Technical Report No. 615

A Monadic Framework for Delimited Continuations

by

R. Kent Dybvig
Simon Peyton Jones
Amr Sabry

June 2005

Computer Science Department
Indiana University
Bloomington, Indiana 47405-7104

© 2005 R. Kent Dybvig, Simon Peyton Jones, Amr Sabry

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 seem to 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. In the late eighties a new family of control operators were introduced that apparently went "beyond continuations" [5, 8, 16] and "beyond monads" [26]. 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 [24]. Delimited continuations are also referred to as *subcontinuations* [14], since they represent the remainder of a subcomputation rather than of a computation as a whole.

Without the unifying frameworks of CPS or monads, it is difficult to understand, compare, implement, and reason about the various control operators for subcontinuations, their typing properties, and logical foundations. In this paper 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 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 delimited continuations (Section 3), using a technique first used by Moreau and Queinnec [20]. We simplify the continuation semantics in two ways (Section 3.4) to

*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.

produce a novel continuation semantics. The latter demonstrates 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.

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 an efficient Scheme implementation which we present in Section 5.

We then tackle the problem of *typing* delimited continuations. Building on the monadic semantics, we give a typed account of the subcontinuation operators that makes explicit where control effects can occur, and where they cannot (Section 6). In particular, our design is the first to offer statically-checked guarantees of encapsulation of control effects. We introduce an operator `runCC` which 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 *callcc*. The library implementation is itself strongly typed, which helps enormously when writing its rather tricky code.

2 Control Operators

The literature describes several families of control operators for subcontinuations. In this section, we introduce the family of four operators that we study in detail and relate them to other operators in the literature.

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:

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

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 subexpression and uses the resulting value, which must be a prompt, to delimit the current continuation during the evaluation of its second subexpression.
- The *withSubCont* operator evaluates both of its subexpressions, 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 on a representation of the captured subcontinuation. If more than one activation of *pushPrompt* with prompt p is still active, the most recent activation, *i.e.*, the one that delimits the smallest subcontinuation, is selected.
- The *pushSubCont* operator evaluates its first subexpression to yield a subcontinuation k, then evaluates its second subexpression 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.

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*:

$$run\ e \quad = \quad (\lambda p_0.pushPrompt\ p_0\ e)\ newPrompt$$

Assuming this has been done, we can define a *withCont* operator to manipulate the entire continuation via this prompt:

$$withCont\ e \quad = \quad withSubCont\ p_0\ (\lambda k.pushPrompt\ p_0\ (e\ k))$$

We do not assume a built-in top-level prompt as part of our model. Doing so would not complicate either the semantics or the implementation of our operators. It would, however, be a poor design from a software-engineering perspective, since a built-in top-level prompt gives subprograms possibly undesirable control over the main program.

Given *withCont* we can model Scheme's *call-with-current-continuation* (here abbreviated *callcc*), which captures the current continuation and passes a functional encapsulation of the continuation to its argument:

$$callcc \quad = \quad \lambda f.withCont\ (\lambda k.pushSubCont\ k\ (f\ (reifyA\ k)))$$

where:

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

When applied to a function f, *callcc* captures the entire continuation k using *withCont*, uses *pushSubCont* to reinstate a copy of k, and applies f to a functional representation of k, namely (*reifyA* k). When applied to a value v, this functional representation aborts its context, reinstates k, and returns v to k.

Felleisen's $\mathcal{C}$ [6] 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 $\mathcal{F}$ [6] 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 [7, 5, 3] 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 $\mathcal{F}$ [6]. 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, *reset* and *shift* [3] 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:

- ${}^{-}\mathcal{F}^{-}$ does not leave the prompt behind or include it in the subcontinuation; this is like *cupto* [13], and our operator *withSubCont*.
- ${}^{-}\mathcal{F}^{+}$ does not leave the prompt behind, but does include it in the subcontinuation; this is like a *spawn* controller [15].
- ${}^{+}\mathcal{F}^{-}$ leaves the prompt behind, but does not include it in the subcontinuation; this is the delimited $\mathcal{F}$ operator [7].
- ${}^{+}\mathcal{F}^{+}$ leaves the prompt behind and includes it in the subcontinuation; this is the *shift* operator [3].

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 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:

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

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 [3, 23]. Other proposals instead allow new prompts to be generated dynamically, like *spawn* [15, 14]. 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 expressive 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:

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

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 [20] proposed a pair of operators, *marker* and *call/pc*, 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 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:

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

Such an operator is easily added to our system given the implementation approach we present later in this paper. 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.* [13] 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.

