A Monadic Framework for Delimited Continuations

R. Kent Dybvig
Indiana University

Simon Peyton Jones
Microsoft Research

Amr Sabry∗
Indiana University

Abstract

Delimited continuations are more expressive than traditional abortive continuations and they apparently require a framework beyond traditional continuation-passing style (CPS). We show that this is not the case: standard CPS is sufficient to explain the common control operators for delimited continuations. We demonstrate this fact and present an implementation as a Scheme library. We then investigate a typed account of delimited continuations that makes explicit where control effects can occur. This results in a monadic framework for typed and encapsulated delimited continuations, which we design and implement as a Haskell library.

1 Introduction

Continuation-passing style (CPS) and its generalisation to monadic style are the standard mathematical frameworks for understanding (and sometimes implementing) control operators. Starting in the late eighties a new class of control operators were introduced that apparently went "beyond continuations" (Felleisen *et al.*, 1987b; Felleisen, 1988; Hieb & Dybvig, 1990; Danvy & Filinski, 1990) and "beyond monads" (Wadler, 1994). These control operators permit the manipulation of *delimited continuations* that represent only part of the remainder of a computation, and they also support the *composition* of continuations, even though such operations are not directly supported by standard continuation models (Strachey & Wadsworth, 1974). Delimited continuations are also referred to as *subcontinuations* (Hieb *et al.*, 1994), since they represent the remainder of a subcomputation rather than of a computation as a whole. We use the terms interchangeably in this article.

Without the unifying frameworks of CPS or monads, it is difficult to understand, compare, implement, and reason about the various control operators for delimited continuations, their typing properties, and logical foundations. In this article, we design such a unifying framework based on continuation semantics, then generalise it to a typed monadic semantics. We illustrate this framework with a basic set of

∗ Supported by National Science Foundation grant number CCR-0196063, by a Visiting Researcher position at Microsoft Research, Cambridge, U.K., and by a Visiting Professor position at the University of Genova, Italy.

control operators that can be used to model the most common control operators from the literature (Section 2).

We give both an operational semantics and an abstract and expressive continuation semantics for these operators (Section 3). With refinements to the continuation semantics (Section 3.4), we demonstrate that any program employing delimited continuations can be evaluated via a single, completely standard CPS translation, when provided with appropriate meta-arguments and run-time versions of our operators that manipulate these arguments. The refined semantics also establishes that the implementation of delimited continuations is independent of the representation of the undelimited *partial continuations* between delimiters. Partial continuations might be represented, for example, by CPS functions, sequences of individual stack frames, or stack segments containing contiguous blocks of stack frames as in Hieb *et al.*'s efficient implementation of traditional continuations (1990).

We then factor the continuation semantics into two parts: a translation into a monadic language that specifies the order of evaluation, and a library that implements the control operators themselves (Section 4). This semantics can easily be used to guide a straightforward implementation in Scheme, as we show in Section 5.

We then tackle the problem of *typing* delimited continuations. Building on the monadic semantics, we give a typed account of the delimited continuation operators that makes explicit where control effects can occur, and where they cannot (Section 6). In particular, our design builds on ideas by Thielecke (2003), Launchbury and Peyton Jones (1995) to offer statically checked guarantees of encapsulation of control effects. We introduce an operator `runCC` that encapsulates a computation that uses control effects internally but is purely functional when viewed externally.

Once the monadic effects have been made apparent by the first translation, the control operators can be implemented as a *typed library.* This offers the opportunity to prototype design variations—of both implementation approach and library interface—in a lightweight way. We make this concrete, using Haskell as an implementation of the monadic language, by providing three different prototype implementations of the control operators (Section 7). The first of these implementations is suitable for low-level manipulations of the stack, the second suitable for a CPS compilation strategy, and the third suitable for a language that provides access to the entire abortive continuation using an operator like *call-with-current-continuation* (*callcc* for short) or Felleisen's $\mathcal{C}$ operator (1987b). The library implementation is itself strongly typed (except for a function that compares the internal representations of the control delimiters as explained in Section 7). The types helped enormously when writing the library's rather tricky code.

The Scheme and Haskell code described in the article is available at `http://www.cs.indiana.edu/~sabry/papers/CC_code.tar.gz` under the MIT License.

2 Control Operators

Many operators for delimited control have been described in the literature. We take no position in this article on which is best, but instead adopt a set of four control operators that can be used to construct a wide variety of delimited control abstractions. In this section, we describe these building blocks and show how they can be used to model the most common delimited control operators.

2.1 Our Operators

The operators in our family are *newPrompt*, *pushPrompt*, *withSubCont*, and *pushSubCont*. We explain them in the context of a conventional, call-by-value λ-calculus:

$$
\begin{array}{llcl}
\text{(Variables)} & x, y, \ldots & & \\
\text{(Expressions)} & e & ::= & x \mid \lambda x.e \mid e\ e \\
 & & \mid & \mathit{newPrompt} \mid \mathit{pushPrompt}\ e\ e \\
 & & \mid & \mathit{withSubCont}\ e\ e \mid \mathit{pushSubCont}\ e\ e
\end{array}
$$

The semantics is given formally in Section 3. Intuitively the operators behave as follows:

- The *newPrompt* operator creates a new prompt, distinct from all existing prompts.
- The *pushPrompt* operator evaluates its first operand and uses the resulting value, which must be a prompt, to delimit the current continuation during the evaluation of its second operand.
- The *withSubCont* operator evaluates both of its operands, yielding a prompt p and a function f. It captures a portion of the current continuation back to but not including the activation of *pushPrompt* with prompt p, aborts the current continuation back to and including the activation of *pushPrompt*, and invokes f, passing it an abstract value representing the captured subcontinuation. If more than one activation of *pushPrompt* with prompt p is still active, the most recent enclosing activation, *i.e.*, the one that delimits the smallest subcontinuation, is selected.
- The *pushSubCont* operator evaluates its first operand to yield a subcontinuation k, then evaluates its second operand in a continuation that composes k with the current continuation.

While *newPrompt* and *withSubCont* can be treated as functions, the operators *pushPrompt* and *pushSubCont* must be treated as syntactic constructs since they exhibit a non-standard evaluation order.

Example: Illustrating the operators

The following simple example illustrates the operators in a language augmented with conditionals, arithmetic, and constants:

$$
\begin{array}{l}
(\lambda p.\ 2 + \mathit{pushPrompt}\ p \\
\qquad\qquad \text{if } (\mathit{withSubCont}\ p \\
\qquad\qquad\qquad (\lambda k.\ (\mathit{pushSubCont}\ k\ \text{False})\ +\ (\mathit{pushSubCont}\ k\ \text{True}))) \\
\qquad\qquad\quad \text{then } 3 \\
\qquad\qquad\quad \text{else } 4) \\
\quad \mathit{newPrompt}
\end{array}
$$

The fresh prompt p bound by the outer λ-expression is pushed just prior to the evaluation of the if expression, with the outer application of + pending. The application of *withSubCont* captures and aborts the subcontinuation back to the prompt, so the entire *pushPrompt* form reduces to the body of the inner λ-expression. Thus, we are left with:

$$2 \; + \; ((\mathit{pushSubCont}\; k \; \text{False}) \; + \; (\mathit{pushSubCont}\; k \; \text{True}))$$

where the subcontinuation k is, in essence, a function that accepts a boolean value and returns one of the two values 3 or 4, i.e.:

$$\lambda b. \text{ if } b \text{ then } 3 \text{ else } 4$$

The first pushSubCont supplies False to the subcontinuation, yielding 4, and the second supplies True, yielding 3, so the entire expression reduces to:

$$2 + 4 + 3$$

and the final answer is, of course, 9.

2.2 Relationship with Existing Operators

Traditional continuations represent the *entire* rest of the computation from a given execution point, and, when reinstated, they *abort* the context of their use. Our operators can access the entire continuation only if a top-level prompt is pushed before the execution of a program, which is easily done by running the program in the context of a top-level binding for a prompt created by *newPrompt*:

$$\mathit{run}\; e \quad = \quad (\lambda p_0.\mathit{pushPrompt}\; p_0 \; e) \; \mathit{newPrompt}$$

Assuming this has been done, we can define a *withCont* operator to manipulate (*i.e.*, capture and abort) the entire continuation via this prompt:

$$\mathit{withCont}\; e \quad = \quad \mathit{withSubCont}\; p_0 \; (\lambda k.\mathit{pushPrompt}\; p_0 \; (e \; k))$$

Because of the ease with which a top-level prompt can be introduced in the source language, we do not include a built-in top-level prompt as part of our model. Doing so might even be considered a poor design from a software-engineering perspective, since a built-in top-level prompt gives subprograms possibly undesirable control over the main program. Furthermore, if the operators are typed (as they will be shortly), it is not clear what the type of the top-level prompt would be.

Given *withCont*, we can model Scheme's *call-with-current-continuation* (here abbreviated *callcc*), which captures the current continuation, encapsulates it into an *escape procedure*, and passes this escape procedure to its argument:

$$\mathit{callcc} \quad = \quad \lambda f.\mathit{withCont}\; (\lambda k.\mathit{pushSubCont}\; k \; (f \; (\mathit{reifyA}\; k)))$$

where:

$$\begin{aligned} \mathit{reifyA}\; k \quad &= \quad \lambda v.\mathit{abort}\; (\mathit{pushSubCont}\; k \; v) \\ \mathit{abort}\; e \quad &= \quad \mathit{withCont}\; (\lambda\_.e) \end{aligned}$$

When applied to a function f, *callcc* captures and aborts the entire continuation k

using *withCont*, uses *pushSubCont* to reinstate a copy of k, and applies f to the escape procedure $(\mathit{reifyA}\ k)$. When applied to a value v, this escape procedure aborts its entire context, reinstates k as the current entire continuation, and returns v to k.

Felleisen's $\mathcal{C}$ (1987a) is a variant of *callcc* that aborts the current continuation when it captures the continuation. It can be modelled similarly:

$$\mathcal{C} \quad = \quad \lambda f.\mathit{withCont}\ (\lambda k.f\ \ (\mathit{reifyA}\ k))$$

Like continuations reified by *callcc*, a continuation reified by $\mathcal{C}$ aborts the current continuation when it is invoked. In contrast, the operator $\mathcal{F}$ (Felleisen & Friedman, 1987; Felleisen *et al.*, 1987b) also captures and aborts the entire continuation, but the reified continuation is functional, or composable, as with our subcontinuations. It can be modelled with a non-aborting *reify* operator:

$$\mathcal{F} \quad = \quad \lambda f.\mathit{withCont}\ (\lambda k.f\ \ (\mathit{reify}\ k))$$

where:

$$\mathit{reify}\ k \quad = \quad \lambda v.\mathit{pushSubCont}\ k\ v$$

When prompts appear other than at top level, they serve as control delimiters (Felleisen *et al.*, 1987a; Felleisen, 1988; Danvy & Filinski, 1990) and allow programs to capture and abort a subcontinuation, *i.e.*, a continuation representing *part of* the remainder of the computation rather than *all* of it. The first control delimiter to be introduced was Felleisen's # (prompt), which delimits, *i.e.*, marks the base of, the continuation captured and aborted by $\mathcal{F}$(Felleisen *et al.*, 1987a; Felleisen, 1988). The addition of first-class control delimiters was motivated by both practical and semantic considerations: in fact, control delimiters appear necessary for full abstraction in any denotational model of continuations (Sitaram & Felleisen, 1990b; Sitaram & Felleisen, 1991).

In the presence of prompts, the operator $\mathcal{F}$ captures and aborts the continuation up to but not including the closest enclosing prompt. This means that the prompt remains in place after a call to $\mathcal{F}$, and the captured subcontinuation does not include the prompt. Variants of $\mathcal{F}$ have been introduced since that do not leave behind the prompt when a subcontinuation is captured, or do include the prompt in the captured subcontinuation. For example, the *reset* and *shift* operators of Danvy and Filinski (1990) are similar to # and $\mathcal{F}$, but *shift* both leaves behind the prompt when a subcontinuation is captured and includes the prompt in the captured subcontinuation.

To illustrate these differences, we introduce a classification of control operators in terms of four variants of $\mathcal{F}$ that differ according to whether the continuation-capture operator (a) leaves behind the prompt on the stack after capturing the continuation and (b) includes the prompt at the base of the captured subcontinuation:

- $^{-}\mathcal{F}^{-}$ neither leaves the prompt behind nor includes it in the subcontinuation; this is like our operator *withSubCont* and *cupto* (Gunter *et al.*, 1995).
- $^{-}\mathcal{F}^{+}$ does not leave the prompt behind, but does include it in the subcontinuation; this is like a *spawn* controller (Hieb & Dybvig, 1990).

$^{+}\mathcal{F}^{-}$ leaves the prompt behind, but does not include it in the subcontinuation; this is the delimited $\mathcal{F}$ operator (Felleisen *et al.*, 1987a).

$^{+}\mathcal{F}^{+}$ both leaves the prompt behind and includes it in the subcontinuation; this is the *shift* operator (Danvy & Filinski, 1990).

In all cases, the traditional interface is that the captured subcontinuation is reified as a function. Using our primitives and a single prompt #, which can be established in a manner similar to p_0 above, these operators can be defined as follows:

$$\begin{array}{rcl}
^{-}\mathcal{F}^{-} & = & \lambda f.\mathit{withSubCont}\ \#\ (\lambda k.f\ (\mathit{reify}\ k)) \\
^{-}\mathcal{F}^{+} & = & \lambda f.\mathit{withSubCont}\ \#\ (\lambda k.f\ (\mathit{reifyP}\ \#\ k)) \\
^{+}\mathcal{F}^{-} & = & \lambda f.\mathit{withSubCont}\ \#\ (\lambda k.\mathit{pushPrompt}\ \#\ (f\ (\mathit{reify}\ k))) \\
^{+}\mathcal{F}^{+} & = & \lambda f.\mathit{withSubCont}\ \#\ (\lambda k.\mathit{pushPrompt}\ \#\ (f\ (\mathit{reifyP}\ \#\ k)))
\end{array}$$

where:

$$\begin{array}{rcl}
\mathit{reify}\ k & = & \lambda v.\mathit{pushSubCont}\ k\ v \\
\mathit{reifyP}\ p\ k & = & \lambda v.\mathit{pushPrompt}\ p\ (\mathit{pushSubCont}\ k\ v)
\end{array}$$

The definitions of $^{+}\mathcal{F}^{-}$ and $^{+}\mathcal{F}^{+}$ are consistent with the "folklore theorem" that *shift* can be simulated using $\mathcal{F}$ by wrapping each application of the continuation with a prompt (Biernacki & Danvy, 2006): the only difference in their definitions is that the application of the reified continuation uses an extra *pushPrompt* in the case of $^{+}\mathcal{F}^{+}$.

