ans = ans

    datatype tunneled = SUCCESS of ans
                      | FAILURE of exn

    val mk : (tunneled -> void) ref = ref (fn _ => raise MissingReset)

    fun return x = coerce (!mk x)

    fun reset thunk =
        (case escape (fn k => let val m = !mk
                              in
                                  mk := (fn r => (mk := m; k r));
                                  return (SUCCESS (thunk ())
                                          handle x => FAILURE x)
                              end) of
            SUCCESS v => v
          | FAILURE x => raise x)

    fun shift f =
        escape (fn k => return (SUCCESS (f (fn v => reset (fn () => coerce (k v))))
                                handle x => FAILURE x))
end;
```

**Figure 5.** The Great Escape in Standard ML of New Jersey.