A robust implementation of delimited control

Zena M. Ariola
University of Oregon
ariola@cs.uoregon.edu

Hugo Herbelin
INRIA-Futurs
Hugo.Herbelin@inria.fr

David Herman
Mozilla Research
dherman@mozilla.com

Daniel Keith
University of Oregon
dkeith@cs.uoregon.edu

Abstract

Filinski provides an encoding of *shift* and *reset* in terms of undelimited abortive control and state in the context of a call-by-value language. The correctness of the encoding is shown with respect to a continuation-passing style semantics and it relies on the program being surrounded by a top-level reset. We present an alternative correctness proof based on an operational semantics of *shift* and *reset* which, in the presence of a top-level reset, is sound and complete with respect to the continuation-passing style semantics. We then revisit the encoding and identify the cause of a space leak as a lack of preservation of strictness. In the encoding of shift, a non-strict invocation of the top-level continuation is made strict. We propose a new encoding and prove its correctness. The new encoding eliminates the space leak and provides more robust error handling. In particular, the correctness of the new encoding does not require the program to be surrounded by a top-level reset.

1 Introduction

In his seminal paper, Filinski [6] showed that the delimited control operators *shift* and *reset* are complete with respect to monadic effects and can be represented in a call-by-value language such as SML in terms of the abortive control operator `callcc` and one mutable variable. The correctness of the encoding is with respect to a continuation passing style semantics. In this paper, we show the correctness of the encoding based on an operational approach, which reveals some limitations in the encoding. In addition to suffering from a well-known space leak, Filinski's encoding relies on the program being surrounded by a top-level reset. Without a top-level reset, the encoding behaves erratically—formally, it produces a value which might be different from the one produced by Kameyama and Hasegawa axiomatization of *shift* and *reset* [11]. Enforcing this invariant operationally would require access to the top-level continuation, which SML does not provide.

We present an alternative encoding which provides better error handling and memory use and show its correctness. The issue with the original encoding is that strictness is not preserved. In an expression of the form

`shift (fn k =>` e`)`

there is an implicit invocation of the top-level continuation which is non-strict: a jump to the top-level is performed before evaluating the argument. In the encoding, that invocation is made explicit through the presence of the abort operator, which however comes with a strict semantics. A similar problem occurs in the encoding of `callcc` in terms of $\mathscr{C}$ [8]. In an expression of the form

`callcc (fn k =>` e`)`

there is an implicit occurrence of `k` just before e. That occurrence is non-strict, that is, it can be called with an expression instead of a value. In the encoding, that occurrence becomes strict.

We start with a review of abortive and composable continuations in Sections 2 and 3. In Section 4, we present Filinski's encoding of composable control in terms of (undelimited) abortive control and state and discuss its limitations. In Section 5, we present a new encoding and its correctness. In Section 6, we present a new encoding of `callcc` in terms of $\mathscr{C}$. We conclude in Section 7.

2 Abortive control

We start with a call-by-value λ-calculus whose syntax and operational semantics are described below:

$$\begin{array}{lllll} \textit{Expressions} & e & ::= & v \mid e\ e \\ \textit{Values} & v & ::= & \lambda x.e \mid x \\ \textit{Evaluation contexts} & E & ::= & [\,] \mid E\ e \mid v\ E \end{array}$$

$$E[(\lambda x.e)\ v] \longmapsto E[e[v/x]] \tag{1}$$

(As usual, $\longmapsto\!\!\!\!\!\rightarrow$ stands for the reflexive and transitive closure of $\longmapsto$.) Languages such as SML and Scheme extend λ-calculus with primitives for modifying the flow of control. An example is `callcc`, which reifies the *continuation* of an expression, i.e., the remaining work to be done after evaluating the expression, as a function. Consider the expression $(1+2)+(5+4)$; assuming a left-to-right order of evaluation, the continuation of the subexpression $5+4$ is the function $\lambda x.3+x$. We can give this function a name by writing $(1+2)+\texttt{callcc}(\lambda k.5+4)$. In general, the evaluation of the expression $\texttt{callcc}(\lambda k.e)$ binds the current continuation to k before evaluating e.

A continuation bound by `callcc` in fact differs from ordinary functions: once it is called, control does not return to the caller. We refer to this kind of value as an *abortive continuation*. Traditionally, abortive behavior is specified operationally with an abort operator, written $\mathscr{A}$. For example, the continuation in our example term would be written $\lambda x.\mathscr{A}\ (3+x)$ rather than $\lambda x.3+x$. This semantics is captured formally by the following two rules:

$$\begin{array}{lll} E[\texttt{callcc}(\lambda k.e)] & \longmapsto & E[e[\lambda x.\mathscr{A}\ E[x]/k]] \\ E[\mathscr{A}\ e] & \longmapsto & e \end{array}$$

Example 2.1. Consider the following reduction:

$$\begin{array}{lll} & (1+2)+\texttt{callcc}(\lambda k.4+(k\ 2)) & (E \equiv 3+[\,]) \\ \longmapsto\!\!\!\!\!\rightarrow & 3+((4+(k\ 2))[\lambda x\mathscr{A}\ (3+x)/k]) & \\ \equiv & 3+(4+((\lambda x.\mathscr{A}\ (3+x))\ 2)) & \\ \longmapsto & 3+(4+\mathscr{A}\ (3+2)) & (E \equiv 3+(4+[\,])) \\ \longmapsto & 3+2 & \end{array}$$

Notice how when continuation k is invoked the context $3+(4+[\,])$ is abandoned.