Each variant of $\mathcal{F}$ has merits. $^{+}\mathcal{F}^{-}$ and $^{-}\mathcal{F}^{-}$ produce purely functional subcontinuations that do not reinstate a prompt; these subcontinuations can be understood as ordinary functions. With $^{-}\mathcal{F}^{+}$, the prompt is attached to and only to a given subcontinuation, so that the subcontinuation represents a self-contained subcomputation. $^{+}\mathcal{F}^{+}$ has a close relationship with traditional CPS and behaves in a manner that can be understood in terms of that relationship. $^{+}\mathcal{F}^{-}$ and $^{+}\mathcal{F}^{+}$ both encapsulate all effects within their dynamic extent, since the prompt remains even after a subcontinuation is captured.

$^{+}\mathcal{F}^{-}$ and $^{-}\mathcal{F}^{+}$ share an intuitively appealing identity property, which is that capturing and immediately reinstating a subcontinuation is effectively a no-op. In other words the following two equations are sound with respect to the semantics of the operators:

$$\begin{array}{rcll}
^{+}\mathcal{F}^{-}\ (\lambda k.kv) & = & v & \text{if } k \notin FV(v) \\
^{-}\mathcal{F}^{+}\ (\lambda k.kv) & = & v & \text{if } k \notin FV(v)
\end{array}$$

where $FV(e)$ gives the free variables of term e. In the former case, the prompt is left behind when the continuation is captured, and in the latter, the prompt is reinstated when the continuation is invoked. The corresponding identity does not hold for $^{-}\mathcal{F}^{-}$, since capturing and reinstating a subcontinuation results in the net elimination of one prompt. The same operation with $^{+}\mathcal{F}^{+}$ results in the net introduction of one prompt. In a system supporting only $^{+}\mathcal{F}^{+}$, however, the extra prompt makes no difference denotationally, and pairs of adjacent prompts can be collapsed into one prompt to avoid build-up of the control stack operationally.

We have chosen $^{-}\mathcal{F}^{-}$ semantics for our building-block operator *withSubCont* because the $^{-}\mathcal{F}^{-}$ semantics easily models each of the others with the trivial addition of *pushPrompt* forms where necessary to leave behind or reinstate a prompt, as demonstrated by the definitions above. While Shan (2004) and Kiselyov (2005) have demonstrated that one can use the $^{+}\mathcal{F}^{+}$ semantics (in the form of *shift*) to implement the semantics of $^{-}\mathcal{F}^{-}$, $^{-}\mathcal{F}^{+}$, and $^{+}\mathcal{F}^{-}$, and their technique could be adapted to work with $^{+}\mathcal{F}^{-}$ or $^{-}\mathcal{F}^{+}$ as the basis operator, the technique requires a complex syntactic redefinition of the prompt operator. This additional complexity makes these other variants less suitable as basic building blocks for the variety of proposed control abstractions.

A natural extension of the framework with a single fixed prompt is to allow multiple prompts. Some proposals generalise the single prompt by allowing hierarchies of prompts and control operators, like $reset_n$ and $shift_n$ (Danvy & Filinski, 1990; Sitaram & Felleisen, 1990a). Other proposals instead allow new prompts to be generated dynamically, like *spawn* (Hieb & Dybvig, 1990; Hieb *et al.*, 1994). In such systems, the base of each subcontinuation is rooted at a different prompt, and each generated prompt is associated with a function that can be used for accessing the continuation up to that prompt. This is more convenient than either single prompts or hierarchies of prompts and allows arbitrary nesting and composition of subcontinuation-based abstractions. In our framework, *spawn* is defined as follows:

$$\begin{aligned} \mathit{spawn} &= \lambda f.(\lambda p.\mathit{pushPrompt}\ p\ (f\ (^{-}\mathcal{F}^{+}\ p)))\ \mathit{newPrompt} \\ ^{-}\mathcal{F}^{+} &= \lambda p.\lambda f.\mathit{withSubCont}\ p\ (\lambda k.f\ (\mathit{reifyP}\ p\ k)) \end{aligned}$$

where we have generalised the definition of $^{-}\mathcal{F}^{+}$ to take a prompt argument p instead of referring to the fixed prompt #. Thus, *spawn* generates a new prompt, pushes this prompt, creates a control operator that can access this prompt, and makes this specialised control operator available to its argument f.

Moreau and Queinnec (1994) proposed a pair of operators, *marker* and *call/pc*, based on an earlier mechanism by Queinnec and Serpette (1991), that provide functionality similar to that of *spawn*. The *marker* operator generates a new prompt and pushes it, and *call/pc* captures and aborts the subcontinuation rooted at a given prompt. The key difference from *spawn* is that the continuation reified by *call/pc* is stripped of *all* intervening prompts, even though they are necessarily unrelated to the prompt at the base. We could model this behaviour in our system with the addition of a *strip* operator as follows:

$$\begin{aligned} \mathit{marker}\ e &= (\lambda p.\mathit{pushPrompt}\ p\ (e\ p))\ \mathit{newPrompt} \\ \mathit{call/pc} &= \lambda p.\lambda f.\mathit{withSubCont}\ p\ (\lambda k.f\ (\mathit{reify}\ (\mathit{strip}\ k))) \end{aligned}$$

Since captured continuations are stripped of their prompts, this system ensures that prompts can be pushed only *explicitly*, by *pushPrompt*, never implicitly by reinstatement of a subcontinuation. Such an operator is easily added to our system given the implementation approach we present later in this article. We do not do so, however, because the stripping behaviour of *call/pc* is unique in the world of control operators and, in our opinion, not useful, since it inhibits the nesting of control abstractions.

Gunter *et al.* (1995) proposed a set of three operators that provide functionality similar to *spawn* but make the creation and use of prompts explicit: *new_prompt* creates a new prompt, **set** p **in** e pushes the prompt p during the execution of e, and **cupto** p **as** x **in** e captures and aborts the current continuation up to prompt p, then evaluates e within a binding of x to a function representing the captured continuation. In our framework, these operators can be defined as follows.

$$\begin{array}{rcl} \textit{new\_prompt} & = & \textit{newPrompt} \\ \textbf{set}\ p\ \textbf{in}\ e & = & \textit{pushPrompt}\ p\ e \\ \textbf{cupto}\ p\ \textbf{as}\ x\ \textbf{in}\ e & = & \textit{withSubCont}\ p\ (\lambda k.(\lambda x.e)\ (\lambda v.\textit{pushSubCont}\ k\ v)) \end{array}$$

In addition to minor syntactic details, our framework differs from the one by Gunter *et al.* in the representation of continuations and how they are used. In Gunter *et al.*'s framework, continuations are represented as functions; in ours, the representation is left abstract. More significantly, in Gunter *et al.*'s framework, a continuation is always applied to a value, which effects an immediate return to the continuation of the **set** form that captured the topmost segment of the continuation. Our *pushSubCont* operator is more general in that it permits the evaluation of an arbitrary expression in the context of the captured continuation, obviating the awkward protocol of returning a thunk to be thawed in the context of the continuation (Dybvig & Hieb, 1989).

Each control abstraction covered above supports fewer operators than our set, which has four, so it is natural to wonder whether four is actually the "right" number. In fact, any system that supports dynamic prompt generation must provide mechanisms for (1) creating a delimiter, (2) delimiting the current continuation, (3) capturing a continuation, (4) aborting a continuation, and (5) reinstating a continuation. Systems that support a single delimiter need only provide mechanisms 2–5, and systems that do not support delimiters at all need only provide mechanisms 3–5. Some mechanisms may be combined in a single operator. For example, all of the abstractions described above, except *callcc*, combine mechanisms 3 and 4, *callcc* combines mechanisms 4 and 5, and *spawn* combines mechanisms 1 and 2. Some mechanisms may be performed implicitly by functions created by the abstraction. For example, all of the abstractions described above, including *callcc*, embed captured continuations in functions that perform the mechanism for reinstating the embedded continuation implicitly when invoked; *spawn* also embeds the prompt in a function that aborts and captures a continuation when invoked. Following most of the other abstractions, we have chosen to combine mechanisms 3 and 4, in our *withSubCont* operator, so that the portion of the current continuation represented by a captured subcontinuation need not exist in multiple places at the same time. This choice results in no loss of expressiveness, since it is a trivial matter to immediately reinstate the captured subcontinuation. We have chosen to make each of the other mechanisms explicit and separate, for the sake of simplicity and expressiveness, and to avoid the embedding of continuations and prompts in functions so that we can leave their representations abstract. While not necessarily the best choices for higher-level control abstractions, we believe that these are appropriate choices for a set of control-abstraction building blocks.

3 Operational and Continuation Semantics

In this section, we develop both an operational semantics and a continuation semantics for the call-by-value λ-calculus embedding of our operators. We review the CPS semantics traditionally used for simple control operators like *callcc* and explain why it is insufficient for *delimited* continuations (Section 3.1). We then discuss why neither of the two standard approaches to extending the CPS translation for delimited continuations is entirely satisfactory for our purposes (Section 3.2). Thus motivated, we develop the semantics in both an operational style and a continuation-passing style in Sections 3.3 and 3.4. We conclude the section by establishing the correctness of the CPS semantics with respect to the operational semantics.

3.1 Standard CPS Semantics

For the pure call-by-value λ-calculus, the CPS semantics is defined as follows. The map $\mathcal{P}[\![.]\!]$ takes an expression in the call-by-value calculus of Section 2.1 (without the control operations for now) and returns an expression in a conventional target language. More precisely, the target language of the CPS translation is a call-by-name λ-calculus with the usual theory consisting of the β and η axioms. The result of the translation is always a λ-expression that expects a continuation and returns the final answer of the program:

$$\begin{array}{rcl}
\mathcal{P}[\![v]\!] & = & \lambda\kappa.\kappa\ \mathcal{V}[\![v]\!] \\
\mathcal{P}[\![e_1 e_2]\!] & = & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda f.\mathcal{P}[\![e_2]\!]\ (\lambda a.f\ a\ \kappa)) \\
\\
\mathcal{V}[\![x]\!] & = & x \\
\mathcal{V}[\![\lambda x.e]\!] & = & \lambda x.\lambda\kappa'.\mathcal{P}[\![e]\!]\ \kappa'
\end{array}$$

The translation of a complete program is given by $\mathcal{P}[\![e]\!]\ \kappa_0$, where κ_0 is the initial continuation $\lambda v.v$. The translation introduces variables that are assumed not to occur free in the input expression.

Adding *callcc* to the pure fragment is straightforward:

$$\mathcal{P}[\![\mathit{callcc}\ e]\!] \quad = \quad \lambda\kappa.\mathcal{P}[\![e]\!]\ (\lambda f.f\ (\lambda x.\lambda\kappa'.\kappa\ x)\ \kappa)$$

After evaluating its operand, *callcc* applies the resulting function f to a function encapsulating the captured continuation κ, in the same continuation. If the function encapsulating κ is applied to a value, it reinstates the captured continuation by passing the value to κ, discarding the current continuation κ'.

Handling even a single prompt is not so straightforward. What we want is a way to split a continuation κ into two pieces at the prompt. The continuation is represented as a pure function, however, so splitting it is not an option. What we need instead is a richer representation of the continuation that supports two operations: $\kappa^p_\uparrow$ representing the portion of κ above (younger than) the prompt p, and $\kappa^p_\downarrow$ representing the portion of κ below (older than) the prompt p. We review in the next section two ideas that have been previously used to express such operations on continuations.

3.2 Traditional Solutions

Two basic approaches have been proposed to deal with the fact that the representation of continuations as functions is not sufficiently expressive:

1. Abstract continuation semantics (Felleisen *et al.*, 1988). This approach develops an *algebra of contexts* that is expressive enough to support the required operations on continuations. From the algebra, two representations for continuations are derived: one as a sequence of frames, and the other as objects with two methods for invoking and updating the continuation. In the first representation, the operations $\kappa^p_\uparrow$ and $\kappa^p_\downarrow$ can be realised by traversing the sequence up to the first prompt and returning the appropriate subsequence, and the composition of continuations can be implemented by appending the two corresponding sequences.
2. Metacontinuations (Danvy & Filinski, 1990). Since we must split the continuation above and below the prompt, we can simply maintain two separate parameters to the CPS semantics. The first parameter κ corresponds to the portion of the evaluation context above the first prompt, and the second parameter γ corresponds to the portion of the evaluation context below the first prompt. The parameter κ is treated as a *partial continuation*, *i.e.*, a function from values to *partial* answers that must be delivered to the second parameter γ to provide *final* answers. In other words, given the two continuation parameters κ and γ and a value v one would compute the final answer using $\gamma(\kappa v)$. If the nested application is itself expressed in CPS as $\kappa v \gamma$, it becomes apparent that γ is a continuation of the continuation, or in other words a *metacontinuation*.