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

In fact, aside from minor syntactic details, our framework differs from the one by Gunter *et al.* only in the representation of continuations and how they are used. In the latter framework, continuations are represented as functions; in ours, the representation is left abstract. Furthermore, 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 [4].

The operators ${}^{-}\mathcal{F}^{+}$ and ${}^{+}\mathcal{F}^{-}$ share an intuitively appealing identity property, which is that capturing and immediately reinstating a subcontinuation is effectively a no-op:

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

In contrast, the operator ${}^{-}\mathcal{F}^{+}$ takes away the prompt, but its subcontinuation reinstates it, while ${}^{+}\mathcal{F}^{-}$ leaves the prompt, and its subcontinuation does not reinstate it. Thus, the same operation with ${}^{-}\mathcal{F}^{-}$ results in the net elimination of one prompt, while the same operation with ${}^{+}\mathcal{F}^{+}$ results in the net introduction of one prompt. Although this would seem to make ${}^{-}\mathcal{F}^{+}$ or ${}^{+}\mathcal{F}^{-}$ better choices, we have chosen ${}^{-}\mathcal{F}^{-}$ semantics for our primitive operator *withSubCont*, because the ${}^{-}\mathcal{F}^{-}$ semantics easily models each the others with the trivial addition of *pushPrompt* forms where necessary to leave behind or reinstate a prompt. While Shan [22] has demonstrated that one can use even ${}^{+}\mathcal{F}^{+}$ semantics (in the form of *shift*) to implement the semantics of ${}^{-}\mathcal{F}^{-}$, ${}^{-}\mathcal{F}^{+}$, and ${}^{+}\mathcal{F}^{-}$, doing so requires a complex syntactic redefinition of the prompt operator, using a trampolining mechanism similar to one used by Sitaram and Felleisen [23] to implement hierarchies of # and $\mathcal{F}$.

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 (Section 3.2). Thus motivated, we develop the semantics in both an operational style and a continuation 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 subexpression, *callcc* applies the resulting function f to a function encapsulating the captured continuation in the same continuation. If the function encapsulating the captured continuation is applied to a value, it aborts the current continuation and reinstates the captured continuation by passing the value to the captured continuation and dropping the current continuation.

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 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 the prompt p, and $\kappa^p_\downarrow$ representing the portion of κ below the prompt p.

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 [8]. 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 realized 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 [3]. Since we must split the continuation above and below the prompt, why not maintain two separate parameters to the CPS semantics? The first parameter κ will correspond to the portion of the evaluation context above the first prompt, and the second parameter γ will correspond 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. 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. The metacontinuation approach also requires that the program undergo two CPS conversion passes. The first is a nonstandard one that exposes the continuation but leaves behind non-tail calls representing the metacontinuation, and the second is a standard one that exposes the metacontinuation. Additional complexity is involved in the presence of multiple prompts. Handling static hierarchical prompts requires additional CPS conversion passes [3], and while we conjecture that the trampolining reset operators of Sitaram and Felleisen and of Shan can be extended to handle dynamically generated prompts, this would further complicate that mechanism.

On the other hand, the algebra of contexts is too concrete. 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 [12], even though prompts may be many frames apart.

3.3 Operational Semantics

It turns out that we can strike a middle ground that provides all the expressiveness of the sequence of frames approach while leaving the representation of continuations abstract. We do this by adopting features of both of the traditional approaches to modelling delimited continuations. We borrow from the metacontinuation approach the notion of a split continuation. From the algebra of contexts, we borrow the representation of a continuation as a sequence. The key insight is that we need represent only the metacontinuation as a sequence while leaving the representation of partial continuations fully abstract. This technique was first applied by Moreau and Queinnec in a semantics for *marker* and *call/pc* [20].

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 inte-

Group I. Searching for a redex:

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

Group II. Executing a redex:

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

Group III. Returning a value:

$$
\begin{array}{rcll}
(v, D, E, q) & \longmapsto & (D[v], \Box, E, q) & D \neq \Box \\
(v, \Box, p : E, q) & \longmapsto & (v, \Box, E, q) & \\
(v, \Box, D : E, q) & \longmapsto & (v, D, E, q) &
\end{array}
$$

Figure 1: Operational semantics

gers, 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.