To better distinguish between the abort traditionally present in the operational rules of control operators from the abort present in a program, we adopt a different notation for continuations. We write $\lambda x.\mathscr{A}\ E[x]$ as $\langle E\rangle$. We rephrase the operational semantics of `callcc` as follows:

$$\begin{array}{lll} E[\texttt{callcc}(\lambda k.e)] & \longmapsto & E[e[\langle E\rangle/k]] \\ E[\langle E'\rangle\ v] & \longmapsto & E'[v] \end{array}$$

Another example of an abortive operator is $\mathscr{C}$ [4]. Given an expression of the form $\mathscr{C}(\lambda k.e)$, after binding k to the continuation, control does not return to the surrounding context but returns to the top-level:

$$E[\mathscr{C}(\lambda k.e)] \longmapsto e[\langle E\rangle/k]$$

For example:

$$(1+2)+\mathscr{C}(\lambda k.5+4) \longmapsto\!\!\!\!\!\rightarrow 5+4\ .$$

The abort operator $\mathscr{A}$ is expressible in terms of $\mathscr{C}$:[1]

$$\mathscr{A}\ e = \mathscr{C}(\lambda \_.e)$$

[1] We use _ for a variable which does not occur free in the body of an abstraction.

3 Composable control

Danvy and Filinski [3] and Felleisen [4] proposed the *prompt* or *reset* operator, written #, to *delimit* the context captured by a continuation. We define the operator $\mathscr{K}$ as the analogue to `callcc` in the presence of a prompt. For example, in the expression:

$$2+\#(1+\mathscr{K}(\lambda k.e))$$

the continuation k stands for the context $\langle 1+[]\rangle$ instead of $\langle 2+\#(1+[])\rangle$. The delimited expression $\mathscr{C}(\lambda k.e)$ becomes $\mathscr{K}(\lambda k.\mathscr{A}\ e)$. The syntax and semantics of a call-by-value lambda-calculus extended with delimited control expressions (referred to as $\lambda_{\mathscr{K}\mathscr{A}\#}$) are as follows:

$$\begin{array}{lllll}
\text{Expression} & e & ::= & v \mid e\ e \mid \mathscr{K}(\lambda x.e) \mid \mathscr{A}\ e \mid \#e \\
\text{Value} & v & ::= & x \mid \lambda x.e \mid \langle E\rangle \\
\text{Context} & F & ::= & [] \mid F\ e \mid v\ F \mid \#F \\
\text{Delimited Context} & E & ::= & [] \mid E\ e \mid v\ E
\end{array}$$

$$\begin{array}{lcl}
F[\#E[\mathscr{K}(\lambda k.e)]] & \longmapsto & F[\#E[e[\langle E\rangle/k]]] \\
F[\#E[\mathscr{A}\ e]] & \longmapsto & F[\#e] \\
F[\#E[\langle E'\rangle\ v]] & \longmapsto & F[\#[E'[v]]] \\
F[\#v] & \longmapsto & F[v]
\end{array}$$

Unlike functions, abortive continuations are not composable: for example, the expression $2+\#(1+\mathscr{K}(\lambda k.(k\ (k\ 2))))$ evaluates to 5 rather than 6. By contrast, *composable continuations* [4, 3], which we denote $\langle\!\langle E\rangle\!\rangle$, are functional representations of continuations that do not abort. The shift expression, written $\mathscr{S}(\lambda x.e)$, is analogous to a $\mathscr{C}$-expression but captures a composable continuation. The syntax and semantics of a call-by-value lambda-calculus extended with shift and reset (referred to as $\lambda_{\mathscr{S}\#}$) is as follows:

$$\begin{array}{lllll}
\text{Expression} & e & ::= & v \mid e\ e \mid \mathscr{S}(\lambda x.e) \mid \#e \\
\text{Value} & v & ::= & x \mid \lambda x.e \mid \langle\!\langle E\rangle\!\rangle
\end{array}$$

The definitions of contexts E and F are as before.

$$\begin{array}{lcl}
F[\#E[\mathscr{S}(\lambda k.e)]] & \longmapsto & F[\#(e[\langle\!\langle E\rangle\!\rangle/k])] \\
F[\langle\!\langle E\rangle\!\rangle\ v] & \longmapsto & F[\#E[v]] \\
F[\#v] & \longmapsto & F[v]
\end{array}$$

Example 3.1.

$$\begin{array}{ll}
 & 2+\#(1+\mathscr{S}(\lambda k.k\ (k\ 2))) \\
\longmapsto & 2+\#(\langle\!\langle 1+[]\rangle\!\rangle((\langle\!\langle 1+[]\rangle\!\rangle\ 2))) \\
\longmapsto & 2+\#(\langle\!\langle 1+[]\rangle\!\rangle((\#(1+2)))) \\
\longmapsto\!\!\!\!\rightarrow & 2+4
\end{array}$$

In the following, we are going to show the correctness of Filinski's encoding of shift and reset with respect to the above operational semantics. Therefore, our first goal is to relate this semantics to the continuation-passing style specification of shift and reset. We make use of Kameyama and Hasegawa axiomatization (written as $=_{KH}$) of shift and reset, which is shown sound and complete with respect to the continuation-passing style semantics [11]. We write e^{kh} for the expression obtained by writing $\#e$ as $\langle e\rangle$ and $\langle\!\langle E\rangle\!\rangle$ as $\lambda x.\langle E[x]\rangle$.

Theorem 3.2. *Given a $\lambda_{\mathscr{S}\#}$ expression e:*

1. *If $e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#} v$ then $e^{kh} =_{KH} v^{kh}$.*

2. *If* $e^{kh} =_{KH} v$ *then* $\exists v'$*, such that* $\#e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#} v'$.

Notice that in order to reach completeness the expression needs to be surrounded by a top-level reset. This is due to the presence of the $\mathscr{S}$*-elim* axiom:

$$\mathscr{S}(\lambda k.k\ e) = e, \text{ if } k \text{ does not occur free in } e \ .$$

Whereas $\mathscr{S}(\lambda k.k\ 1) =_{KH} 1$, the expression $\mathscr{S}(\lambda k.k\ 1)$ is "stuck" according to the operational semantics. However, if a program is surrounded by a reset then the $\mathscr{S}$*-elim* axiom is not needed, and completeness holds.