Unfortunately, neither approach is ideal for our purposes. The metacontinuation approach leads to control operators with the ${}^+\mathcal{F}^+$ semantics, from which the other semantic variants may be obtained only with difficulty, as discussed in Section 2. While this is suitable for *shift* and *reset*, which exhibit the ${}^+\mathcal{F}^+$ semantics precisely because they are defined via the metacontinuation approach, it is not suitable for our *withSubCont* operator, which exhibits the more basic ${}^-\mathcal{F}^-$ semantics by design. The standard metacontinuation approach also performs two CPS conversion passes: a nonstandard one that exposes the continuation but leaves behind non-tail calls representing the metacontinuation, followed by a standard one that exposes the metacontinuation. Dynamically generated prompts pose additional problems. A static hierarchy of *reset* and *shift* operators can be implemented via additional CPS translation passes (Danvy & Filinski, 1990), but this technique does not extend to dynamically generated prompts. It is possible to handle dynamically generated prompts with an adaptation of the reset handlers of Sitaram and Felleisen (1990a) or Shan (2004), but doing so further complicates these mechanisms (Oleg Kiselyov, personal communication, 2005).

On the other hand, the algebra of contexts is unnecessarily concrete, *i.e.*, not truly abstract, despite the name "abstract continuation semantics." We would like to use the semantics not only as a specification for what programs mean, but also as a stepping stone to an implementation. The algebra of contexts over-specifies this implementation. For example, although a common run-time representation of continuations is indeed as a stack of frames, it is not the only one. We would prefer a model that allows control operators to be built on top of *any* existing abstraction of continuations, for example, on top of a CPS intermediate language representing continuations as functions, or on top of a language with an implementation of *callcc* that gives access to some unknown representation of the continuation. Furthermore, even if the implementation does use a stack of frames, the semantics suggests that an implementation must loop through these frames individually (Gasbichler & Sperber, 2002), even though prompts may be many frames apart. We would like the cost of a control operation to be proportional to the number of control points that are involved, not the (potentially much larger) number of frames.

3.3 Operational Semantics

It turns out that we can strike a middle ground that provides the expressiveness of the sequence of frames approach while leaving the representation of continuations truly abstract. We do this by adopting features of both of the traditional approaches to modelling delimited continuations. From the algebra of contexts, we borrow the representation of a continuation as a sequence, while from the metacontinuation approach, we borrow the clear delineation between partial continuations and metacontinuations and the abstract representation of partial continuations. In essence, we implement the algebra of contexts in a more abstract way, with the concrete subsequences of frames between control delimiters replaced by abstract partial continuations. From another perspective, we implement the metacontinuation approach with the abstract metacontinuation replaced by a sequence of partial continuations and delimiters. A similar approach was first used by Moreau and Queinnec (1994) in a semantics for *marker* and *call/pc*, although it was not developed as fully as we develop it here.

We begin by formalising the idea using an abstract machine that manipulates syntactic representations of continuations. In order to express the intermediate results of evaluation as syntactic terms, we extend the set of expressions with prompt values p represented as integers, with contexts D representing delimited (partial) continuations, and with sequences E of prompts and delimited contexts representing the rest of the continuation beyond the first delimiter, *i.e.*, the metacontinuation. None of these constructs may appear in source programs.

$$
\begin{array}{llcl}
\text{(Prompt names)} & p, q, \ldots & \in & \mathbb{N} \\
\text{(Values)} & v & ::= & x \mid \lambda x.e \mid p \mid E \\
\text{(Delimited Contexts)} & D & ::= & \square \mid D\ e \mid v\ D \\
 & & \mid & \mathit{pushPrompt}\ D\ e \mid \mathit{pushSubCont}\ D\ e \\
 & & \mid & \mathit{withSubCont}\ D\ e \mid \mathit{withSubCont}\ p\ D \\
\text{(Sequences)} & E & ::= & [] \mid p : E \mid D : E
\end{array}
$$

The operational semantics rewrites configurations of the form (e, D, E, p) where e is the current expression of interest, D is the *current context* (up to but not including

Group I. Searching for a redex:

(ee', D, E, q)	$\longmapsto$	$(e, D[\Box\ e'], E, q)$	e non-value
(ve, D, E, q)	$\longmapsto$	$(e, D[v\ \Box], E, q)$	e non-value
$(\mathit{pushPrompt}\ e\ e', D, E, q)$	$\longmapsto$	$(e, D[\mathit{pushPrompt}\ \Box\ e'], E, q)$	e non-value
$(\mathit{withSubCont}\ e\ e', D, E, q)$	$\longmapsto$	$(e, D[\mathit{withSubCont}\ \Box\ e'], E, q)$	e non-value
$(\mathit{withSubCont}\ p\ e, D, E, q)$	$\longmapsto$	$(e, D[\mathit{withSubCont}\ p\ \Box], E, q)$	e non-value
$(\mathit{pushSubCont}\ e\ e', D, E, q)$	$\longmapsto$	$(e, D[\mathit{pushSubCont}\ \Box\ e'], E, q)$	e non-value

Group II. Executing a redex:

$((\lambda x.e)v, D, E, q)$	$\longmapsto$	$(e[v/x], D, E, q)$	
$(\mathit{newPrompt}, D, E, q)$	$\longmapsto$	$(q, D, E, q+1)$	
$(\mathit{pushPrompt}\ p\ e, D, E, q)$	$\longmapsto$	$(e, \Box, p : D : E, q)$	
$(\mathit{withSubCont}\ p\ v, D, E, q)$	$\longmapsto$	$(v\ (D : E^p_\uparrow), \Box, E^p_\downarrow, q)$	
$(\mathit{pushSubCont}\ E'\ e, D, E, q)$	$\longmapsto$	$(e, \Box, E' +\!+ (D : E), q)$	

Group III. Returning a value:

(v, D, E, q)	$\longmapsto$	$(D[v], \Box, E, q)$	$D \neq \Box$
$(v, \Box, p : E, q)$	$\longmapsto$	$(v, \Box, E, q)$	
$(v, \Box, D : E, q)$	$\longmapsto$	(v, D, E, q)	

Fig. 1. Operational semantics

any pushed prompts), E is the rest of the context represented as a sequence, and p is a global counter used to generate new prompt values. The empty delimited context, sometimes called a "hole," is denoted using the box $\Box$; sequences are represented using the Haskell syntactic conventions for lists with $[]$ representing the empty sequence, : representing *cons*, and ++ representing *append*. The notation $D[e]$ denotes the result of replacing the hole of the context D with the expression e. In order to split a sequence at a particular prompt, we use the operations $E^p_\uparrow$ and $E^p_\downarrow$ specified as follows. If E contains a pushed prompt p then it can be uniquely decomposed to the form $E' +\!+ (p : E'')$ with $p \notin E'$, then $E^p_\uparrow = E'$ and $E^p_\downarrow = E''$. In other words, $E^p_\uparrow$ gives the subsequence before the first occurrence of p, and $E^p_\downarrow$ gives the subsequence after the first occurrence of p.

The evaluation of a closed expression e starts with rewriting of the configuration $(e, \Box, [], 0)$ and terminates with a final configuration of the form $(v, \Box, [], q)$. The rewriting rules in Figure 1 are organised in three groups explained below.

The first group of reductions simply builds the context D according to the *order of evaluation* specified by the definition of contexts. For example, in a function application, the evaluation context $v\ D$ specifies that rewrites take place inside the argument only if the function is a value v, thereby specifying that the function must be evaluated before the argument. Similar contexts for *pushPrompt* and *withSubCont* ensure that their first (prompt-valued) argument is evaluated first. None of the rules in this first group uses the components E or q of the configuration.

The second group of reductions specifies the actual actions to be performed by each construct. Function application is modelled by a β_v reduction as usual. The generation of a new prompt value uses the global counter and increments it as a side effect. Pushing a prompt empties the current context (which can only extend up to the first pushed prompt). The context D before the operation is itself saved in the sequence E. The fourth rewrite rule captures subcontinuation values. The control context surrounding the operation is split into three: a part $(D : E')$ which is *before* the *innermost* pushed prompt p; the prompt p itself; and the rest of the sequence E *after* the first prompt and which may include other pushed occurrences of p, dynamically more distant from the active subexpression. The sequence E' may itself contain pushed prompts as long as they are different from p. The operation (*withSubCont* p v) captures the subcontinuation up to but *not including* the prompt p, aborts the captured subcontinuation and the prompt p, and passes the captured continuation to v. The operation is undefined if the prompt p is not pushed in the current continuation. In the last rule, E' is a continuation value obtained by evaluating the first argument of *pushSubCont*; the right-hand side simply installs E' before the current continuation $(D : E)$. Notice that *the second operand of pushSubCont, namely e, is not evaluated*: there is no context *pushSubCont* v D, so no evaluation can occur inside *pushSubCont*'s second argument until the *pushSubCont* form is rewritten.

The final group of reductions pops back the context D and the elements from E when the current expression has been reduced to a value.

3.4 An Expressive and Abstract CPS Semantics

Using the ideas developed in the previous sections, we now develop a CPS translation for the call-by-value λ-calculus embedding of our operators, which will serve as the basis for the monadic semantics presented in Section 4. The translation is given in Figure 2 and refined in Figures 3 and 4.

We begin with the CPS translation in Figure 2. The continuation κ has a conventional functional representation and corresponds to the control context D in the operational semantics. The metacontinuation γ is a sequence of prompts and continuations, and corresponds to the sequence E in the operational semantics. The metacontinuation and the global prompt counter q are simply passed along as additional parameters.

The translation of a complete program e is $\mathcal{P}[\![e]\!]\kappa_0$ $[]$ 0 where κ_0 is the initial partial continuation, the constant $[]$ is the initial empty metacontinuation, and 0 is the first generated prompt name. The initial continuation κ_0 is $\lambda v.\lambda\gamma.\lambda q.\mathcal{K}(v, \gamma, q)$: it takes a value v, a metacontinuation γ, and the global counter q and applies the metacontinuation to the value using an operation $\mathcal{K}(.,.,.)$. This application simulates the third group of reductions in the operational semantics. The initial continuation is used not only at the top level but also whenever a control operation is performed as the clauses for *pushPrompt*, *withSubCont*, and *pushSubCont* show.

The target language of the CPS translation is the same call-by-name λ-calculus with the equational theory based on $\beta\eta$ but extended with constants for building and destructing lists, with integers and associated operations needed to manipulate the global counter, and with the operation $\mathcal{K}(.,.,.)$ defined by the three equations in Figure 2. To reason about the correctness of the CPS translation with respect to the operational semantics, we only the pure theory consisting of $\beta\eta$.

$$
\begin{array}{rcl}
\mathcal{P}[\![v]\!] & = & \lambda\kappa.\lambda\gamma.\lambda q.\kappa\ \mathcal{V}[\![v]\!]\ \gamma\ q \\
\mathcal{P}[\![e_1 e_2]\!] & = & \lambda\kappa.\lambda\gamma.\lambda q. \\
& & \quad \mathcal{P}[\![e_1]\!]\ (\lambda f.\lambda\gamma'.\lambda q'. \\
& & \quad \mathcal{P}[\![e_2]\!]\ (\lambda a.\lambda\gamma''.\lambda q''.f\ a\ \kappa\ \gamma''\ q'')\ \gamma'\ q')\ \gamma\ q \\
\\
\mathcal{P}[\![\mathit{newPrompt}]\!] & = & \lambda\kappa.\lambda\gamma.\lambda q.\kappa\ q\ \gamma\ (q+1) \\
\mathcal{P}[\![\mathit{pushPrompt}\ e_1\ e_2]\!] & = & \lambda\kappa.\lambda\gamma'.\lambda q. \\
& & \quad \mathcal{P}[\![e_1]\!]\ (\lambda p.\lambda\gamma.\lambda q'.\mathcal{P}[\![e_2]\!]\ \kappa_0\ (p:\kappa:\gamma)\ q')\ \gamma'\ q \\
\mathcal{P}[\![\mathit{withSubCont}\ e_1\ e_2]\!] & = & \lambda\kappa.\lambda\gamma''.\lambda q'. \\
& & \quad \mathcal{P}[\![e_1]\!]\ (\lambda p.\lambda\gamma'.\lambda q. \\
& & \quad \mathcal{P}[\![e_2]\!]\ (\lambda f.\lambda\gamma.\lambda q''.f\ (\kappa:\gamma^p_\uparrow)\ \kappa_0\ \gamma^p_\downarrow\ q'')\ \gamma'\ q)\ \gamma''\ q' \\
\mathcal{P}[\![\mathit{pushSubCont}\ e_1\ e_2]\!] & = & \lambda\kappa.\lambda\gamma''.\lambda q. \\
& & \quad \mathcal{P}[\![e_1]\!]\ (\lambda\gamma'.\lambda\gamma.\lambda q'. \\
& & \quad \mathcal{P}[\![e_2]\!]\ \kappa_0\ (\gamma'{+\!\!+}(\kappa:\gamma))\ q')\ \gamma''\ q \\
\\
\mathcal{V}[\![x]\!] & = & x \\
\mathcal{V}[\![\lambda x.e]\!] & = & \lambda x.\lambda\kappa'.\lambda\gamma'.\lambda q'.\mathcal{P}[\![e]\!]\ \kappa'\ \gamma'\ q'
\end{array}
$$

where, in the target language:

$$\kappa_0 = \lambda v.\lambda\gamma.\lambda q.\mathcal{K}(v,\gamma,q)$$

and:

$$
\begin{array}{rcl}
\mathcal{K}(v,[],q) & = & v \\
\mathcal{K}(v,p:\gamma,q) & = & \mathcal{K}(v,\gamma,q) \\
\mathcal{K}(v,\kappa:\gamma,q) & = & \kappa\ v\ \gamma\ q
\end{array}
$$

Fig. 2. CPS translation of call-by-value calculus with control

So far, Figure 2 yields for our operators a semantics that is similar in nature to the one that Moreau and Queinnec gave for *marker* and *call/pc*. We now refine it in two different ways. Figure 3 simplifies the CPS translation by η-reducing the equations in Figure 2 to eliminate arguments that are simply passed along. Pure λ-calculus terms have no need to access the metacontinuation or next prompt, and their Figure 3 translations reflect this fact. While the metacontinuation and next prompt are available at all times, they are ignored by the core terms and manipulated only by the control operators, a property shared with Danvy and Filinski's semantics for *shift* and *reset* (1990). The new variant of the CPS translation is equivalent to the original one since η-equations are part of the theory of the CPS target language.