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

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 any pushed prompts), E is the rest of the context represented as a sequence, and q is a global counter used to generate new prompt values. We write and manipulate sequences using the Haskell syntactic conventions for lists. 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 $E' \texttt{++} (p : E'')$ 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 forces the current context (which can only extend up to the first pushed prompt) to be empty. 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 subexpression 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* application 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 but Abstract CPS Semantics

Using the ideas developed in the previous sections, it is possible to develop a CPS translation for the call-by-value λ-calculus embedding of our operators. The translation is given in Figure 2. The continuation κ has a conventional functional representation and corresponds to the context D in the operational semantics. The metacontinuation γ is a sequence of prompts and continuations as before, 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, $[]$ 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

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

where, in the target language:

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

and:

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

Figure 2: CPS translation of call-by-value calculus with control

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 do not need any other axioms regarding lists or integers.

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. 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

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

Figure 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:

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

Figure 4: Factoring the control operations

is treated as a normal function 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 (just once!) using a 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 final variant of the CPS translation is equivalent to the previous two by using $\beta\eta$-equations to abstract the uses of the runtime 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:

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

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 $\mathcal{P}[\![D[e]]\!] = \lambda\kappa.\mathcal{P}[\![e]\!](\lambda x.\mathcal{P}[\![D[x]]\!]\kappa)$

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* $(e, D, S, q) \longmapsto (e', D', S', q')$ *then* $\mathcal{P}[\![(e, D, S, q)]\!] = \mathcal{P}[\![(e', D', S', q')]\!]$.

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 like *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 [9]). 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.

An alternative, and now well-established, technique is to express the order of evaluation by a translation $\mathcal{T}[\![e]\!]$ into a *monadic meta-language*, after which 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.

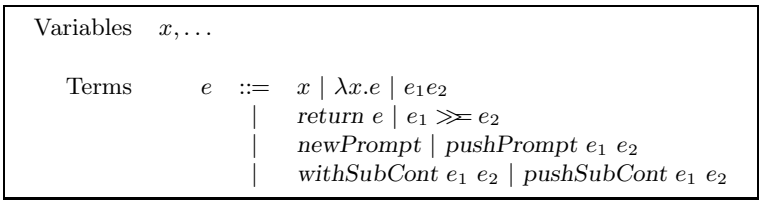

Figure 5: Syntax of the monadic metalanguage

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

Figure 6: Monadic translation of call-by-value calculus with control

By separating the issues related to the order of evaluation from the semantics of control operators, we gain better understanding of both aspects. By using the monadic language, with its clear distinction between terms with no effects and terms of computation type, function calls can no longer trigger computational effects which must be triggered explicitly using the special computation rules of the monad. (See the `loop` example in Section 6.2 for a concrete example of how a typical use of thunks to control the order of evaluation can be better achieved in the monadic metalanguage.) Finally, the separation allows us to focus in the rest of the paper on the more important issues related to the semantics and implementation of the control operators without unnecessary distractions.

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 subexpressions 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. Notice that 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.

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

Figure 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 [19]: they manipulate a concrete representation of the continuation but know nothing about the metacontinuation. The monad-specific control operations are 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 $\mathcal{T}_k[\![e]\!]$. We show that these two translations are equivalent.

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

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 pure functional terms remain pure, and the monadic constructs are aware of the continuation but not the metacontinuation or the generation of new names; 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. The second implementation in Section 5.2 addresses the problem with tail recursion by taking advantage of *Chez Scheme*'s equality property for continuations.

More implementations strategies are possible, for example by converting to CPS, but we do not present them here.

5.1 Direct 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 implicit 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, and the global counter p is eliminated in favour of some hidden state used to generate fresh strings.

Using this approach, nothing needs to be done to implement `return` and `>>=` since the effects are implicit; the four control operators are implemented as follows:

```
(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 code uses a datatype Seq (not shown) with three constructors EmptyS, PushP, and PushSeg. The operations appendSeq and splitSeq are simple operations on that datatype that realize the operations ++ and $\gamma^p_\uparrow$ and $\gamma^p_\downarrow$ from the semantics.

The main challenge in the implementation is actually in setting up the top-level procedure runCC which should also define the procedure abort which corresponds to using 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 mk)
(define abort)

(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. The definition of underflow applies 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)]))
```

This implementation effectively generalises Filinksi's implementation of *shift* and *reset* using SML/NJ's *callcc* and a metacontinuation cell [9] to our family of control operators, which can easily and efficiently support the other control operators described in Section 2.

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?. To do so, we modify runCC to reify the base continuation and store it in the variable *base-k*:

```
(define mk)
(define base-k)
(define abort)

(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.

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, 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 the 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. Furthermore, 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:

```
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 t` is a computation that may have control effects. When the computation is run, the control effects are performed, and a result of type `t` 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. 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 a 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.* [13] 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 [26] 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.