4 Composable control as abortive control and state

We now turn our attention to Filinski's encoding of shift in terms of (undelimited) abortive control and state [6]. We present this encoding in two steps. We start with the encoding of shift in terms of $\mathscr{K}$, # and (delimited) $\mathscr{A}$. Next, we encode $\mathscr{K}$ in terms of its undelimited counterpart `callcc` and implement # and $\mathscr{A}$ by respectively updating and reading a mutable cell.

4.1 Composable control as (delimited) abortive control and prompt

Operationally, the difference between $\mathscr{S}$ and (delimited) $\mathscr{C}$ occurs when a continuation is invoked: a composable continuation behaves like an abortive continuation with a prompt surrounding the application. Thus, we can simulate composable continuations by surrounding each invocation of a continuation with a prompt, indicating that control must return to the caller. This relies on the dynamic nature of prompts [2].

Example 4.1. Wrapping k with a prompt allows it to be composed:

$$\begin{array}{ll} & 2+\#(1+\mathscr{K}(\lambda k.k\ (\#(k\ 2)))) \\ \longmapsto & 2+\#(1+(\langle 1+[]\rangle\ (\#(\langle 1+[]\rangle\ 2)))) \\ \longmapsto & 2+\#(1+(\langle 1+[]\rangle\ (\#(1+2)))) \\ \longmapsto\!\!\!\!\rightarrow & 2+4 \end{array}$$

Formally, $\mathscr{S}$ can be expressed in terms of (delimited) $\mathscr{C}$ and # [17, 2]:

$$\mathscr{S}(\lambda k.e) = \mathscr{C}(\lambda k.e[\lambda x.\#(k\ x)/k]))$$

or equivalently in terms of $\mathscr{K}$, # and (delimited) abort [14]:

$$\mathscr{S}(\lambda k.e) = \mathscr{K}(\lambda k.\mathscr{A}\ e[\lambda x.\#(k\ x)/k]) \tag{2}$$

The correctness of the encoding is expressed next, where $\lceil e \rceil$ denotes the encoding of a $\lambda_{\mathscr{S}\#}$ expression into a $\lambda_{\mathscr{K}\mathscr{A}\#}$ expression. The composable continuation $\langle\!\langle E \rangle\!\rangle$ is encoded as $\lambda x.\#(\langle E \rangle\ x)$.

Lemma 4.2. *Given a* $\lambda_{\mathscr{S}\#}$ *expression* e*:*

1. *If* $e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#} v$ *then* $\exists v', \lceil e \rceil \longmapsto\!\!\!\!\rightarrow_{\mathscr{K}\mathscr{A}\#} v'$*;*

2. *If* $\lceil e \rceil \longmapsto\!\!\!\!\rightarrow_{\mathscr{K}\mathscr{A}\#} v$ *then* $\exists v', e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#} v'$.

4.2 Prompt as abortive control and state

To model the prompt a global variable called *mk* is introduced. A reset expression $\#e$ corresponds to updating *mk*. The abort expression $\mathscr{A}e$ reads *mk* after the evaluation of *e*. In other words, the abort expression is made strict. The delimited control operator $\mathscr{K}$ becomes the undelimited control operator `callcc`. Since Filinski's encoding of a prompt and abort expression introduce several intermediate steps, we opt for hiding those steps under more complex operational rules and leave the prompt and a strict abort in the target language. We call this language $\lambda_{\texttt{callcc}\mathscr{A}_s\#!}$, its syntax is:

Expression	e	::=	$v \mid e\ e \mid \texttt{callcc}(\lambda x.e) \mid \mathscr{A}_s\ e \mid \#e \mid mk := w \mid \texttt{MissingReset}$
Value	v	::=	$x \mid \lambda x.e \mid \langle E \rangle$
Metacontinuation	w	::=	$\lambda\_.\texttt{MissingReset} \mid \lambda x.(mk := w; \langle E \rangle\ x)$
Configuration	c	::=	$e, w \mid \texttt{uncaught:MissingReset}, w$
Evaluation Context	E	::=	$[\,] \mid E\ e \mid v\ E \mid \mathscr{A}_s\ E$

In the following, $e; e'$ stands for $(\lambda\_.e')\ e$:

$$
\begin{array}{lcl}
E[(\lambda x.e)\ v], w & \longmapsto & E[e[v/x]], w \\
E[\#e], w & \longmapsto & E[\mathscr{A}_s\ e], \lambda x.(mk := w; \langle E \rangle\ x) \\
E[\texttt{callcc}(\lambda k.e)], w & \longmapsto & E[e[\langle E \rangle / k]], w \\
E[\langle E' \rangle\ v], w & \longmapsto & E'[v], w \\
E[\mathscr{A}_s\ v], w & \longmapsto & E[w\ v], w \\
E[mk := w'], w & \longmapsto & E[w'], w' \\
E[\texttt{MissingReset}], w & \longmapsto & \texttt{uncaught:MissingReset}, w
\end{array}
$$

The encoding of a $\lambda_{\mathscr{K}\mathscr{A}\#}$ expression into a target language expression, written as $\lceil\cdot\rceil$, simply corresponds to replacing each occurrence of $\mathscr{K}$ and $\mathscr{A}$ with `callcc` and $\mathscr{A}_s$, respectively. Variable *mk* is initialized to a function which always faults. We model this by embedding an expression *e* into the following configuration:

$$\lceil e \rceil, w_i$$

where w_i stands for $\lambda\_.\texttt{MissingReset}$.