Figure 4 takes the simplification one step further. The handling of core terms is as in Figure 3, but the portions of the control operator code that deal directly with the metacontinuation and next prompt have been split out into separate run-time combinators. These combinators are defined simply as target-language constants. The CPS translation itself now deals only with the issue of fixing evaluation order: the translation of core terms is the traditional one (c.f. Section 3.1), and the translation of the control operators expresses only their argument-evaluation properties. For example, the translation of *pushPrompt* says that e_1 is evaluated but e_2 is not, and the translation of *withSubCont* says that it is treated as a normal function

$$
\begin{array}{rcl}
\mathcal{P}[\![v]\!] & = & \lambda\kappa.\kappa\ \mathcal{V}[\![v]\!] \\
\mathcal{P}[\![e_1 e_2]\!] & = & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda f.\mathcal{P}[\![e_2]\!]\ (\lambda a.f\ a\ \kappa)) \\
\\
\mathcal{P}[\![\mathit{newPrompt}]\!] & = & \lambda\kappa.\lambda\gamma.\lambda q.\kappa\ q\ \gamma\ (q+1) \\
\mathcal{P}[\![\mathit{pushPrompt}\ e_1\ e_2]\!] & = & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda p.\lambda\gamma.\mathcal{P}[\![e_2]\!]\ \kappa_0\ (p:\kappa:\gamma)) \\
\mathcal{P}[\![\mathit{withSubCont}\ e_1\ e_2]\!] & = & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda p.\mathcal{P}[\![e_2]\!](\lambda f.\lambda\gamma.f\ (\kappa:\gamma^p_{\uparrow})\ \kappa_0\ \gamma^p_{\downarrow})) \\
\mathcal{P}[\![\mathit{pushSubCont}\ e_1\ e_2]\!] & = & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda\gamma'.\lambda\gamma.\mathcal{P}[\![e_2]\!]\ \kappa_0\ (\gamma'{+\!+}(\kappa:\gamma))) \\
\\
\mathcal{V}[\![x]\!] & = & x \\
\mathcal{V}[\![\lambda x.e]\!] & = & \lambda x.\lambda\kappa'.\mathcal{P}[\![e]\!]\ \kappa'
\end{array}
$$

Fig. 3. CPS translation of call-by-value calculus with control (η-reduced)

$$
\begin{array}{rcl}
\mathcal{P}[\![v]\!] & = & \lambda\kappa.\kappa\ \mathcal{V}[\![v]\!] \\
\mathcal{P}[\![e_1 e_2]\!] & = & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda f.\mathcal{P}[\![e_2]\!]\ (\lambda a.f\ a\ \kappa)) \\
\\
\mathcal{P}[\![\mathit{newPrompt}]\!] & = & \mathit{newPrompt}_c \\
\mathcal{P}[\![\mathit{pushPrompt}\ e_1\ e_2]\!] & = & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda p.\mathit{pushPrompt}_c\ p\ \mathcal{P}[\![e_2]\!]\ \kappa) \\
\mathcal{P}[\![\mathit{withSubCont}\ e_1\ e_2]\!] & = & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda p.\mathcal{P}[\![e_2]\!]\ (\lambda f.\mathit{withSubCont}_c\ p\ f\ \kappa)) \\
\mathcal{P}[\![\mathit{pushSubCont}\ e_1\ e_2]\!] & = & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda\gamma'.\mathit{pushSubCont}_c\ \gamma'\ \mathcal{P}[\![e_2]\!]\ \kappa) \\
\\
\mathcal{V}[\![x]\!] & = & x \\
\mathcal{V}[\![\lambda x.e]\!] & = & \lambda x.\lambda\kappa'.\mathcal{P}[\![e]\!]\ \kappa'
\end{array}
$$

where, in the target language:

$$
\begin{array}{rcl}
\mathit{newPrompt}_c & = & \lambda\kappa.\lambda\gamma.\lambda q.\kappa\ q\ \gamma\ (q+1) \\
\mathit{pushPrompt}_c & = & \lambda p.\lambda t.\lambda\kappa.\lambda\gamma.t\ \kappa_0\ (p:\kappa:\gamma)) \\
\mathit{withSubCont}_c & = & \lambda p.\lambda f.\lambda\kappa.\lambda\gamma.f\ (\kappa:\gamma^p_{\uparrow})\ \kappa_0\ \gamma^p_{\downarrow} \\
\mathit{pushSubCont}_c & = & \lambda\gamma'.\lambda t.\lambda\kappa.\lambda\gamma.t\ \kappa_0\ (\gamma'{+\!+}(\kappa:\gamma))
\end{array}
$$

Fig. 4. Factoring the control operations

application. (Indeed, if we introduced thunks into the interfaces of *pushPrompt* and *pushSubCont*, the CPS conversion algorithm would not need to deal with the control operators in any way, even superficially.) Hence, any program that makes use of delimited continuations can be evaluated by rewriting the program using a single, completely standard CPS conversion algorithm, and subsequently supplying additional "hidden" arguments—the metacontinuation and next prompt—and suitable implementations of our operators that manipulate those arguments. This use of hidden arguments is just an application of the standard technique of currying.

This final variant of the CPS translation is equivalent to the previous two by using $\beta\eta$-equations to abstract the uses of the run-time combinators. This establishes for the first time that a single, completely standard CPS transform suffices to implement a broad variety of delimited control operators.

3.5 Relating the Operational and CPS Semantics

Taking the operational semantics as our specification, we show that the CPS semantics is correct. In order to establish the result, we first extend the CPS translation to the additional syntactic constructs used to define the operational semantics:

$$\mathcal{P}[\![(e, D, E, q)]\!] \quad = \quad \mathcal{P}[\![e]\!](\lambda x.\mathcal{P}[\![D[x]]\!]\kappa_0)\mathcal{V}[\![E]\!]q$$

$$\begin{array}{rcl} \mathcal{V}[\![q]\!] & = & q \\ \mathcal{V}[\![[]]\!] & = & [] \\ \mathcal{V}[\![p : E]\!] & = & p : \mathcal{V}[\![E]\!] \\ \mathcal{V}[\![D : E]\!] & = & (\lambda x.\mathcal{P}[\![D[x]]\!]\kappa_0) : \mathcal{V}[\![E]\!] \end{array}$$

The transformation of values is extended to prompt names and sequences in a natural way, assuming the CPS target language includes lists. A delimited context D translates to a continuation $(\lambda x.\mathcal{P}[\![D[x]]\!]\kappa_0)$. In general, the translation of an expression $D[e]$ is equivalent to the translation of e in a continuation representing D.

Proposition 3.1
$\mathcal{P}[\![D[e]]\!] =_{\beta\eta} \lambda\kappa.\mathcal{P}[\![e]\!](\lambda x.\mathcal{P}[\![D[x]]\!]\kappa)$

Proof. By induction on D. □

As explained in the previous section, the equality on CPS terms consists of the $\beta\eta$ axioms plus the equations defining $\mathcal{K}(.,.,.)$ given in Figure 2. The main correctness result is easily stated and proved.

Proposition 3.2
If $(e, D, E, q) \longmapsto (e', D', E', q')$ then $\mathcal{P}[\![(e, D, E, q)]\!] =_{\beta\eta} \mathcal{P}[\![(e', D', E', q')]\!]$.

4 Monadic Semantics

The CPS semantics plays two complementary roles: it specifies the *order of evaluation* among subexpressions, and it specifies the *semantics of the control operators.*

The order of evaluation is important, because it directly affects the semantics of control effects. For example, adding *pushPrompt* as an ordinary function to a call-by-value language like Scheme or ML gives the wrong semantics, because in an expression such as *pushPrompt* e_1 e_2 the default parameter-passing mechanism would evaluate e_2 *before* invoking the *pushPrompt* operation. Since the whole point of *pushPrompt* is to introduce a prompt to which control operations in e_2 can refer, evaluating those control operations before pushing the prompt defeats the purpose of the operation. One solution is to treat the control operators as syntactic constructs, as we have done so far. Another is to use thunks to manually delay and force the evaluation of e_2 at the appropriate times (see for example the embedding of `reset` in ML by Filinski (1994)). In Scheme the use of thunks would typically be abstracted using the macro language, which is effectively equivalent to adding *pushPrompt* as a syntactic construct. In both cases, however, such encoding tricks distract from and complicate the semantic analysis.

$$
\begin{array}{llcl}
\text{Variables} & x, \ldots & & \\
& & & \\
\text{Terms} & e & ::= & x \mid \lambda x.e \mid e_1 e_2 \\
& & \mid & \mathit{return}\ e \mid e_1 \ggg\!= e_2 \\
& & \mid & \mathit{newPrompt} \mid \mathit{pushPrompt}\ e_1\ e_2 \\
& & \mid & \mathit{withSubCont}\ e_1\ e_2 \mid \mathit{pushSubCont}\ e_1\ e_2
\end{array}
$$

Fig. 5. Syntax of the monadic metalanguage

$$
\begin{array}{rcl}
\mathcal{T}[\![x]\!] & = & \mathit{return}\ x \\
\mathcal{T}[\![\lambda x.e]\!] & = & \mathit{return}\ (\lambda x.\mathcal{T}[\![e]\!]) \\
\mathcal{T}[\![e_1 e_2]\!] & = & \mathcal{T}[\![e_1]\!] \ggg\!= \lambda f.\mathcal{T}[\![e_2]\!] \ggg\!= \lambda a.f\ a \\
& & \\
\mathcal{T}[\![\mathit{newPrompt}]\!] & = & \mathit{newPrompt} \\
\mathcal{T}[\![\mathit{pushPrompt}\ e_1\ e_2]\!] & = & \mathcal{T}[\![e_1]\!] \ggg\!= \lambda p.\mathit{pushPrompt}\ p\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![\mathit{withSubCont}\ e_1\ e_2]\!] & = & \mathcal{T}[\![e_1]\!] \ggg\!= \lambda p.\mathcal{T}[\![e_2]\!] \ggg\!= \lambda f.\mathit{withSubCont}\ p\ f \\
\mathcal{T}[\![\mathit{pushSubCont}\ e_1\ e_2]\!] & = & \mathcal{T}[\![e_1]\!] \ggg\!= \lambda s.\mathit{pushSubCont}\ s\ \mathcal{T}[\![e_2]\!]
\end{array}
$$

Fig. 6. Monadic translation of call-by-value calculus with control

An alternative, and now well-established (Hatcliff & Danvy, 1994), technique is to express the order of evaluation by a translation $\mathcal{T}[\![e]\!]$ into a *monadic metalanguage* (Moggi, 1991). After this translation, the behaviour of the control operators can be expressed by defining them as constants, just as we did in Section 3.4. This separation allows us to study the issues related to the order of evaluation separately from the semantics of the control operators. More importantly it allows us later to introduce a monadic typing discipline to track and encapsulate the control effects.

4.1 A Monadic Metalanguage with Prompts and Continuations

The monadic translation $\mathcal{T}[\![e]\!]$ takes a source-language term to a term in a monadic metalanguage, whose syntax is given in Figure 5. The monadic metalanguage extends the λ-calculus with a monadic type constructor and associated operations. These operations include `return` and `>>=`, which explain how to sequence the effects in question, together with additional monad-specific operations. In our case, these operations are *newPrompt*, *pushPrompt*, *withSubCont*, and *pushSubCont*. The monadic metalanguage is typed, but because there are several ways to type the operations, we defer the type issues until Section 6.

The monadic translation is in Figure 6. For function applications and *withSubCont*, the effects of the operands are performed from left to right before the application. For *pushPrompt* and *pushSubCont*, only the effects of e_1 are performed before the application, while the effects of e_2 are performed after the prompt or the subcontinuation are pushed. The translation says nothing about the semantics of the control operators that appear in the target of the translation; it simply enforces the proper sequencing.

$$
\begin{array}{rcl}
\mathit{return}_k & = & \lambda t.\lambda\kappa.\kappa\ t \\
\gg\!=_k & = & \lambda t_1.\lambda t_2.\lambda\kappa.t_1\ (\lambda v.t_2\ v\ \kappa) \\
\\
\mathit{newPrompt}_k & = & \lambda\kappa.\lambda\gamma.\lambda q.\kappa\ q\ \gamma\ (q+1) \\
\mathit{pushPrompt}_k & = & \lambda p.\lambda t.\lambda\kappa.\lambda\gamma.t\ \kappa_0\ (p : \kappa : \gamma)) \\
\mathit{withSubCont}_k & = & \lambda p.\lambda f.\lambda\kappa.\lambda\gamma.f\ (\kappa : \gamma^p_\uparrow)\ \kappa_0\ \gamma^p_\downarrow \\
\mathit{pushSubCont}_k & = & \lambda\gamma'.\lambda t.\lambda\kappa.\lambda\gamma.t\ \kappa_0\ (\gamma' \mathbin{++} (\kappa : \gamma))
\end{array}
$$

Fig. 7. CPS instance of monadic metalanguage

4.2 Semantics of the Monadic Metalanguage

The monadic metalanguage can be instantiated with the continuation monad to produce terms in the same CPS target language of Section 3. The instantiation is given in Figure 7. The constructors *return* and $\gg\!=$ are given the standard definitions for the CPS monad (Moggi, 1991): they manipulate a concrete representation of the continuation but know nothing about the metacontinuation. The monad-specific control operations are identical to those defined in Figure 4.

4.3 Relating the CPS and Monadic Semantics

We have two distinct ways of translating a source term e to CPS: a direct translation $\mathcal{P}[\![e]\!]$ and a monadic translation $\mathcal{T}[\![e]\!]$ followed by an instantiation to the CPS monad, which we denote using $\mathcal{T}_k[\![e]\!]$. These two translations are equivalent.

Proposition 4.1
$\mathcal{P}[\![e]\!] =_{\beta\eta} \mathcal{T}_k[\![e]\!]$

Proof. By induction on the structure of e proceeding by cases. □

In summary, the CPS semantics of Figure 3 has been teased into two parts that can be studied (and implemented) independently. The aspects relevant to the order of evaluation are factored out in the translation to the monadic metalanguage; the monadic constructs are aware of the continuation but not the metacontinuation or the generation of new names; and the latter are manipulated exclusively by our control operators, which themselves do not manipulate the representation of the continuation.