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 like this 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 (either of them) 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 it explicit that the argument to a continuation must be a *value*, of type `a`, rather than a computation of type *`CC`* `a`. This interface of `callcc` has a well-known stack-space leak, 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 continuation, passing a value of type `(() → a )` instead of a value of type `a`. 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: the fact that the *interface* of `runCC` specifies a fixed type *Obs* means that every use of the control operators is limited to return the same fixed type. One really wants to be able to run monadic computations that return values of arbitrary types. Naïvely replacing *Obs* by an arbitrary type is however unsound as it would allow interactions among control operations executing under different occurrences of `runCC`. For example, changing the type of `runCC` to:

```
runCC :: CC a → a
```

would permit the following:

```
abortP :: Prompt r b → CC r b → CC r a
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 body of `badc` 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`*.

The solution 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* [15].) As Thielecke [25] recently showed, there is an intimate relation between regions and the type of observables. Indeed what *defines* a region of control is that the type of observables can be made local to the region. Fortunately this situation is rather similar to the well-understood situation of encapsulating state in Haskell [18], and our solution is quite similar. We add a *region parameter* `r` to every type constructor and enforce non-interference and localisation of control actions by using polymorphism. The refined interface becomes:

```
module CC where

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

instance Monad (CC r)

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

In the new interface, the types *`CC`*, *`Prompt`*, and *`SubCont`* are each given an additional type parameter `r` 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`* `r1 a` if the current computation has type *`CC`* `r2 a` where `r1` and `r2` are different regions. The type of `runCC` shows that it takes an effectful computation, of type *`CC`* `r a`, runs it, and returns an ordinary, pure, value of type `a`. This encapsulation is enforced by giving `runCC` a rank-2 type: its argument must be polymorphic in the region `r`.

6.4 Examples

Encapsulation using `runCC` provides a convenient way to isolate regions of control from each other. If a computation labelled by `r1` pushes a prompt, then a computation labelled by a different `r2` cannot access that prompt and hence cannot abort or duplicate computations

up to that prompt. Moreover the type system will enforce 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.

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 version of the monadic types with regions (implementing the other basic interface is simpler). Providing these three typed implementations has several advantages:

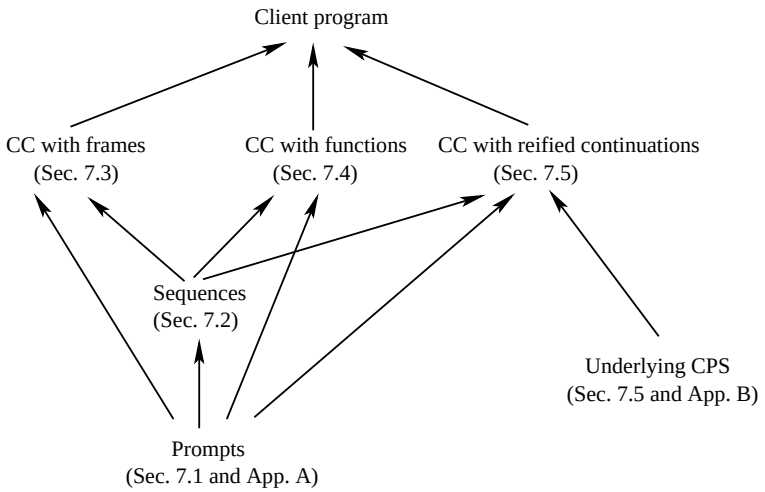

Figure 8: Module dependencies

- 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 will show that it is possible to focus on each aspect in a separate module.
- The semantics in Figures 3 and 7 is quite complex and the types are non-trivial. We found the executable Haskell specification to be invaluable in debugging the semantic definitions.
- The executable specification naturally provides an extension of Haskell with our control operators, but it also provides a blueprint for embedding our control operators in other languages like Scheme or ML either by modifying the runtime system, or extending a CPS compiler, or as a source level library which builds on top of `callcc`.

The implementation uses several modules whose inter-dependencies are given in Figure 8. The client program can import any of the three implementation 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 (CC with reified) also imports a module which manipulates an underlying "native" representation of continuations that is only available via control operators. The implementation uses several constructs that are not part of Haskell 98, although they have become standard extensions to the basic language. In particular, we use existential types to express the typing of a sequence of function (continuation) compositions; and we use universal types for encapsulation of control effects and for capturing an invariant related to continuations and metacontinuations.