Lemma 4.3. *Given a $\lambda_{\mathscr{K}\mathscr{A}\#}$ expression e:*

1. *If $e \longmapsto\!\!\!\!\rightarrow_{\mathscr{K}\mathscr{A}\#} v$ then $\exists v', \lceil e \rceil, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!} v', w_i$;*
2. *If $\lceil \#e \rceil, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!} v, w_i$ then $\exists v', \#e \longmapsto\!\!\!\!\rightarrow_{\mathscr{K}\mathscr{A}\#} v'$.*

We combine the two encodings and write $\lceil\!\lceil e \rceil\!\rceil$ for the encoding of a $\lambda_{\mathscr{S}\#}$ expression into the language with `callcc` and state.

Theorem 4.4. *Given a $\lambda_{\mathscr{S}\#}$ expression e:*

1. *If $e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#} v$ then $\exists v', \lceil\!\lceil e \rceil\!\rceil, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!} v', w_i$;*
2. *If $\lceil\!\lceil \#e \rceil\!\rceil, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!} v, w_i$ then $\exists v', \#e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#} v'$.*

4.3 Space leak

Let us now revisit the encoding with respect to intensional properties such as space. Consider the following definition:

$$\begin{array}{lcl} \mathit{loop}\ 1 & = & 1 \\ \mathit{loop}\ n & = & \mathscr{S}(\lambda k.\mathit{loop}\,(n\text{-}1)) \end{array}$$

According to the operational semantics of $\mathscr{S}$, the execution of $\#(\mathit{loop}\ 3)$ always occurs at a bounded distance from the root:

$$\#(\mathit{loop}\ 3) \longmapsto \#.\mathscr{S}(\lambda k.\mathit{loop}\ (3\text{-}1)) \longmapsto \#(\mathit{loop}\ (3\text{-}1)) \longmapsto\!\!\!\!\rightarrow \#(\mathit{loop}\ (2\text{-}1))$$

The same happens for its encoding in $\lambda_{\mathscr{K}\mathscr{A}\#}$:

$$\#(\mathit{loop}\ 3) \longmapsto \#.\mathscr{K}(\lambda k.\mathscr{A}\ (\mathit{loop}\ (3\text{-}1))) \longmapsto \#.\mathscr{A}\ (\mathit{loop}\ (3\text{-}1)) \longmapsto \#\mathit{loop}\ (3\text{-}1) \longmapsto\!\!\!\!\rightarrow \#\mathit{loop}\ (2\text{-}1)$$

However, the translation to $\lambda_{\texttt{callcc}\mathscr{A}_s\#!}$ does not preserve the property of reduction at a bounded distance from the top. We underline the redex perfomed, unless it is the top redex::

$$\begin{array}{ll} \#(\mathit{loop}\ 3), w_i & \longmapsto \\ \mathscr{A}_s(\underline{\mathit{loop}\ 3}), \lambda x.mk := w_i; \langle[]\rangle x & \longmapsto \\ \mathscr{A}_s(\underline{\texttt{callcc}(\lambda k.\mathscr{A}_s(\mathit{loop}(3\text{-}1)))}), \lambda x.mk := w_i; \langle[]\rangle x & \longmapsto \\ \mathscr{A}_s(\mathscr{A}_s(\mathit{loop}\underline{(3\text{-}1)})), \lambda x.mk := w_i; \langle[]\rangle x & \longmapsto\!\!\!\!\rightarrow \\ \mathscr{A}_s(\mathscr{A}_s(\mathscr{A}_s(\mathit{loop}\underline{(2\text{-}1)}))), \lambda x.mk := w_i; \langle[]\rangle x & \end{array}$$

In other words, the SML implementation leads to a space leak. A term of the form $\mathscr{S}(\lambda k.e)$ implicitly invokes the continuation corresponding to the nearest prompt. In the first step of the encoding , that implicit invocation to the top-level manifests itself in the $\mathscr{A}$ operator. Whereas other occurrences of the continuations are strict, evaluating their argument before performing the jump, the invocation of the top-level continuation follows a non-strict discipline. However, that property is not preserved by the second step of the encoding since the $\mathscr{A}$ operator is made strict.

4.4 Robustness of the encoding

We have shown correctness of the encoding with respect to evaluation to values. We now discuss the behavior of the encoding when the top-level reset is missing. Since the metacontinuation captured in *mk* always produces an error when invoked, we were lead to believe that a shift expression with an undefined top-level would raise an error, and indeed running the following example with the actual implementation in SML supported our belief:

Example 4.5.
```
shift (fn k => 99);
uncaught exception MissingReset
```

We were therefore surprised to observe the following result:

Example 4.6.
```
shift (fn k => (k 1) + 3);
val it = 1 : int
```

It was not clear to us why the above term did not produce an error. Even more surprisingly, we continued with the re-evaluation of our previous term:

Example 4.7.
```
shift (fn k => 99);
Error: throw from one top-level expression into another
```
which now exhibited different behavior than before.

To be clear, these behaviors do not contradict the correctness of Filinski's encoding. According to Filinski's type and effect system [7], the programs in question are ill-effect-typed, and therefore the implementation is unconstrained. We were nevertheless surprised by this behavior, which we can now explain.