5 Scheme Implementation

Given the specification of our control operators, we turn to the problem of implementing them in the context of real languages. Our development so far has been in an untyped framework, which makes it suitable as the basis of a Scheme implementation.

We actually present two Scheme implementations, which use `call/cc` to capture partial continuations directly and maintain the metacontinuation as a global variable. The first implementation in Section 5.1 is directly based on the semantics but does not handle tail recursion properly. This implementation effectively generalises Filinski's implementation of *shift* and *reset* using SML/NJ's *callcc* and a metacontinuation cell (Filinski, 1994) to our family of control operators.

The second implementation in Section 5.2 addresses the problem with tail recursion by taking advantage of *Chez Scheme*'s equality property for continuations.

Although not presented here, CPS-based versions of both implementations are included with the Scheme and Haskell described in this article at `http://www.cs.indiana.edu/~sabry/papers/CC_code.tar.gz`.

5.1 Basic Implementation

We begin by defining **`pushPrompt`** and **`pushSubCont`** as syntactic abstractions that expand into calls to `$pushPrompt` and `$pushSubCont`. In each case, the body is represented as a thunk to delay its evaluation:

```
(define-syntax pushPrompt
  (syntax-rules ()
    [(_ p e1 e2 ...)
     ($pushPrompt p (lambda () e1 e2 ...))]))

(define-syntax pushSubCont
  (syntax-rules ()
    [(_ subk e1 e2 ...)
     ($pushSubCont subk (lambda () e1 e2 ...))]))
```

Having dealt with the special syntactic forms, we need only implement the monadic metalanguage in Scheme, *i.e.*, implement the semantics given in Figure 7. It is possible to implement this semantics by passing around a continuation, a metacontinuation, and a global counter, or by using the computational effects of Scheme as follows. To access the partial continuation κ, we use `call/cc`, while the metacontinuation γ is always available as a global variable. New prompts are implemented as freshly allocated strings, which effectively replaces the global counter p with the implicit state of the memory management system.

Using this approach, nothing needs to be done to implement *return* and $\gg\!=$ since the effects are implicit; the four control operators are implemented as follows:

```
(define abort)
(define mk)

(define ($pushPrompt p th)
  (call/cc (lambda (k)
    (set! mk (PushP p (PushSeg k mk)))
    (abort th))))
```

```
(define (withSubCont p f)
  (let-values ([(subk mk*) (splitSeq p mk)])
    (set! mk mk*)
    (call/cc (lambda (k)
      (abort (lambda () (f (PushSeg k subk))))))))

(define ($pushSubCont subk th)
  (call/cc (lambda (k)
    (set! mk (appendSeq subk (PushSeg k mk)))
    (abort th))))

(define newPrompt (lambda () (string #\p)))
```

The first two definitions establish empty bindings for *abort* and *mk*; the value of *abort* and initial value of *mk* are established by `runCC` below. The code uses a datatype `Seq` (not shown) to represent metacontinuations, with three constructors `EmptyS`, `PushP` (push prompt), and `PushSeg` (push continuation segment). The procedures `appendSeq` and `splitSeq` perform simple operations on that datatype; `appendSeq` realises `++` from the semantics, while `splitSeq` implements $\gamma^p_\uparrow$ and $\gamma^p_\downarrow$ in one operation. The syntactic abstraction **`let-values`** is used to bind the two values ($\gamma^p_\uparrow$ and $\gamma^p_\downarrow$) returned by `splitSeq` to the variables *subk* and *mk∗*.

The main challenge is in setting up the top-level procedure `runCC` to define the procedure *abort*, which corresponds to the initial continuation κ_0 from the semantics. As the semantics specifies, the definition of `runCC` should provide the initial continuation, the initial metacontinuation, and the initial global counter. In our case this reduces to initialising the metacontinuation and capturing the base continuation. When applied this base continuation should inspect and invoke the metacontinuation:

```
(define (runCC th)
  (set! mk (EmptyS))
  (underflow ((call/cc (lambda (k)
                (set! abort k)
                (abort th))))))
```

The procedure *abort* accepts a thunk and thaws it in a base continuation that encapsulates only the call to `underflow` to avoid any unnecessary stack build-up (Dybvig & Hieb, 1989). The `underflow` procedure is used to apply the global metacontinuation *mk*:

```
(define (underflow v)
  (Seq-case mk
    [(EmptyS) v]
    [(PushP _ mk*) (set! mk mk*) (underflow v)]
    [(PushSeg k mk*) (set! mk mk*) (k v)]))
```

5.2 Proper Tail Recursion

The procedure below repeatedly captures and pushes an empty subcontinuation:

```
(define (tailtest)
  (let ([p (newPrompt)])
    (pushPrompt p
      (withSubCont p
        (lambda (s)
          (pushSubCont s (tailtest)))))))
```

In a properly tail-recursive implementation this test should run without any growth in a process's memory image. The implementation presented above does not treat tail recursion properly, since each **pushSubCont** of *s* adds a new (empty) subcontinuation onto the metacontinuation, and the metacontinuation grows without bound. The same comment applies to the operational semantics in Section 3.3, which always pushes D on E even if D is the empty context. It is a simple matter to recognise this situation in the semantics to avoid pushing empty contexts. In order to implement such an optimisation, however, the code must have some way to detect empty subcontinuations. In *Chez Scheme*, this is accomplished by comparing the current continuation against a base continuation using `eqv?`.[1] To do so, we modify `runCC` to reify the base continuation and store it in the variable *base-k*:

```
(define base-k)

(define (runCC th)
  (set! mk (EmptyS))
  (underflow
    (call/cc (lambda (k1)
      (set! base-k k1)
      ((call/cc (lambda (k2)
         (set! abort k2)
         (abort th))))))))
```

We then define a wrapper for the `PushSeg` constructor that pushes a continuation onto the stack only if it is not the base continuation:

```
(define (PushSeg/t k seq)
  (if (eqv? k base-k)
      seq
      (PushSeg k seq)))
```

This wrapper is used in place of `PushSeg` in the implementations of our control operators.

[1] Unfortunately, this makes the code unportable; to our knowledge, no other Scheme system allows `eqv?` to be used on continuations in this manner.

6 Towards a Typed Implementation: Monadic Types in Haskell

We now turn our attention to the problem of typing our control operators. In order to study the typing issues in a concrete setting, we implement the monadic metalanguage in Haskell. This implementation allows us to use advanced type features like interfaces, type classes, nested polymorphism, generalised abstract data types, and existentials.

So the plan is this. We will write programs directly in Haskell, in effect relying on the programmer to perform the monadic translation $\mathcal{T}[\![e]\!]$. Then we need only to provide a Haskell *library* that defines a continuation monad *CC*, together with its basic `return` and `>>=` operators, and its control operators `pushPrompt`, etc. The result is a typed, executable program that uses delimited continuations. It may not be an *efficient* implementation of delimited continuations, but it serves as an excellent design laboratory, as we will see in Section 7.

We found that static typing was unusually valuable in exploring this design space: the code is often short but tricky, and the type system found many bugs immediately. More fundamentally, as we explore in this section, the typed framework allows us to securely encapsulate algorithms that use control effects internally, but which are entirely pure when seen from the outside.

6.1 Monad with Fixed Observable Type

We first introduce the simplest types for the monadic library. Since Haskell does not provide a mechanism for defining interfaces, the following declaration is informal:

```
module SimpleCC where

data CC a           -- Abstract
instance Monad CC

data Prompt a       -- Abstract
data SubCont a b    -- Abstract

type Obs = ...      -- Arbitrary but fixed
runCC       ::  CC Obs → Obs

newPrompt   ::  CC (Prompt a)
pushPrompt  ::  Prompt a → CC a → CC a
withSubCont ::  Prompt b → (SubCont a b → CC b) → CC a
pushSubCont ::  SubCont a b → CC a → CC b
```

The interface includes the type constructor *CC*. The idea is that a term of type (*CC* `a`) is a computation that may have control effects. When the computation is run, the control effects are performed, and a result of type `a` is returned. The type *CC* must be an instance of the class *Monad*; that is, it must be equipped with the operators `return` and `>>=`:

```
return :: a → CC a
(>>=)  :: CC a → (a → CC b) → CC b
```

We saw in Section 4.1 how these two operators are used.

The interface also includes two abstract type constructors for prompts *Prompt* and subcontinuations *SubCont*. The type *Prompt* a is the type of prompts to which a value of type a can be returned, *i.e.*, if p of type *Prompt* a is pushed on the stack, then the only values that can flow to and from this stack frame are values of type a. The type *SubCont* a b is the type of subcontinuations to which a value of type a can be passed and which return a value of type b.

Following conventional continuation semantics, the interface defines an arbitrary but fixed type *Obs*, the type of observables; a complete sub-program that uses continuations must have type *CC Obs*. To execute such a program, the function runCC takes a computation which returns a value of the fixed type *Obs* and supplies it with the initial context (initial continuation, metacontinuation, and counter for prompt names) to get the final observable value.

The types of the control operators are a monadic variant of the types given by Gunter *et al.* (1995) for the similar operators. Each occurrence of newPrompt generates a new prompt of an arbitrary but fixed type a. The type of pushPrompt shows that a prompt of type *Prompt* a can only be pushed on a computation of type *CC* a which expects a value of type a. If the type of withSubCont p f is *CC* a then the entire expression returns a value of type a to its continuation; the continuation is assumed to contain a prompt p of type *Prompt* b; the portion of the continuation spanning from a to b is captured as a value of type *SubCont* a b which is passed to f. Since the remaining continuation expects a value of type b, the return type of f is *CC* b. A similar scenario explains the type of pushSubCont.

Wadler (1994) studies several systems of monadic types for composable continuations. His first system is similar to the one we consider in this section. Written in our notation, the types he considers for the operators are:

```
runCC       ::  CC Obs → Obs
pushPrompt  ::  CC Obs → CC Obs
withSubCont ::  (SubCont a Obs → CC Obs) → CC a
pushSubCont ::  SubCont a Obs → CC a → CC Obs
```

Indeed our interface reduces to the above, if we remove the ability to generate new prompts and use one fixed and *implicit* prompt of the observable type instead.

Wadler (1994) considers two other, more general, type systems for composable continuations. The second system, called "Murthy types" (Murthy, 1992), parameterises the type *CC* by the answer type. We will present a generalisation of this system in Section 6.3. The third system presented, called "Danvy and Filinski types" (Danvy & Filinski, 1989), is the most expressive but unfortunately is not a monad and hence will not be considered further.

6.2 Examples

The following short examples aim to give a little more intuition for the monadic interface. The examples use the following Haskell conventions:

- A λ-expression is written `λx → e`, and extends as far to the right as possible.
- We make heavy use of the right-associating, low-precedence infix application operator `$`, defined as `f $ x = f x`. Its purpose is to avoid excessive parentheses; for example, instead of `(f (g (h x)))` we can write `(f $ g $ h x)`.
- Haskell provides convenient syntax, known as "**do**-notation" for composing monadic expressions. For example:

  ```
  do { x1 ← e1 ;
       x2 ← e2 ;
       return (x1+x2) }
  ```

  is equivalent to

  ```
  e1 >>= (λx1 → e2 >>= (λx2 → return (x1+x2)))
  ```

  The evaluation of the expression (in either notation) first executes `e1` and its control effects. The value returned by the execution of `e1` is bound to `x1` and then the same process is repeated with `e2`. Finally, the sum `x1+x2` is returned. A sequence of monadic computations is usually expressed using **do**-notation but we occasionally use the bind operator `>>=` explicitly.
- Whitespace (instead of semi-colons) is often used as a separator of monadic actions with indentation (instead of braces) indicating grouping.

Given the above conventions, the term:

```
withSubCont p $ λk →
pushSubCont k $
do x ← do y1 ← e1
          e2
   e
```

parses as:

```
withSubCont p (λk →
  pushSubCont k (do { x ← (do { y1 ← e1; e2}); e}))
```

which in turn is equivalent to:

```
withSubCont p (λk →
  pushSubCont k ((e1 >>= λy1 → e2) >>= λx → e)
```

We first revisit our examples with the top-level prompt p_0 and `callcc` from Section 2.2. The top-level prompt would have type *Prompt Obs* and the definitions of `abort` and `callcc` would be typed as follows:

```
abort :: CC Obs → CC a
abort e = withCont (λ _ → e)

callcc :: ((a → CC b) → CC a) → CC a
callcc f = withCont $ λk →
             pushSubCont k $
             f (λv → abort (pushSubCont k (return v)))
```

As expected the type of `abort` refers to the top level type of observables. The type of `callcc` makes explicit that the argument of a continuation must be a *value*, of type `a`, rather than a computation of type *CC* `a`. This interface of `callcc` can lead to space leaks, however. For example, consider:

```
loop :: Int → CC Int
loop 0 = return 0
loop n = callcc (λk → do { r ← loop (n-1); k r })
```

When the recursive call to `loop (n-1)` returns, the continuation `k` is invoked, which abandons the entire current stack, using the call to `abort` inside the definition of `callcc`. So the recursive call to `loop` takes place on top of a stack that will never be used. If the recursive call increases the size of the stack before looping, as is the case here, the result is that the stack grows proportional to the depth of recursion.

The usual solution to this problem is to thunkify the argument to the escape procedure, passing a value of type `(() → a )` instead of a value of type `a` (Dybvig & Hieb, 1989). In our monadic framework, we can be more explicit by defining `callcc` as follows:

```
callcc :: ((CC a → CC b) → CC a) → CC a
callcc f = withCont $ λk →
             pushSubCont k $
             f (λc → abort (pushSubCont k c))
```

where it is explicit that the continuation is applied to a *computation* of type *CC* `a`. Using the new variant of `callcc` we can write our `loop` example as follows:

```
loop :: Int → CC Int
loop 0 = return 0
loop n = callcc (λk → k (loop (n-1)))
```

Now the context is aborted before the recursive call to `loop` is made, and the function becomes properly tail-recursive.