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 the only unsafe (in the sense that the type system cannot prove it safe) coercion in our code to this small module:

```
module Prompt where
  data P r a        -- Abstract
  data Prompt r a   -- Abstract

  instance Monad (P r)

  runP       ::  (∀ r.  P r a) → a
  newPrompt  ::  P r (Prompt r a)
  eqPrompt   ::  Prompt r a → Prompt r b → Maybe (a → b, b → a)
```

The module provides the abstract type of prompts and a monad `(P r)` which sequences the prompt supply. This guarantees that generated prompts are globally unique within all computations tagged by the type parameter of the region `r`. The implementation of the module is in Appendix A.

The operation `runP` plays the role of encapsulating a computation that uses prompts, guaranteeing that no information about the prompts is either imported by its argument or exported by it. The operation `eqPrompt` compares two prompts of possibly *different* types by looking at their internal representation. If the two prompts have a different internal representation we just return the value *Nothing*. But if the two prompts have the same internal representation, then they must have the same type (if our implementation is correct and if the prompt values are unforgeable). In this case we return two coercions to witness the type equality. The coercions are implemented as identity functions, but since the Haskell type system cannot be used to reason about this type equality, the coercion functions are generated using an unsafe implementation-dependent primitive.

The module *Prompt* is imported in each of the following implementations of the *CC* interface. The import is qualified so that uses of component *X* of the *Prompt* module will appear as *Prompt.X*.

7.2 Sequences

All our implementations manipulate sequences of prompts and control segments. The control segments are frames in the first case, functions in the second case, and abstract continuations with an unknown representation in the third case. We provide here a general sequence type that can be instantiated for each case. The two operations we require on sequences (`split` and `append`) need to be defined only once.

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

```
module Seq where

data Seq s r a b =  EmptyS (a → b)
                 |  PushP (Prompt.Prompt r a) (Seq s r a b)
                 |  ∀ c.  PushSeg (s r a c) (Seq s r c b)
                 |  ∀ c.  PushCO (a → c) (Seq s r c b)
```

```
splitSeq   ::  Prompt.Prompt r b → Seq s r a ans → (Seq s r a b, Seq s r b ans)

appendSeq  ::  Seq s r a b → Seq s r b ans → Seq s r a ans
```

The basic structure of the *Seq* type is that of a list, which can be empty (*EmptyS*) or has three "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 parameters `a` and `b` represent the type of values received and produced by the aggregate sequence of prompts and segments. The type parameter `r` is used to identify the region of control to which the prompts and control segments belong. The type parameter `s` is the abstract constructor of control segments that will be varied to produce the various implementations. We now consider the constructors:

- We would really like to declare the empty sequence *EmptyS* as:

  ```
  data Seq s r a b = EmptyS | ...
  ```

  but then *EmptyS* would have the type *Seq* `s r a b` which is too polymorphic. An empty sequence should have type *Seq* `s r a a`. Haskell does not allow data types to be restricted in this way (but there is recent work on possible extensions and encodings [28, 2]), so we instead give *EmptyS* an argument that provides evidence that `a = b`, in the form of a function from `a` to `b`. Now we can define:

  ```
  emptyS :: Seq s r a a
  emptyS = EmptyS id
  ```

- The *PushP* constructor is simple: it simply pushes a (suitably-typed) prompt onto the sequence.

- The *PushSeg* constructor pushes a *control segment* which represents either an individual frame or a 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 "$\forall c$" in the declaration is a widely-used Haskell extension that allows an existentially-quantified type variable `c` to be used in a data type declaration [17]. It is used here to express the fact that if the control segment takes a value of type `a` to one of some type `c`, and the rest of the sequence takes a value of that type `c` to a value of type `b`, then the composition of the two takes a value of type `a` to a value of type `b` irrespective of what `c` is.

- The *PushCO* constructor pushes a coercion function (again always the identity in our code) onto a sequence. The coercions are the ones obtained from the *Prompt* module that witness the type equality of two prompts.