Example 4.8. We show the execution of Example 4.5 :

$$\begin{array}{rl} & \llbracket \mathscr{S}(\lambda k.99) \rrbracket, \lambda\_.\texttt{MissingReset} \\ & \lceil \mathscr{K}(\lambda k.\mathscr{A}\ 99) \rceil, \lambda\_.\texttt{MissingReset} \\ = & \texttt{callcc}(\lambda k.\mathscr{A}_s\ 99), \lambda\_.\texttt{MissingReset} \\ \longmapsto\!\!\!\!\!\rightarrow & (\lambda\_.\texttt{MissingReset})\ 99, \lambda\_.\texttt{MissingReset} \\ \longmapsto\!\!\!\!\!\rightarrow & \texttt{uncaught: MissingReset}, \lambda\_.\texttt{MissingReset} \end{array}$$

Now consider the term $\mathscr{S}(\lambda k.(k\ 1)+3)$ of Example 4.6. Consistent with the previous example, we were expecting the encoding to produce an error. Surprisingly, the result is 1.

$$\begin{array}{rl} & \llbracket \mathscr{S}(\lambda k.(k\ 1)+3) \rrbracket, \lambda\_.\texttt{MissingReset} \\ = & \texttt{callcc}(\lambda k.\mathscr{A}_s\ (((\lambda x.\#(k\ x))\ 1)+3)), \lambda\_.\texttt{MissingReset} \\ \longmapsto\!\!\!\!\!\rightarrow & \mathscr{A}_s\ (((\lambda x.\#(\langle [] \rangle\ x))\ 1)+3), \lambda\_.\texttt{MissingReset} \\ \longmapsto & \mathscr{A}_s\ (\#(\langle [] \rangle\ 1)+3), \lambda\_.\texttt{MissingReset} \\ \longmapsto & \mathscr{A}_s\ (\mathscr{A}_s\ (\langle [] \rangle\ 1)+3), \lambda x.(mk := w_i; \langle \mathscr{A}_s\ ([]+3) \rangle\ x) \\ \longmapsto & 1, \lambda x.(mk := w_i; \langle \mathscr{A}_s\ ([]+3) \rangle\ x) \end{array}$$

Notice that a useful invariant does not hold, namely that at the end of the execution, the variable mk should contain its initial value. We continue with the expression of Example 4.7:

$$\begin{array}{rl} & \llbracket \mathscr{S}(\lambda k.99) \rrbracket, \lambda x.(mk := w_i; \langle \mathscr{A}_s\ ([]+3) \rangle\ x) \\ = & \texttt{callcc}(\lambda k.\mathscr{A}_s 99), \lambda x.(mk := w_i; \langle \mathscr{A}_s\ ([]+3) \rangle\ x) \\ \longmapsto\!\!\!\!\!\rightarrow & (\lambda x.(mk := w_i; \langle \mathscr{A}_s\ ([]+3) \rangle\ x))\ 99, \lambda x.(mk := w_i; \langle \mathscr{A}_s\ ([]+3) \rangle\ x) \\ \longmapsto\!\!\!\!\!\rightarrow & \langle \mathscr{A}_s\ ([]+3) \rangle\ 99, w_i \end{array}$$

Because SML's interactive shell disallows returning to old interactions, the above configuration causes the error seen in Example 4.7.

Since using *shift* without a top-level *reset* is a client error, Filinski's specification imposes no requirements on the behavior of these examples. Indeed, these programs would be rejected as ill-typed in his framework. However, in an implementation language such as SML, the type system does not guarantee the presence of a top-level *reset*. A robust implementation should therefore defend against such invalid uses of *shift* by raising an error.

5 An alternative encoding

Since SML does not have access to the top-level continuation, we force the implementation to raise an error when that is not explicitly set. To that end, we add the following operational rules to the semantics of $\lambda_{\mathscr{K}\mathscr{A}\#}$:

$$\begin{array}{lcl} E[\mathscr{K}(\lambda k.e)] & \longmapsto & \texttt{uncaught:MissingReset} \\ E[\langle E' \rangle\ v] & \longmapsto & \texttt{uncaught:MissingReset} \\ E[\mathscr{A}\ e]] & \longmapsto & \texttt{uncaught:MissingReset} \end{array}$$

Similarly, we add the following rule to the semantics of $\lambda_{\mathscr{S}\#}$:

$$E[\mathscr{S}(\lambda k.e)] \longmapsto \texttt{uncaught:MissingReset}$$

The new operational semantics of shift and reset loses soundness and completeness with respect to Kameyama and Hasegawa axiomatization since, according to the operational semantics, $\mathscr{S}(\lambda k.1)$ would raise an error but return 1 in Kameyama and Hasegawa axiomatization. To validate the $\mathscr{S}$*-elim* axiom the implementation would need access to the top-level continuation. Since this is not possible in SML, we opt to sacrifice the axiom and work instead with Kameyama and Hasegawa axiomatization without the $\mathscr{S}$*-elim* axiom, written as KH_{-}. The correctness of the new operational semantics becomes:

Theorem 5.1. *Given a $\lambda_{\mathscr{S}\#}$ expression e:*

1. *If $e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#} v$ then $e^{kh} =_{KH_{-}} v^{kh}$.*
2. *If $e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#}$ `uncaught:MissingReset` then $e^{kh} =_{KH_{-}} \mathscr{S}(\lambda k.e')$ for an expression e'.*
3. *If $e^{kh} =_{KH_{-}} v$ then $\exists v'$, such that $e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#} v'$.*
4. *If $e^{kh} =_{KH_{-}} \mathscr{S}(\lambda k.e')$ then $e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#}$ `uncaught:MissingReset`.*

We now present an alternative encoding which provides better error handling. To regain the non-strict behavior in the second step of the encoding we apply a standard delay/force transformation. We let $\lfloor e \rfloor$ denote the new encoding of a $\lambda_{\mathscr{K}\mathscr{A}\#}$ expression into a $\lambda_{\texttt{callcc}\mathscr{A}_s\#!}$ expression, which in addition to replacing $\mathscr{K}$ with `callcc` is defined as follows (we use $()$ to represent some arbitrary constant value) :

$$\begin{array}{rcl} \lfloor \#e \rfloor & = & (\#((\lambda x.\lambda \_.x)\ \lfloor e \rfloor))() \\ \lfloor \mathscr{A} e \rfloor & = & \mathscr{A}_s(\lambda \_.\lfloor \#e \rfloor) \end{array}$$

$\lfloor \lceil e \rceil \rfloor$ stands for the combination of the regular encoding into $\lambda_{\mathscr{K}\mathscr{A}\#}$ plus the new encoding.