6.3 Encapsulation

The monadic interface we have considered so far has the advantage of being simple, but it has a major limitation: since the encapsulation operator `runCC` specifies a fixed type *Obs*, every encapsulated computation that internally uses control operators is limited to return the same fixed type. One really wants to be able to write monadic computations that return values of arbitrary types and that can be used in arbitrary contexts. Naïvely replacing *Obs* by an arbitrary type is however unsound as it would allow interactions among control operations executing under different occurrences of `runCC`.

The problem can be explained as follows. As the semantics shows, `withSubCont` must essentially search the metacontinuation until it finds a pushed prompt that is identical to its prompt argument. A straightforward implementation of this search would use the internal representation of the prompts ignoring their types. Clearly the soundness of the idea rests on the requirement that prompts of different types never have the same internal representation. This requirement is violated if we naïvely relax the type of `runCC`. For example, even though it is impossible to realize in Haskell, we can easily imagine a hypothetical implementation of `runCC` with the type:

```
runCC :: CC a → a
```

Such an implementation would admit the following program:

```
abortP p e = withSubCont p (λ _ → e)

badc = let p1 :: Prompt Int  = runCC newPrompt
           p2 :: Prompt Bool = runCC newPrompt
       in 1 + runCC (pushPrompt p1 (abortP p2 (return True)))
```

Because it has a pure type, the result of `runCC e` for any `e` must be a *pure* expression without any side effects. In particular the two occurrences of `runCC` in the right-hand sides of the **let** cannot interact via a global symbol table or anything similar to guarantee that they return distinct prompts `p1` and `p2`. Therefore, nothing forces the two prompts `p1` and `p2` to have different internal representations. In the case when they do have the same representation, *i.e.*, they are intensionally equal, the jump to `p2` reaches `p1` instead, which causes the evaluation to add `1` to *True*.

This problem has actually been identified in the early implementations of the SML/NJ top-level interactive loop (Harper *et al.*, 1993); its solution then was to associate every prompt with a different time-stamp and to dynamically check that a continuation captured within a given time-stamp does not jump to a prompt with a different time-stamp. The solution we propose to this type soundness problem is to confine the control effects to certain *regions*. (This is also desirable from the perspective of programming methodology. For a longer discussion of this point, we refer the reader to the arguments leading to the design of *spawn* (Hieb & Dybvig, 1990).) As Thielecke (2003) recently showed, there is an intimate relation between regions and the type of observables. In particular:

- A region determines a local answer type for continuations
- Pure expressions are parametrically polymorphic in their answer types

This situation is similar to the well-understood situation of encapsulating state in Haskell (Launchbury & Peyton Jones, 1995), and our Haskell encoding is quite similar. We add a *region parameter* `ans`, which specifies the answer type produced by the region, to every type constructor and enforce non-interference and localisation of control actions by using rank-2 polymorphism. The refined interface becomes:

```
module CC where

data CC ans a           -- Abstract
data Prompt ans a       -- Abstract
data SubCont ans a b    -- Abstract

instance Monad (CC ans)

runCC        ::  (∀ ans. CC ans a) → a
newPrompt    ::  CC ans (Prompt ans a)
pushPrompt   ::  Prompt ans a → CC ans a → CC ans a
withSubCont  ::  Prompt ans b → (SubCont ans a b → CC ans b)
                 → CC ans a
pushSubCont  ::  SubCont ans a b → CC ans a → CC ans b
```

In the new interface, the types `CC`, `Prompt`, and `SubCont` are each given an additional type parameter `ans` which represents their *control region* as far as the type system is concerned. The type of each operator insists that its arguments and results come from a common region. So, for example, one cannot push a prompt of type `Prompt String a` if the current computation has type `CC Bool a`. The type of `runCC` shows that it takes an effectful computation, of type `CC ans a`, runs it, and returns an ordinary, pure, value of type `a`. This encapsulation is enforced by giving `runCC` a rank-2: its argument must be polymorphic in the region `ans`. Rank-2 types are an extension of the Haskell standard that is supported by several implementations: the concrete syntax for a type `∀ r.t` is `forall r. t`.

6.4 Examples

Encapsulation using `runCC` provides a convenient way to isolate regions of control from each other. If a computation labelled by a region parameter `ans1` pushes a prompt, then a computation labelled by a different type `ans2` cannot access that prompt and hence cannot abort or duplicate computations up to that prompt. Moreover the type system enforces this restriction: there is no way for the prompt to somehow leak using a global reference or higher-order function.

The following two expressions can be encapsulated either because they perform no effects at all (`g0`), or because their effects are completely localised and hence invisible to the outside world (`g1`):

```
g0 = 1 + runCC (do x ← return 1; return (x+1))
g1 = 1 + runCC (do  p ← newPrompt
                    pushPrompt p $
                      withSubCont p $ λ sk →
                      pushSubCont sk (pushSubCont sk (return 2)))
```

A more interesting example of encapsulation uses continuations in a way similar to exceptions, to abort several recursive calls as an optimisation. The control effect is completely localised and hence encapsulated:

```
productM :: [Int] → Int
productM xs = runCC (do  p ← newPrompt
                         pushPrompt p (loop xs p))
  where  loop []      p  =  return 1
         loop (0:_)   p  =  abortP p (return 0)
         loop (x:xs)  p  =  do r ← loop xs p; return (x*r)
```

In the example, we recursively traverse a list pushing multiplication frames on the stack. Before starting the loop, we push a prompt on the stack to mark the point at which the recursion will eventually return. If we reach the end of the list, we return normally, performing the multiplications on the way back. If we encounter a `0`, however, we simply erase the pending multiplication frames and return the final result `0` to the caller of `loop`.

Expressions may violate encapsulation for a variety of reasons:

```
b0 = runCC (do p ← newPrompt; return p)

b1 = do p ← newPrompt
        pushPrompt p $
          withSubCont p $ λ sk →
          return (runCC (pushSubCont sk (return 1)))
```

Example `b0` attempts to export a local prompt, while `b1` attempts to use a subcontinuation captured from outside its region. Both are invalid, and are rejected by the type checker.

Well-typed expressions may still cause run-time errors as the type system does not keep track of how many times a prompt is pushed, which would be necessary to guarantee that every dynamic control action is matched by a pushed prompt. For example, in the first example, the prompt is pushed zero times and hence the control action is invalid. In the second example, the prompt is pushed once, but the code attempts to capture the continuation up to this prompt twice and hence also fails with an error:

```
r0 = runCC (do p ← newPrompt
               abortP p (return 5))

r1 = runCC (do p ← newPrompt
               pushPrompt p $
                 do x ← withSubCont p (λ_ → abortP p (return 5))
                    return 0)
```

Filinski (1999) proposes a type-and-effect system for layered monadic effects that guarantees statically that well-typed programs do not fail with such 'missing prompt" errors. Alternatively, one can express the needed invariants using *contracts* in the module interface as argued by Findler *et al.* (2001).

7 Haskell Implementation

We can now turn the semantics of Figure 7 into a *typed and executable* specification, by giving three different implementations of the *CC* library whose interface we presented in Section 6.3. (We focus on the more interesting interface in Section 6.3; implementing the other basic interface in Section 6.1 is simpler.) Each of the three implementations represent continuations as sequences of *control segments* and prompts. The first represents control segments as individual frames, the second as CPS functions, and the third as abstract continuation objects obtained via a variant of `callcc`. Providing these three typed implementations serves several purposes:

- It clarifies some of the informal arguments we made about the separation of concerns between the continuation, the metacontinuation, and the generation of prompts. Indeed we show that it is possible to focus on each aspect in a separate module.
- The semantics in Sections 3 and 4 are quite complex and the types are non-trivial. We found the executable Haskell specification to be invaluable in debugging the semantic definitions.
- The first Haskell implementation can be seen as an abstract specification of how to implement our framework by directly modifying the run-time system, manipulating individual stack frames and stack fragments.
- The second Haskell implementation can be seen as an abstract specification of how to implement our framework in a CPS compiler, which already has an explicit and functional representation of continuations.
- The third Haskell implementation can be seen as an abstract specification of how to implement our framework in a source level library that builds on top of control operators such as `callcc` that give access to the continuation.

These source-level implementations are intended to be illustrative rather than efficient. An efficient implementation can be created at a lower level by representing control segments as contiguous blocks of stack frames, as in the efficient implementation of traditional continuations described by Hieb *et al.* (1990).

The implementation uses several modules, whose inter-dependencies are given in Figure 8. The client program can import any of the three implementations of the *CC* monad described below in detail. All implementations of the *CC* monad import two helper modules for implementing sequences and for generating prompt names. The last implementation of *CC* (with reified continuations) also imports a module that manipulates an underlying "native" representation of continuations that is only available via control operators. The implementations use several constructs that are not part of Haskell 98, although they have become standard extensions to the basic language. In particular, we use generalised abstract data types (GADTs) (Xi *et al.*, 2003; Cheney & Hinze, 2002; Pottier & Régis-Gianas, 2006) to express the typing of a sequence of function (continuation) compositions, in Section 7.2 and to give more precise types to some of the constructors. We also use *higher-rank universal types* to encapsulate control effects (in the typing of `runCC` in Sections 7.3-7.5),

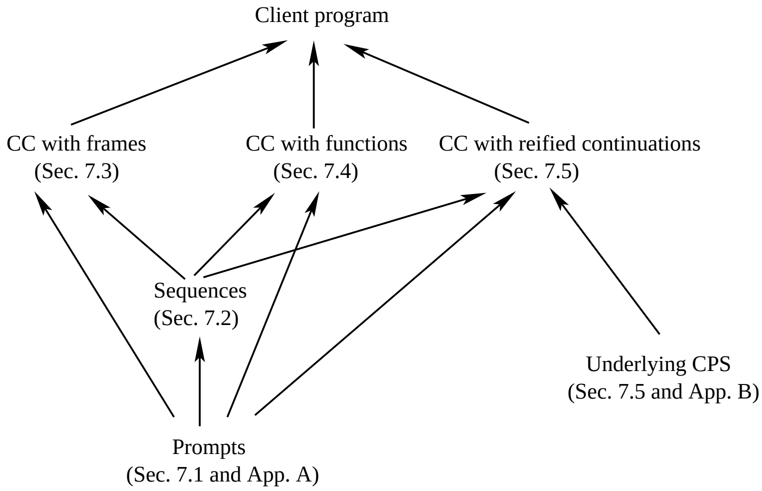

Fig. 8. Module dependencies

and to capture an invariant related to continuations and metacontinuations (in the definiton of *CC* in Sections 7.4 and 7.5).

7.1 Generating Prompts

The module *Prompt* implements the dynamic generation of prompts. It is isolated here because the issue of name generation is independent of continuations, and because it allows us to isolate in this small module the only unsafe (in the sense that the type system cannot prove it safe) coercion in our code.[2] The coercion allows us to safely implement `withSubCont`, which compares the internal representations prompts of different types.

```
module Prompt where

newtype Prompt ans a  -- Abstract
newtype P ans a       -- Abstract

instance Monad (P ans)

data Equal a b where
  EQUAL     :: Equal a a
  NOT_EQUAL :: Equal a b

runP           ::  P ans ans → ans
```

[2] In a private communication, Oleg Kiselyov suggests an alternative approach in which every prompt is associated with a built-in reference cell. The resulting implementation includes no coercions but is however much removed from the semantics we propose so we do not consider it further in this article and leave a detailed comparison to future work.

```
newPromptName  ::  P ans (Prompt ans a)
eqPrompt       ::  Prompt ans a → Prompt ans b → Equal a b
```

The module provides the abstract type of prompts and a monad (*P* ans), which constrains accesses to the prompt supply to guarantee that generated prompts are globally unique within all computations tagged by the type parameter of the region ans. The prompts generated in different regions may have the same internal representation but the encapsulation mechanism for regions described in Section 6.3 prevents such prompts from being used simultaneously. The implementation of the module is in Appendix A.

The operation runP takes a computation that uses prompts, provides the initial supply of prompts, and discards the supply of prompts from the final result. The type *Equal* is a GADT that gives evidence for the equality of two types as shown by the type of the constructor *EQUAL*. The operation eqPrompt compares two prompts of possibly *different* types by looking at their internal representation: if the two prompts are equal then the function produces evidence that their types must also be equal. This is safe assuming that our implementation is correct and that the prompt values are unforgeable. Since these facts are not checkable by the type system, the implementation of eqPrompt includes the single use of an unsafe, implementation-dependent, coercion in our code.

The module *Prompt* is imported in each of the following implementations of the *CC* interface.

7.2 Sequences

All of our implementations represent continuations as sequences of control segments and prompts. The control segments are single frames in the first implementation and partial continuations in the second and third implementations. In the second implementation, partial continuations are represented by functions, while in the third implementation, partial continuations are abstract continuations with an unknown representation. We provide here a general sequence type that can be instantiated for each case. The three operations we require on sequences (appendSubSeq, pushSeq, and splitSeq) need to be defined only once.

The interface of the module *Seq* is given below:

```
module Seq where

data Seq contseg ans a where
    EmptyS :: Seq contseg ans ans
    PushP :: Prompt ans a → Seq contseg ans a → Seq contseg ans a
    PushSeg :: contseg ans a b → Seq contseg ans b
               → Seq contseg ans a

type SubSeq contseg ans a b      -- Abstract
```

```
appendSubSeq :: SubSeq contseg ans a b
             → SubSeq contseg ans b c
             → SubSeq contseg ans a c

pushSeq :: SubSeq contseg ans a b → Seq contseg ans b
        → Seq contseg ans a

splitSeq :: Prompt ans b → Seq contseg ans a →
            (SubSeq contseg ans a b, Seq contseg ans b)
```

The basic structure of the *Seq* type is that of a list, which can be empty (*EmptyS*) or one of two "cons" variants each with another *Seq* in the tail. We first give the intuitive meaning of each type parameter and then explain the various constructors in turn. The type parameter `a` represents the type of values received by the aggregate sequence of prompts and segments. The type parameter `ans` is used to identify the region of control to which the prompts and control segments belong. The type parameter `contseg` is the abstract constructor of control segments that is varied to produce the various implementations.