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. To append functional sequences, we recursively traverse the first until we reach its base case. The base case provides a coercion function which can be used to build a coercion frame to maintain the proper types.

```
splitSeq :: Prompt.Prompt r b → Seq s r a ans → (Seq s r a b, Seq s r b ans)
splitSeq p (EmptyS _)        =   error ("Prompt was not found on the stack")
splitSeq p (PushP p' sk)     =   case Prompt.eqPrompt p' p of
                                   Nothing →
                                     let (subk,sk') = splitSeq p sk
                                     in (PushP p' subk, sk')
                                   Just (a2b,b2a) →
                                     (EmptyS a2b, PushCO b2a sk)
splitSeq p (PushSeg seg sk)  =   let (subk,sk') = splitSeq p sk
                                 in (PushSeg seg subk, sk')
splitSeq p (PushCO f sk)     =   let (subk,sk') = splitSeq p sk
                                 in (PushCO f subk, sk')

appendSeq :: Seq s r a b → Seq s r b ans → Seq s r a ans
appendSeq (EmptyS f) sk          =   PushCO f sk
appendSeq (PushP p subk) sk      =   PushP p (appendSeq subk sk)
appendSeq (PushSeg seg subk) sk  =   PushSeg seg (appendSeq subk sk)
appendSeq (PushCO f subk) sk     =   PushCO f (appendSeq subk sk)
```

7.3 Continuations as Sequences of Frames

Now 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 in one data-structure which consists of a sequence of frames and prompts. Although we worked hard to avoid making this representation the *only* representation possible, it is a possible representation which is useful if one chooses to modify the runtime system to implement the control operators [12]. A frame of type *Frame* `r a b` is a function which 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 an arbitrary (and hence universally quantified) type `obs`. The final result should be of type `obs` but is slightly more complicated since we are making the allocation of prompts explicit: the final result is instead a computation which delivers the value of type `obs` after possibly generating prompts. Thus the complete definitions of the datatypes are:

```
data Frame r a b = Frame (a → CC r b)
type Cont r a b = Seq Frame r a b

data CC r a = CC (∀ obs.  Cont r a obs → Prompt.P r obs)
type Prompt r a = Prompt.Prompt r a
type SubCont r a b = Seq Frame r a b
```

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

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

appk :: Cont r a obs → a → Prompt.P r obs
appk (EmptyS f) v             =  return (f v)
appk (PushP _ k) v            =  appk k v
appk (PushSeg (Frame f) k) v  =  let CC e = f v in e k
appk (PushCO f k) v           =  appk k (f v)

runCC :: (∀ r.  CC r a) → a
runCC ce = Prompt.runP (let CC e = ce in e (EmptyS id))
```

The function `appk` serves as an interpreter, transforming a sequence *data structure*, of type *Cont* `r a obs`, into a *function*. The implementation of `appk` is straightforward but needs a coercion in the *EmptyS* case, without which the function would not be well-typed.

The implementation of the four control operators is now straightforward:

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

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

withSubCont :: Prompt r b → (SubCont r a b → CC r b) → CC r a
withSubCont p f = CC (λk → case splitSeq p k of
                             (subk,k') →  let CC e = f subk in e k')

pushSubCont :: SubCont r a b → CC r a → CC r b
pushSubCont subk (CC e) = CC (λk → e (appendSeq subk k))
```

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

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:

```
data Cont r a b = Cont (a → MC r b)
type MetaCont r a b = Seq Cont r a b
```

```
data CC r a = CC (∀ b.  Cont r a b → MC r b)
data MC r b = MC (∀ ans.  MetaCont r b ans → Prompt.P r ans)
type Prompt r a = Prompt.Prompt r a
type SubCont r a b = Seq Cont r a b
```

The type `ans` is quantified as above. 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. Had we not quantified the type variables `b` and `ans` in the definitions, then we would have had to either fix them to arbitrary types or we would have had to make them additional parameters to the type constructors.

In more detail, if we remove the quantification from the definitions of the types *CC* and *MC* (and remove the tags to simplify the discussion), we might get:

```
type CC r ans b a = (a → MC r ans b) → MC r ans b
type MC r ans b = MetaCont r b ans → Prompt.P r ans
```

The type variable `ans` that used to be quantified is now a parameter to the *MC* constructor, which means it has also to be a parameter to the *CC* constructor. The *CC* constructor also needs to take as a parameter the type `b` that used to be quantified. If we ignore the dynamic generation of prompts (and hence also the type parameter `r`) we get:

```
type CC ans b a = (a → MC ans b) → MC ans b
type MC ans b = MetaCont b ans → ans
```

which is identical to the "Murthy types" considered by Wadler [26]. These types are however not expressive enough. Alternatively, the quantified type variables can be eliminated by fixing the type `ans` to be an arbitrary but fixed type *Obs*:

```
type CC b a = (a → MC b) → MC b
type MC b = MetaCont b Obs → Obs
```

which is identical to the restricted two-level types considered by Wader [26] and to our interface in Section 6.3.