Theorem 5.2. *Given a $\lambda_{\mathscr{S}\#}$ expression e:*

1. *$e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#}$ `uncaught:MissingReset` iff $\lfloor \lceil e \rceil \rfloor, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!}$ `uncaught:MissingReset`$, w_i$.*
2. *If $e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#} v$ then $\exists v', \lfloor \lceil e \rceil \rfloor, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!} v', w_i$;*
   if $\lfloor \lceil e \rceil \rfloor, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!} v, w_i$ then $\exists v', e \longmapsto\!\!\!\!\rightarrow_{\mathscr{S}\#} v'$.

Corollary 5.3. *Given a $\lambda_{\mathscr{S}\#}$ expression e,*

1. *If $e^{kh} =_{KH_{-}} v$ then $\exists v', \lfloor \lceil e \rceil \rfloor, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!} v', w_i$;*
   If $e^{kh} =_{KH_{-}} \mathscr{S}(\lambda k.e')$ then $\lfloor \lceil e \rceil \rfloor, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!}$ `uncaught:MissingReset`$, w_i$.
2. *If $\lfloor \lceil e \rceil \rfloor, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!} v, w_i$ then $\exists v', e^{kh} =_{KH_{-}} v'$;*
   if $\lfloor \lceil e \rceil \rfloor, w_i \longmapsto\!\!\!\!\rightarrow_{\texttt{callcc}\mathscr{A}_s\#!}$ `uncaught:MissingReset`$, w_i$ then $e^{kh} =_{KH_{-}} \mathscr{S}(\lambda k.e')$ for an expression e'.

If an expression *e* needs to capture a delimited continuation and no delimiter is present, then $\lfloor \lceil e \rceil \rfloor, w_i$ produces an error, whereas Filinski's encoding produces a value which might be different from the one produced by Kameyama and Hasegawa axiomatization.

Example 5.4. Going back to the example of Section 4.3, the new encoding behaves as follows ($loop\ n = \lfloor\lceil \mathscr{S}(\lambda k.loop(n\text{-}1))\rceil\rfloor$).We underline the redex perfomed, unless it is the top redex:

$$\begin{array}{rl}
 & \lfloor\lceil \#(loop\ 3)\rceil\rfloor, w_i \\
= & (\underline{\#((\lambda x.\lambda_.x)(loop\ 3))})(), w_i \\
\longmapsto & (\mathscr{A}_s((\lambda x.\lambda_.x)(\underline{loop\ 3})))(), \lambda x.mk := w_i; \langle[]\ ()\rangle x \\
\longmapsto & (\mathscr{A}_s((\lambda x.\lambda_.x)\underline{\texttt{callcc}(\lambda k.\mathscr{A}_s(\lambda_.\#((\lambda x.\lambda_.x)loop\ (3\text{-}1))()))}))(), \lambda x.mk := w_i; \langle[]\ ()\rangle x \\
\longmapsto & (\mathscr{A}_s((\lambda x.\lambda_.x)\underline{\mathscr{A}_s(\lambda_.\#((\lambda x.\lambda_.x)loop\ (3\text{-}1))()))}))(), \lambda x.mk := w_i; \langle[]\ ()\rangle x \\
\longmapsto\!\!\!\!\longmapsto & (\mathscr{A}_s((\lambda x.\lambda_.x)(\underline{\langle[]\ ()\rangle(\lambda_.\#((\lambda x.\lambda_.x)loop\ (3\text{-}1))()))}))(), w_i \\
\longmapsto & (\lambda_.\#((\lambda x.\lambda_.x)loop\ (3\text{-}1))())(), w_i \\
\longmapsto & \#((\lambda x.\lambda_.x)loop\ (3\text{-}1))(), w_i
\end{array}$$

Reduction does not always occur at the top of an expression. However, the depth of the redex contracted is bounded. With respect to Example 4.6 the new encoding behaves as follows:

$$\begin{array}{rl}
 & \lfloor\lceil \mathscr{S}(\lambda k.(k\ 1)+3)\rceil\rfloor, w_i \\
= & \texttt{callcc}(\lambda k.\mathscr{A}_s\ (\lambda_.(\#(\lambda x.\lambda_.x)(((\lambda x.(\#(\lambda x.\lambda_.x)(k\ x))\ ())1)+3))())), w_i \\
\longmapsto\!\!\!\!\longmapsto & \mathscr{A}_s\ (\lambda_.(\#(\lambda x.\lambda_.x)(((\lambda x.(\#(\lambda x.\lambda_.x)(k\ x))\ ())1)+3))), w_i \\
\longmapsto & (\lambda_.\texttt{MissingReset})(\lambda_.(\#(\lambda x.\lambda_.x)(((\lambda x.(\#(\lambda x.\lambda_.x)(k\ x))\ ())1)+3))), w_i \\
\longmapsto & \texttt{uncaught:MissingReset}, w_i
\end{array}$$

5.1 SML implementation of the alternative encoding

```
functor NewControl (type ans) : CONTROL =
struct
    open Escape
    exception MissingReset
    val mk : ((unit -> ans) -> void) ref = ref (fn _ => raise MissingReset)
    fun abort thunk = coerce (!mk thunk)
    type ans = ans
    fun reset h = escape (fn k =>
                            let val m = !mk
                            in
                                mk := (fn x => (mk := m; k x));
                                abort (let val x = h ()
                                       in fn () => x
                                       end)
                            end) ()
    fun shift h =
        escape (fn k =>
                    abort (fn () =>
                            reset (fn () =>
                                      h (fn v =>
                                            reset (fn () => coerce (k v))))))
    fun C h =
        escape (fn k =>
                    abort (fn () => reset (fn () =>
                                            h (fn v => coerce (k v)))))
end;
```