The type *Seq* is also a GADT. The constructor *EmptyS* has a type that unifies `a` and `ans`. The *PushP* constructor is simple: it simply pushes a (suitably typed) prompt onto the sequence. The *PushSeg* constructor pushes a control segment representing either an individual frame or a partial continuation. When searching for prompts in sequences, we never need to inspect control segments so the precise details of what constitutes a segment is not relevant at this point. The type of *PushSeg* includes a parameter `b` which is *existentially quantified*. It is used here to express the fact that if the control segment takes a value of type `a` to one of some type `b`, and the rest of the sequence takes a value of that type `b`, then the composition of the two takes a value of type `a` irrespective of what `b` is.

The operations to split and append sequences are defined below. To split the sequence at a given prompt, we traverse it, comparing the prompts along the way. If a prompt matches the desired prompt, we use the `eqPrompt` function to obtain a coercion that forces the types to be equal.

```
type SubSeq contseg ans a b =
  Seq contseg ans b → Seq contseg ans a

emptySubSeq :: SubSeq contseg ans a a
emptySubSeq = id

appendSubSeq = (.)

pushSeq = ($)

splitSeq p EmptyS
  = error ("Prompt was not found on the stack")
```

```
  splitSeq p (PushP p' sk)
    = case eqPrompt p' p of
        EQUAL     → (emptySubSeq, sk)
        NOT_EQUAL → case splitSeq p sk of
  splitSeq p (PushSeg seg sk)
    = case splitSeq p sk of
        (subk,sk') → (appendSubSeq (PushSeg seg) subk, sk')
```

7.3 Continuations as Sequences of Frames

Now that we are equipped with utility libraries `Prompt` and `Seq`, we are ready to write our first implementation of the `CC` library itself. In this first implementation, the continuation and metacontinuation are merged into one data-structure that consists of a sequence of frames and prompts. A frame of type `Frame ans a b` is a function that given a value of type `a` returns a `b`-computation which performs the next computation step. The continuation is a sequence of these frames and prompts, which consumes values of type `a` and returns a computation that delivers the value of type `ans` after possibly generating prompts. Thus the complete definitions of the datatypes are:

```
newtype Frame ans a b = Frame (a → CC ans b)

type Cont ans a = Seq Frame ans a
type SubCont ans a b = SubSeq Frame ans a b

newtype CC ans a = CC (Cont ans a → P ans ans)
unCC (CC c) = c
```

Given these data types, here is how we make `CC` an instance of the `Monad` class, by implementing `return` and `(>>=)`:

```
instance Monad (CC ans) where
  return v = CC (λk → appk k v)
  (CC e1) >>= e2 = CC (λk → e1 (PushSeg (Frame e2) k))

-- Applies a control segment
appseg :: Frame ans a b → a → CC ans b
appseg (Frame fr) a = fr a

-- Applies a continuation
appk :: Cont ans a → a → P ans ans
appk EmptyS a = return a
appk (PushP _ k') a = appk k' a
appk (PushSeg seg k') a = unCC (appseg seg a) k'
```

```
-- Runs a CPS term
runTerm :: CC ans a → P ans ans
runTerm c = unCC c emptyS

runCC :: (∀ ans. CC ans a) → a
runCC ce = runP (runTerm c)
```

The function `appk` serves as an interpreter that transforms a sequence *data structure*, of type *Cont* `ans a`, into a *function*.

The implementation of the four control operators is now straightforward:

```
newPrompt :: CC ans (Prompt ans a)
newPrompt = CC (λk → do p ← newPromptName; appk k p)

pushPrompt :: Prompt ans a → CC ans a → CC ans a
pushPrompt p (CC e) = CC (λk → e (PushP p k))

withSubCont :: Prompt ans b → (SubCont ans a b → CC ans b)
              → CC ans a
withSubCont p f =
    CC (λk → let (subk,k') = splitSeq p k
              in unCC (f subk) k')

pushSubCont :: SubCont ans a b → CC ans a → CC ans b
pushSubCont subk (CC e) = CC (λk → e (pushSeq subk k))
```

As already apparent in the continuation semantics, the only control operator that is aware of the prompt supply is `newPrompt`.

This implementation is properly tail-recursive (in the sense of Section 5.2). The only segments that are pushed are in the implementation of `>>=`, *i.e.*, the only segments that are pushed are segments corresponding to user code. However continuation segments may unnecessarily accumulate without bound because some of our operations are non-strict in the continuation. For example, the implementation of `pushPrompt` is not strict in its continuation argument; changing it to:

```
pushPrompt p (CC e) = CC (λk → k `seq` e (PushP p k))
```

improves the space usage in several situations.

While it offers a straightforward mechanism for adding support for our control operators to an existing run-time system, this implementation requires each control operation to loop through sequences of individual frames, as in Gasbichler and Sperber's direct implementation of shift and reset (Gasbichler & Sperber, 2002), an inefficiency avoided by our other two implementations.

7.4 Continuations as Functions

This implementation is identical to the semantics given in Figure 7. The continuation is represented as a function from values to *metaCPS terms*. MetaCPS terms are

CPS terms that accept metacontinuations and deliver answers. Metacontinuations are represented as sequences of continuations and prompts. The type definitions are:

```
newtype Cont ans a b = Cont (a → MC ans b)
unCont (Cont k) = k

type MetaCont ans a = Seq Cont ans a
type SubCont ans a b = SubSeq Cont ans a b

newtype CC ans a = CC (∀ b. Cont ans a b → MC ans b)
unCC (CC c) = c

newtype MC ans b = MC (MetaCont ans b → P ans ans)
unMC (MC m) = m
```

The type `b` used as an articulation point between the continuation and metacontinuation is completely arbitrary and hence universally quantified. This quantification captures an invariant that the interface between a continuation and a metacontinuation is arbitrary as long as they agree on it.

The type declarations can be explained in a more uniform way by observing that they can be generated by applying the continuation monad transformer to the metaCPS terms. In more detail, consider the continuation monad transformer:

```
newtype ContT m a = ContT (∀ b. (a → m b) → m b)
```

Applying this transformer to the type `(MC ans)` produces:

```
ContT (∀ b. (a → MC ans b) → MC ans b)
```

which is isomorphic to the representation of *CC*:

```
CC (∀ b. Cont ans a b → MC ans b)
```

The *CC* type provides the monadic combinators `return` and `>>=`:

```
instance Monad (CC ans) where
  return e = CC (λ (Cont k) → k e)
  (CC e1) >>= e2 = CC (λk → e1 (Cont (λv1 → unCC (e2 v1) k)))
```

The code above shows that the *CC* monad is a completely standard continuation monad: in particular the monadic combinators (and hence the translation of pure functions and applications) knows nothing about the metacontinuation.

To run a complete computation, we must of course provide an initial continuation and metacontinuation. The function `runCC` takes a computation and supplies it with the initial continuation; this returns another computation, which expects the initial metacontinuation.

```
-- Applies a control segment (a first level continuation)
appseg :: Cont ans a b → a → MC ans b
```

```
appseg (Cont k) a = k a

-- Runs a first level CPS term
runOne :: CC ans a → MC ans a
runOne c = unCC c initk
  where initk = Cont (λa → MC (λmk → appmk mk a))

-- Applies a second level continuation
appmk :: MetaCont ans a → a → P ans ans
appmk EmptyS a = return a
appmk (PushP _ mk') a = appmk mk' a
appmk (PushSeg k mk') a = unMC (appseg k a) mk'

-- Runs a second level CPS term
runTwo :: MC ans ans → P ans ans
runTwo c = unMC c emptyS

runCC :: (∀ ans. CC ans a) → a
runCC ce = runP (runTwo (runOne ce))
```

The initial continuation `initk` applies the metacontinuation via `appmk`, which in turn uses `appseg` to apply an individual control segment.

The exported operators are now implemented as follows:

```
newPrompt :: CC ans (Prompt ans a)
newPrompt = CC (λk → MC (λmk → do p ← newPromptName
                                    unMC (appseg k p) mk))

pushPrompt :: Prompt ans a → CC ans a → CC ans a
pushPrompt p e =
    CC (λk → MC (λmk →
      unMC (runOne e) (PushP p (PushSeg k mk))))

withSubCont :: Prompt ans b → (SubCont ans a b → CC ans b)
               → CC ans a
withSubCont p f =
    CC (λk → MC (λmk →
       let (subk,mk') = splitSeq p mk
           e = f (appendSubSeq (PushSeg k) subk)
       in unMC (runOne e) mk'))

pushSubCont :: SubCont ans a b → CC ans a → CC ans b
pushSubCont subk e =
    CC (λk → MC (λmk →
       unMC (runOne e) (pushSeq subk (PushSeg k mk))))
```

This implementation is not tail-recursive as there are several occurrences of the constructor *PushSeg* that could push the identity continuation.

7.5 Continuations Reified by a Control Operator

This third implementation even more clearly formalises the separation of concerns between continuation and metacontinuation: it uses *two* CPS monads: the underlying monad *CPS.M*, which manipulates a concrete representation of the continuation *CPS.K*, and the main monad implementing the *CC*-interface. The representation of the continuation *CPS.K* is hidden from the latter monad. The main monad must treat the type *CPS.K* as an abstract type: it can only capture and invoke the continuation manipulated by the underlying monad. In principle we could use `callcc` to capture the continuation like we did in Section 5 with the Scheme implementation. However this requires an initialisation phase to set up the initial continuation and bind it to a globally accessible `abort` procedure. It is more elegant in this case to use Felleisen's $\mathcal{C}$ operator (1987a), which includes the `abort` action as part of its definition.

First we assume we are given an underlying CPS monad with the following signature:

```
module CPS where

newtype K ans a     -- Abstract
newtype M ans a     -- Abstract

instance Monad (M ans)

throw :: K ans a → M ans a → M ans b
c     :: (K ans a → ans) → M ans a

runM  :: M ans ans → ans
```

The control operator `c` gives access to the continuation, which is an abstract type and aborts to the top level at the same time. The only thing we can do with this continuation is to invoke it using `throw`. A computation involving `c` and `throw` can be performed using `runM` to return its final answer. The implementation of this monad is standard and is included in Appendix B.

Given this underlying CPS monad, we implement *CC* as follows:

```
newtype Cont ans a b = Cont (CPS.K (MC ans b) a)
unCont (Cont k) = k

type MetaCont ans a = Seq Cont ans a
type SubCont ans a b = SubSeq Cont ans a b

newtype CC ans a = CC (∀ b. CPS.M (MC ans b) a)
```

```
unCC (CC c) = c

newtype MC ans b = MC (MetaCont ans b → P ans ans)
unMC (MC m) = m
```

The type *CC* is simply a wrapper for *CPS.M* and its monadic operations are identical to the ones of *CPS.M* modulo some tagging and un-tagging of the values:

```
instance Monad (CC ans) where
  return e = CC (return e)
  (CC e1) >>= e2 = CC (do v1 ← e1; unCC (e2 v1))
```

When run, an underlying *CPS.M* computation must inspect the metacontinuation and should return only when the sequence is empty. Hence every *CPS.M* evaluation starts with an underflow frame that inspects the stack:

```
-- Applies a control segment (a first level continuation)
appseg :: Cont ans a b → a → MC ans b
appseg (Cont k) a = CPS.runM (CPS.throw k (return a))

-- Runs a first level CPS term
runOne :: CC ans a → MC ans a
runOne m = CPS.runM (do a ← unCC m; initkF a)
  where initkF a = return (MC (λmk → appmk mk a))

-- Applies a second level continuation
appmk :: MetaCont ans a → a → P ans ans
appmk EmptyS a = return a
appmk (PushP _ mk') a = appmk mk' a
appmk (PushSeg k mk') a = unMC (appseg k a) mk'

-- Runs a second level CPS term
runTwo :: MC ans ans → P ans ans
runTwo c = unMC c emptyS

runCC :: (∀ ans. CC ans a) → a
runCC ce = runP (runTwo (runOne ce))
```

The exported operations are now implemented as follows:

```
newPrompt :: CC ans (Prompt ans a)
newPrompt =
    CC (CPS.c (λk → MC (λmk →
      do p ← newPromptName
         unMC (appseg (Cont k) p) mk)))
```

```
pushPrompt :: Prompt ans a → CC ans a → CC ans a
pushPrompt p e =
    CC (CPS.c (λk → MC (λmk →
        unMC (runOne e) (PushP p (PushSeg (Cont k) mk)))))

withSubCont :: Prompt ans b → (SubCont ans a b → CC ans b)
               → CC ans a
withSubCont p f =
    CC (CPS.c (λk → MC (λmk →
        let (subk,mk') = splitSeq p mk
            e = f (appendSubSeq (PushSeg (Cont k)) subk)
        in unMC (runOne e) mk')))
pushSubCont :: SubCont ans a b → CC ans a → CC ans b
pushSubCont subk e =
    CC (CPS.c (λk → MC (λmk →
        unMC (runOne e) (pushSeq subk (PushSeg (Cont k) mk)))))
```

This implementation of our control operators generalises previous direct-style implementations of *shift* and *reset* (Filinski, 1994; Filinski, 1996) as well as implementations of $\mathcal{F}$ and prompt (Sitaram & Felleisen, 1990a).

This implementation is also not tail-recursive: it can only be made tail-recursive if the underlying *CPS* module provides a primitive for recognising the identity continuation or, as we illustrated in the Scheme code (Section 5), a primitive for pointer equality of continuations.

8 Conclusions and further work

We have presented a typed monadic framework in which one can define and experiment with control operators that manipulate delimited continuations. This framework offers several advantages over previous work:

- It provides a set of basic building blocks that easily model the most common control operators from the literature (Section 2).
- It provides a clear separation of several entangled issues that complicate the semantics of such control operators: non-standard evaluation order, manipulation and representation of the continuation, manipulation and representation of the metacontinuation, and generation of new prompt names.
- It is strongly typed and allows one to encapsulate control effects to local regions of control (Section 6.3).
- It can be implemented on top of any traditional implementation of continuations, including a single, standard CPS translation (Sections 5 and 7).

As shown in Sections 5 and 7, our framework is implemented both in Scheme and Haskell, providing executable specifications of the control operators as well as specifications of other possible implementations in other languages and environments.

A more efficient low-level implementation of our operators can be realized by representing partial continuations as actual stack segments, as in the implementation of traditional continuations described by Hieb *et al.* (1990).

Our framework has already been used as a basis for several interesting applications:

- As a computational effect, backtracking consists of an implementation of the *MonadPlus* interface in Haskell. This interface provides two constructs for introducing a choice junction and for denoting failure. Although stream-based implementations of the interface are possible, continuation-based implementations are often preferred as they avoid a large interpretive overhead (Hinze, 2000). Realistic applications require facilities beyond the simple *MonadPlus* interface to control, guide, and manage the choices; it was not known how to implement such facilities in continuation-based models of backtracking. In a recent collaboration with others, we have used our monadic framework of delimited continuations to design and implement a Haskell library for adding all the desired advanced backtracking primitives to arbitrary monads (Kiselyov *et al.*, 2005). The same ideas are also used in a larger context to implement a complete declarative logic programming system (Friedman & Kiselyov, 2005).
- A "zipper" is a construction that lets us replace an item deep in a complex data structure, *e.g.*, a tree or a term, without any mutation (Huet, 1997). The resulting data structure is supposed to share as much of its components with the old structure as possible. In a recent post to the Haskell mailing list (`http://www.mail-archive.com/haskell@haskell.org/msg16513.html`), Kiselyov notes that a zipper is essentially a cursor into a data structure, and hence can be realised using our framework of delimited continuations. Unlike previous attempts, the implementation he proposes is polymorphic over the data structure to traverse, and the zipper creation procedure is generic and does not depend on the data structure at all.
- Pugs (`http://www.pugscode.org/`) is an implementation of Perl 6, written in Haskell. The language has several of the usual control structures like threads and coroutines. But in addition, Perl 6 has an unusual execution model that allows the compiler to trigger evaluation of blocks on-the-fly, and the evaluator to trigger compilation of source code. Pugs uses both the zipper above to maintain the current position to evaluate, and our framework directly to represent continuations that can be correctly suspended and resumed even in that unusual execution environment. As the authors state, this "has many interesting applications, including web continuations, omniscient debugging, and mobile code." (Tang, 2005).

We further hope to be able to use our framework to tackle the difficult question of tracking the lifetimes and *extent* (Moreau & Queinnec, 1994) of prompts and continuations in the presence of control operators. This issue has two important practical applications.

First, in order to include the control operators in a production language, it is necessary to understand how they interact with other dynamic entities, such as exceptions. The situation is already complicated without prompts, and implementations like SML/NJ provide two variants of `callcc`: one that closes over the current exception handler and one that does not. The implementation does not otherwise promise to satisfy any invariants. In contrast, Scheme includes a *dynamic-wind* operator that guarantees that certain actions are executed before control enters a region and some other actions are executed before control exits a region, even if the entering and exit are done via (traditional) continuation operators. This mechanism was generalised to work with *spawn* in the form of *control filters* by Hieb *et al.* (1994). The interaction of *dynamic-wind* and similar mechanisms with other control abstractions is not yet well-understood.

The second point is closely related to the first point above. If the lifetime of prompts is well-understood, it should be possible to design static checks to enforce that control operations always refer to existing prompts. Recent work (Ariola *et al.*, 2004; Nanevski, 2004) suggests that one must move to a type-and-effect system in order guarantee such properties, but such effects can in principle be expressed in the monadic framework (Wadler, 1998). In the case of *shift* and *reset*, Filinski (1999) does indeed propose a type-and-effect system for layered monadic effects that both keeps track of the interactions between control abstractions (making some programs that mix them inappropriately ill-typed), and guarantees statically that well-typed programs do not fail with “missing prompt” errors. It would be interesting to study how to generalise this work to deal with multiple prompts.

Acknowledgements

We thank the anonymous reviewers, Olivier Danvy, Matthias Felleisen, Andrzej Filinski, Daniel P. Friedman, Oleg Kiselyov, Shriram Krishnamurthi, Simon Marlow, and Philip Wadler for discussions and helpful comments. We would also like to thank Eugenio Moggi for critical comments on an attempted type system for tracking dangling prompts. We also especially thank Oscar Waddell for major contributions to the research ideas and early drafts of the article.

A Implementation of the Prompt Module

```
module Prompt (
  P, Prompt,
  runP, newPromptName, eqPrompt, Equal(..)
) where

import GHC.Prim( unsafeCoerce# )
-- The GHC-specific unsafe coerce function

newtype Prompt ans a = Prompt Int

newtype P ans a = P (Int → (Int,a))
unP (P f) = f
```

```
instance Monad (P ans) where
  return e = P (λs → (s,e))
  (P e1) >>= e2 = P (λs1 → case e1 s1 of
                              (s2,v1) → unP (e2 v1) s2)

runP :: P ans ans → ans
runP pe = snd (unP pe 0)

newPromptName :: P ans (Prompt ans a)
newPromptName = P (λnp → (np+1, Prompt np))

data Equal a b where
  EQUAL     :: Equal a a
  NOT_EQUAL :: Equal a b

eqPrompt :: Prompt ans a → Prompt ans b → Equal a b
eqPrompt (Prompt p1) (Prompt p2)
    | p1 ≡ p2  = unsafeCoerce# EQUAL
    | otherwise = NOT_EQUAL
```

B Implementation of the CPS Module

```
module CPS (
  M, K,
  throw, c,
  runM
) where

newtype K ans a = K (a → ans)
newtype M ans a = M (K ans a → ans)

instance Monad (M ans) where
  return e       =  M (λ (K k) → k e)
  (M e1) >>= e2  =  M (λk → e1 (K (λ v1 → let M c = e2 v1 in c k)))

callcc :: (K ans a → M ans a) → M ans a
callcc f = M (λk → let M c = f k in c k)

abort :: ans → M ans a
abort a = M (λ_ → a)

throw :: K ans a → M ans a → M ans b
throw k (M e) = M (λ_ → e k)
```

```
c :: (K ans a → ans) → M ans a
c f = callcc (λk → abort (f k))

runM :: M ans ans → ans
runM (M e) = e (K id)
```

References

Ariola, Zena M., Herbelin, Hugo, & Sabry, Amr. (2004). A type-theoretic foundation of continuations and prompts. *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York.

Biernacki, Dariusz, & Danvy, Olivier. (2006). A simple proof of a folklore theorem about delimited control. *J. Functional Programming*. to appear.

Cheney, James, & Hinze, Ralf. (2002). A lightweight implementation of generics and dynamics. *Pages 90–104 of: Proceedings of the ACM SIGPLAN Workshop on Haskell*. New York: ACM Press.

Danvy, O., & Filinski, A. (1989). *A functional abstraction of typed contexts*. Tech. rept. 89/12. DIKU, University of Copenhagen, Copenhagen, Denmark.

Danvy, Olivier, & Filinski, Andrzej. (1990). Abstracting control. *Pages 151–160 of: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. New York: ACM Press.

Dybvig, R. Kent, & Hieb, Robert. (1989). Engines from continuations. *Computer Languages*, **14**(2), 109–123.

Felleisen, M., Friedman, D. P., Kohlbecker, E., & Duba, B. (1987a). A syntactic theory of sequential control. *Theoretical Computer Science*, **52**(3), 205–237.

Felleisen, Matthias. (1988). The theory and practice of first-class prompts. *Pages 180–190 of: Conference Record of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*. New York: ACM Press.

Felleisen, Matthias, & Friedman, Daniel P. (1987). A reduction semantics for imperative higher-order languages. *Parallel architectures and languages europe*, **259**, 206–223.

Felleisen, Matthias, Friedman, Daniel P., Duba, Bruce, & Merrill, John. (1987b). *Beyond continuations*. Tech. rept. 216. Indiana University Computer Science Department.

Felleisen, Matthias, Wand, Mitchell, Friedman, Daniel P., & Duba, Bruce F. (1988). Abstract continuations: A mathematical semantics for handling full functional jumps. *Pages 52–62 of: Proceedings of the ACM Conference on Lisp and Functional Programming*. New York: ACM Press.

Filinski, Andrzej. (1994). Representing monads. *Pages 446–457 of: Conference Record of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*. New York: ACM Press.

Filinski, Andrzej. 1996 (May). *Controlling effects*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Technical Report CMU-CS-96-119.

Filinski, Andrzej. (1999). Representing layered monads. *Pages 175–188 of: Conference Record of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*. New York: ACM Press.

Findler, Robert Bruce, Latendresse, Mario, & Felleisen, Matthias. (2001). Behavioral contracts and behavioral subtyping. *Pages 229–236 of: Esec/fse-9: Proceedings of the 8th european software engineering conference held jointly with 9th acm sigsoft international symposium on foundations of software engineering*. New York, NY, USA: ACM Press.

Friedman, Daniel P., & Kiselyov, Oleg. (2005). *A declarative applicative logic programming system*. Available from: `http://kanren.sourceforge.net/`.

Gasbichler, Martin, & Sperber, Michael. (2002). Final shift for call/cc:: direct implementation of shift and reset. *Pages 271–282 of: ACM SIGPLAN International Conference on Functional Programming*. ACM Press.

Gunter, Carl A., Rémy, Didier, & Riecke, Jon G. (1995). A generalization of exceptions and control in ML-like languages. *Functional Programming & Computer Architecture*. New York: ACM Press.

Harper, Robert, Duba, Bruce F., & MacQueen, David. (1993). Typing first-class continuations in ML. *Journal of functional programming*, **3**(4), 465–484.

Hatcliff, John, & Danvy, Olivier. (1994). A generic account of continuation-passing styles. *Pages 458–471 of: Conference Record of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*. New York: ACM Press.

Hieb, Robert, & Dybvig, R. Kent. (1990). Continuations and Concurrency. *Pages 128–136 of: Symposium on Principles and Practice of Parallel Programming*. SIGPLAN Notices, vol. 25(3). Seattle, Washington, March 14–16: ACM Press.

Hieb, Robert, Dybvig, R. Kent, & Bruggeman, Carl. 1990 (June). Representing control in the presence of first-class continuations. *Pages 66–77 of: Proceedings of the SIGPLAN '90 conference on programming language design and implementation*.

Hieb, Robert, Dybvig, Kent, & Anderson, III, Claude W. (1994). Subcontinuations. *Lisp and Symbolic Computation*, **7**(1), 83–110.

Hinze, Ralf. (2000). Deriving backtracking monad transformers. *Pages 186–197 of: ICFP '00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*. ACM Press.

Huet, Gérard. (1997). The zipper. *J. funct. program.*, **7**(5), 549–554.

Kiselyov, Oleg. (2005). *How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible*. Tech. rept. TR611. Indiana University.

Kiselyov, Oleg, Shan, Chung-chieh, Friedman, Daniel P., & Sabry, Amr. (2005). Backtracking, interleaving, and terminating monad transformers (Functional Pearl). *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York.

Launchbury, John, & Peyton Jones, Simon L. (1995). State in Haskell. *Lisp and Symbolic Computation*, **8**(4), 293–341.

Moggi, Eugenio. (1991). Notions of computation and monads. *Information and computation*, **93**(1), 55–92.

Moreau, L., & Queinnec, C. (1994). Partial continuations as the difference of continuations. A duumvirate of control operators. *Lecture notes in computer science*, **844**.

Murthy, C. (1992). Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. *Pages 49–71 of: Acm workshop on continuations*.

Nanevski, Aleksandar. (2004). *A modal calculus for named control effects*. Unpublished manuscript.

Pottier, François, & Régis-Gianas, Yann. (2006). Stratified type inference for generalised algebraic data types. *Pages 232–244 of: 33rd acm symposium on principles of programming languages (popl'06)*. ACM.

Queinnec, Christian, & Serpette, Bernard. (1991). A dynamic extent control operator for partial continuations. *Pages 174–184 of: Conference Record of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*. New York: ACM Press.

Shan, Chung-chieh. (2004). Shift to control. *Pages 99–107 of:* Shivers, Olin, & Waddell, Oscar (eds), *Proceedings of the 5th Workshop on Scheme and Functional Programming.* Technical report, Computer Science Department, Indiana University, 2004.

Sitaram, Dorai, & Felleisen, Matthias. (1990a). Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, **3**(1), 67–99.

Sitaram, Dorai, & Felleisen, Matthias. (1990b). Reasoning with continuations ii: full abstraction for models of control. *Pages 161–175 of: Lfp '90: Proceedings of the 1990 acm conference on lisp and functional programming.* New York, NY, USA: ACM Press.

Sitaram, Dorai, & Felleisen, Matthias. (1991). Models of continuations without continuations. *Pages 185–196 of: Popl '91: Proceedings of the 18th acm sigplan-sigact symposium on principles of programming languages.* New York, NY, USA: ACM Press.

Strachey, Christopher, & Wadsworth, Christopher P. (1974). *Continuations A mathematical semantics for handling full jumps.* Technical Monograph PRG-11. Oxford University Computing Laboratory Programming Research Group.

Tang, Autrijus. (2005). *Pugs: Boolstrapping Perl 6 with Haskell.* Available from `http://perlcabal.org/~autrijus/hw2005.pdf`.

Thielecke, Hayo. (2003). From control effects to typed continuation passing. *Pages 139–149 of: Conference Record of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages.* ACM SIGPLAN Notices, vol. 38, 1. New York: ACM Press.

Wadler, Philip. (1994). Monads and composable continuations. *Lisp and Symbolic Computation*, **7**(1), 39–56.

Wadler, Philip. (1998). The marriage of effects and monads. *Pages 63–74 of: ACM SIGPLAN International Conference on Functional Programming.* ACM Press.

Xi, Hongwei, Chen, Chiyan, & Chen, Gang. (2003). Guarded recursive datatype constructors. *Pages 224–235 of:* Norris, Cindy, & Fenwick, Jr. James B. (eds), *Conference Record of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages.* ACM SIGPLAN Notices, vol. 38, 1. New York: ACM Press.