With this implementation, we can see that Examples 4.5 through 4.7 behave as expected:

```
- shift (fn k => 99);
uncaught exception MissingReset
- shift (fn k => (k 1) + 3);
uncaught exception MissingReset
- reset (fn () => (shift (fn k => raise Fail "") handle Fail _ => 99))
  handle Fail _ => 0;
val it = 0 : ans
```

6 Delaying the jump versus delaying evaluation

In the encoding of shift, a non-strict invocation of the top-level continuation is made strict. A similar problem occurs in the encoding of `callcc` in terms of the abortive control operator $\mathscr{C}$, which is expressed as follows:

$$\texttt{callcc}(\lambda k.e) = \mathscr{C}(\lambda k.k\ e) \tag{3}$$

This encoding suffers from a space leak [16, 8]. Since `callcc` does not abandon its continuation, the encoding via the abortive $\mathscr{C}$ restores the continuation by immediately applying k to e. But looking closer, if we consider the semantics of $\texttt{callcc}(\lambda k.e)$ to have an implicit occurrence of k just before e, that occurrence requires special treatment. Whereas other occurrences of k are strict, evaluating their argument before performing the jump, this first implicit occurrence follows a non-strict discipline:

$$E[\langle E'\rangle\ e] \longmapsto E'[e]$$

The encoding 3 delays this jump until *after* evaluating e, which leads to the space leak. A more faithful encoding should delay the evaluation of e until after it restores the aborted continuation. We can achieve this by placing e in a thunk, jumping and then forcing the thunk [9]. Let us denote the encoding of an expression e as $\lceil e \rceil$. It is defined as follows:

$$\begin{array}{lcl} \lceil \texttt{callcc}(\lambda k.e) \rceil & = & \mathscr{C}(\lambda k.(k\ \lambda\_.\lceil e \rceil[\lambda x.(k\ \lambda\_.x)/k]))\ () \\ \lceil \langle E \rangle \rceil & = & \lambda x.(\langle E[[]\ ()]\rangle\ \lambda\_.x) \end{array}$$

By applying the thunk outside the body of $\mathscr{C}$ (using $()$ to represent some arbitrary constant value), we force the evaluation of e after the jump.

Notice how the new encoding renders the implicit occurrence of k explicit. The special status of this occurrence is reflected in the fact that e is turned into a thunk before it is evaluated, whereas subsequent arguments are passed to k fully evaluated. This is a key insight. The specification requires a different evaluation strategy for applying k than the implementation language affords; it is not possible to apply a continuation to an unevaluated expression. So our implementation requires an encoding of expressions that does not rely on the underlying evaluation order of the implementation semantics.

Lemma 6.1. *Given an expression e:*

1. $\forall e'.e \longmapsto\!\!\!\!\rightarrow e' \Rightarrow \lceil e \rceil \longmapsto\!\!\!\!\rightarrow \lceil e' \rceil$
2. $\forall e'.\lceil e \rceil \longmapsto\!\!\!\!\rightarrow e' \Rightarrow \exists e''.e \longmapsto e'',\ e' \longmapsto\!\!\!\!\rightarrow \lceil e'' \rceil$.

Theorem 6.2. *Given an expression e, if* $e \longmapsto\!\!\!\!\rightarrow v$ *then* $\exists v', \lceil e \rceil \longmapsto\!\!\!\!\rightarrow v'$*; if* $\lceil e \rceil \longmapsto\!\!\!\!\rightarrow v$ *then* $\exists v', e \longmapsto\!\!\!\!\rightarrow v'$.

Remark 6.3. *By writing $\langle E \rangle$ as $\lambda x.\mathscr{A}\, E[x]$, the encoding of $\langle E \rangle$ can be expressed in Felleisen reduction theory $\lambda_{\mathscr{C}}$ [5]. It corresponds to multiple applications of the β_v-rule (see 1). On the contrary, the encoding (3) requires the following rule, called β_Ω:*

$$(\lambda x.\mathscr{A}\, E[x])\; e \longmapsto \mathscr{A}\, E[e] \tag{4}$$

This rule is not part of the $\lambda_{\mathscr{C}}$-calculus. It is shown correct with respect to a cps semantics in [11]. Therefore, the original encoding is sound. However, the above rule points out that the difference between the two encodings becomes observable with the addition of dynamic effects such as exceptions. In fact, as given in [11], in the presence of dynamic effects (such as the prompt) the above rule takes a more restricted form. For example, evaluation context E cannot capture an exception which is raised in e, or equivalently E cannot redefine the prompt. Thus, in the presence of exceptions, the soundness of the traditional encoding does not hold, but it holds for the new encoding.

7 Conclusions

In developing syntactic theories we are accustomed to eliminating some syntactic differences between terms. An example is the notion of α-equivalence. We do the same in proving properties of our theories. When our theorems do not hold we opt for stating that the property holds up to some syntactic manipulation of our terms. An example is the standardization theorem, which relates the operational semantics to a reduction theory [15]. The operational semantics enforces a specific order in the application of the rewriting rules. However, the reduction semantics is more flexible; the rules can be applied in any order. In general, one is interested in a more relaxed form of this theorem which relates values: $e \twoheadrightarrow v$ iff $\exists v', e \longmapsto\!\!\!\!\twoheadrightarrow v' \twoheadrightarrow v$. Unfortunately, the above theorem does not hold for $\lambda_{\mathscr{C}}$. The values v and v' are *almost* the same, but the theory does not relate them.

We emphasize the importance of understanding the reasons for this mismatch and moreover its impact. In case of $\lambda_{\mathscr{C}}$, as discussed in Ariola and Herbelin [1], the underlying reason is the absence of a reduction rule which is instead implicitly present in the operational semantics. It is the lack of this rule that prevents the theory from being extended with more expressive rules without losing important properties such as confluence.

Encodings of control operators offer other examples of a mismatch between the specification and the implementation. We present examples of encodings which are correct if one does not observe space or errors. This hides complications which only arise when the encodings are combined with other effects. We discovered that the mismatch is due to the lack of preservation of strictness. For example, the encoding of `callcc` in terms of $\mathscr{C}$ turned an implicit non-strict continuation's invocation into a strict one. Filinski's encoding of delimited control in terms of abortive control and state turned a non-strict abort operator into a strict one. This caused the system to return a "meaningless" result when we tried to use it without surrounding a program with a prompt or reset. From an intensional point of view and in the presence of effects there is a difference between a strict abort, a non-strict abort and not aborting at all [12]. In that respect, Laird's unsoundness result [13] of $\lambda_{\mathscr{C}}$ in the presence of exceptions disregards the abort operation, thus making an inference which is not justified.

Our new encodings preserve the strictness and do not suffer from the associated space leak. However, they still do not appear to address all issues of space consumption, since `callcc` still captures too much of the continuation. By making use of SML of New Jersey's `isolate` function, Herman [10] was able to address this problem but did not prove whether it is a complete solution. We leave as future work the construction of memory models for studying the space consumption of our encoding.

Finally, the new encoding of shift and reset produces an error if the program attempts to capture a delimited continuation and no delimiter is present. In the Appendix we extend our encoding to preserve

the meaning in the presence of exceptions, where the semantics of shift and exceptions is given by Herman [10]. Notice that some behaviors of Filinski's encoding in the presence of exceptions are due to the strict abort. For example, in the following expression

```
reset (fn () => (shift (fn k => raise Fail "") handle Fail _ => 99))
handle Fail _ => 0;
val it = 99 : ans
```

the context surrounding the shift-expression and up to the prompt should be abandoned. This would mean that the exception should be raised and captured by the outermost handler, thus returning `0` instead of `99`. Our proposed encoding does indeed return `0`. Our encodings however, do not preserve the number of times one aborts a computation. For example, our new encoding of `callcc` does not work in the presence of more complex constructs such as Scheme's `dynamic-wind`, which admits very fine observations of control [18]. In particular, it allows one to observe exactly when the abort happens. In future work, we wish to deepen our understanding of how to combine effects in a sound way thus avoiding unexpected behaviors.

References

[1] Z. M. Ariola and H. Herbelin. Control reduction theories: the benefit of structural substitution. *J. Functional Programming*, 18(3):373–419, 2008.

[2] Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of continuations and prompts. In *ACM SIGPLAN International Conference on Functional Programming*, pages 40–53. ACM Press, New York, 2004.

[3] O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, Copenhagen, Denmark, 1989.

[4] M. Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL '88)*, pages 180–190, Jan 1988.

[5] M. Felleisen and R. Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[6] A. Filinski. Representing monads. In *Conf. Record 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'94, Portland, OR, USA, 17–21 Jan. 1994*, pages 446–457, New York, 1994. ACM Press.

[7] A. Filinski. Representing layered monads. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188. ACM Press, 1999.

[8] M. Flatt, G. Yu, R. B. Findler, and M. Felleisen. Adding delimited and composable control to a production programming environment. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional prog ramming*, pages 165–176, New York, NY, USA, 2007. ACM.

[9] R. Heib and R. K. Dybvig. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[10] D. Herman. Functional pearl: The Great Escape. Or, how to jump the border without getting caught. In *ICFP '07: Proceedings of the Twelfth ACM SIGPLAN International Conference on Functional Programming*, pages 157–164, Oct. 2007.

[11] Y. Kameyama and M. Hasegawa. A sound and complete axiomatization of delimited continuations. In *Proc. of 8th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'03, Uppsala, Sweden, 25-29 Aug. 2003*, volume 38(9) of *SIGPLAN Notices*, pages 177–188. ACM Press, New York, 2003.

[12] O. Kiselyov, C.-c. Shan, and A. Sabry. Delimited dynamic binding. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, New York, NY, USA, 2006.

[13] J. Laird. Exceptions, continuations and macro-expressiveness. In *ESOP '02: Proceedings of the 11th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2002.

[14] C. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In *ACM workshop on Continuations*, pages 49–71, 1992.

[15] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Comput. Sci.*, 1:125–159, 1975.

[16] D. Sitaram. *Models of Control and Their Implications for Programming Language Design*. PhD thesis, Rice University, 1994.

[17] D. Sitaram and M. Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 161–175, New York, NY, 1990. ACM.

[18] M. Sperber, R. Kent Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009.

A SML implementation of the encoding in the presence of exceptions

```
functor NewControl2 (type ans) : CONTROL =
struct
    open Escape
    exception MissingReset
    datatype result  = Ok of ans | Fail of exn

    val mk0 : ((unit -> result) -> void) =
        fn r => raise MissingReset;
    val mk = ref mk0;
    fun initialize () = mk := mk0;
    fun abort x = coerce (!mk x)
    type ans = ans

    fun reset t =
        case (escape (fn k =>
                          let val m = !mk
                          in
                              mk := (fn x => (mk := m; k x));
                              abort (let val x = Ok (t ())
                                                 handle x => Fail x
                                     in fn () => x
                                     end)
                          end)
                     ()) of
        Ok x => x
      | Fail x => raise x

    fun shift h =
        escape (fn k =>
                    abort (fn () =>
                              Ok (reset (fn () =>
                                            (h (fn v =>
                                                   reset (fn () =>
                                                              coerce (k v))))))))
end;
```