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

```
instance Monad (CC r) where
  return e          =   CC (λ (Cont k) → k e)
  (CC e1) >>= e2    =   CC (λk → e1 (Cont (λv1 → let CC c = e2 v1 in c 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 a continuation that knows about the metacontinuation. The function `runCC` takes a computation and supplies it with the initial continuation; this returns another computation which expects the initial metacontinuation:

```
abortC :: (Cont r a a → MC r a) → MC r a
abortC e = e (Cont (λv → MC (λmk → appmk mk v)))

appmk :: MetaCont r a ans → a → Prompt.P r ans
appmk (EmptyS f) e             =  return (f e)
appmk (PushP _ sk) e           =  appmk sk e
appmk (PushSeg (Cont k) sk) e  =  let MC mc = k e in mc sk
appmk (PushCO f sk) e          =  appmk sk (f e)

runCC :: (∀ r.  CC r a) → a
runCC ce = Prompt.runP (let CC e = ce
                            MC me = abortC e
                        in me (EmptyS id))
```

The exported operators are now implemented as follows:

```
newPrompt :: CC r (Prompt r a)
newPrompt = CC (λ (Cont k) →
              MC (λmk → do  p ← Prompt.newPrompt
                            let MC me = k p
                            me mk))

pushPrompt :: Prompt r a → CC r a → CC r a
pushPrompt p (CC e) =
    CC (λk → MC (λmk →  let MC me = abortC e
                        in me (PushP p (PushSeg k mk))))

withSubCont :: Prompt r b → (SubCont r a b → CC r b) → CC r a
withSubCont p f =
    CC (λk → MC (λmk →
       let  (subk,mk') = splitSeq p mk
            CC e = f (PushSeg k subk)
            MC me = abortC e
       in me mk'))

pushSubCont :: SubCont r a b → CC r a → CC r b
pushSubCont subk (CC e) =
    CC (λk → MC (λmk →
       let MC me = abortC e
       in me (appendSeq subk (PushSeg k mk))))
```

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: an underlying monad `CPS.M` which manipulates a concrete representation of the continuation `CPS.K` that is hidden from

the main monad implementing the *CC*-interface. The main monad can capture and invoke the continuation manipulated by the underlying monad but must treat the type *CPS.K* as an abstract type.

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

```
module CPS where

data K obs a     -- Abstract
data M obs a     -- Abstract

instance Monad (M obs)

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

runM  :: M obs obs → obs
```

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:

```
data Cont r a b = Cont (CPS.K (MC r b) a)
type MetaCont r a b = Seq Cont r a b

data CC r a = CC (∀ b.  CPS.M (MC r b) a)
data MC r b = MC (∀ ans.  MetaCont r b ans → Prompt.P r ans)
type Prompt r a = Prompt.Prompt r a
type SubCont r a b = Seq Cont r a b
```

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 untagging of the values:

```
instance Monad (CC r) where
  return e        =  CC (return e)
  (CC e1) >>= e2  =  CC (do  v1 ← e1
                             let CC c = e2 v1
                             c)
```

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. The definition of `underflow` is almost in one-to-one correspondence with the definition of the initial continuation in the previous section, and so are the functions `abortC` and `runCC`:

```
abortC :: CPS.M (MC s a) a → MC s a
abortC e = CPS.runM (e >>= underflow)

underflow :: a → CPS.M (MC s a) (MC s a)
underflow v = return (MC (λsk → appmk sk v))

appmk :: MetaCont r a ans → a → Prompt.P r ans
appmk (EmptyS f) v               =  return (f v)
appmk (PushP _ sk') v            =  appmk sk' v
appmk (PushSeg (Cont k) sk') v   =  let MC f = resumeC k v in f sk'
appmk (PushCO f sk') v           =  appmk sk' (f v)

resumeC :: CPS.K (MC s b) a → a → MC s b
resumeC k v = CPS.runM (CPS.throw k (return v))

runCC :: (∀ s.  CC s a) → a
runCC ce = Prompt.runP (let CC e = ce
                            MC sf = abortC e
                        in sf (EmptyS id))
```

The exported operations are now implemented as follows:

```
newPrompt :: CC r (Prompt r a)
newPrompt =
    CC (CPS.c (λk → MC (λsk →
      do p ← Prompt.newPrompt
         let MC sf = resumeC k p
         sf sk)))

pushPrompt :: Prompt r a → CC r a → CC r a
pushPrompt p (CC e) =
    CC (CPS.c (λk → MC (λsk →
        let MC sf = abortC e
        in sf (PushP p (PushSeg (Cont k) sk)))))

withSubCont :: Prompt r b → (SubCont r a b → CC r b) → CC r a
withSubCont p f =
    CC (CPS.c (λk → MC (λsk →
        let (subk,sk') = splitSeq p sk
            CC e = f (PushSeg (Cont k) subk)
            MC sf = abortC e
        in sf sk')))

pushSubCont :: SubCont r a b → CC r a → CC r b
pushSubCont subk (CC e) =
    CC (CPS.c (λk → MC (λsk →
```

```
let sk' = appendSeq subk (PushSeg (Cont k) sk)
    MC sf = abortC e
in sf sk')))
```

This implementation of our control operators generalises previous direct-style implementations of *shift* and *reset* [9, 10] and $\mathcal{F}$ and prompt [23].

8 Conclusions and further work

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

- It provides a clear separation of several entangled issues that complicate the semantics of such control operators: non-standard order of evaluation, 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.
- It can be implemented on top of any traditional implementation of continuations, including a single, standard CPS translation.

We have also described how a CPS or direct-style implementation of functional control operators can be made properly tail recursive.

Our framework is implemented in an almost well-typed Haskell library which provides executable specifications of the control operators as well as specifications of other possible implementations in other languages and environments.

We hope to be able to use our framework to tackle the difficult question of tracking the lifetimes and *extent* [20] 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 **dynamic-wind**, which 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 using continuations. The interaction of such a primitive with arbitrary control operators is not 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 [1, 21] 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 [27]. In the case of *shift* and *reset*, Filinski [11] 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 will not fail with "missing prompt" errors.

Acknowledgements

We thank the anonymous reviewers, Olivier Danvy, Matthias Felleisen, Andrzej Filinski, Dan Friedman, 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 paper.

A Implementation of the Prompt Module

```
module Prompt (
  P, Prompt, runP,
  newPrompt, eqPrompt
) where

data P r a = P (Int → (Int,a))
data Prompt r a = Prompt Int

instance Monad (P r) where
  return e        =  P (λs → (s,e))
  (P e1) >>= e2   =  P (λs1 →  let  (s2,v1)  =  e1 s1
                                    P f2     =  e2 v1
                               in f2 s2)


runP     ::  (∀ r.  P r a) → a
runP pe  =   let P e = pe in snd (e 0)

newPrompt  ::  P r (Prompt r a)
newPrompt  =   P (λs → (s+1, Prompt s))

eqPrompt :: Prompt r a → Prompt r b → Maybe (a → b, b → a)
eqPrompt (Prompt p1) (Prompt p2)
    | p1 ≡ p2    =   Just (coerce id, coerce id)
    | otherwise   =   Nothing

coerce  ::  a → b
coerce  =   ...       -- implementation dependent
```

B Implementation of the CPS Module

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

data K ans a = K (a → ans)
data 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

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

[2] Cheney, J., and Hinze, R. A lightweight implementation of generics and dynamics. In *Proceedings of the ACM SIGPLAN workshop on Haskell* (New York, 2002), ACM Press, pp. 90–104.

[3] Danvy, O., and Filinski, A. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (New York, 1990), ACM Press, pp. 151–160.

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

[5] Felleisen, M. The theory and practice of first-class prompts. In *Conf. Record of 15th Ann. ACM Symp. on Principles of Programming Languages, POPL'88, San Diego, CA, USA, Jan. 1988*. ACM Press, New York, 1988, pp. 180–190.

[6] Felleisen, M., Friedman, D. P., Duba, B., and Merrill, J. Beyond continuations. Tech. Rep. 216, Indiana University Computer Science Department, 1987.

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

[8] Felleisen, M., Wand, M., Friedman, D. P., and Duba, B. F. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming* (New York, July 1988), ACM Press, pp. 52–62.

[9] Filinski, A. Representing monads. In *Conference Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programm ing Languages, Portland, Oregon* (New York, 1994), ACM Press, pp. 446–457.

[10] Filinski, A. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1996. Technical Report CMU-CS-96-119.

[11] Filinski, A. Representing layered monads. In *Conf. Record of 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'99, San Antonio, TX, USA, 20–22 Jan. 1999*. ACM Press, New York, 1999, pp. 175–188.

[12] Gasbichler, M., and Sperber, M. Final shift for call/cc:: direct implementation of shift and reset. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming* (2002), ACM Press, pp. 271–282.

[13] Gunter, C. A., Rémy, D., and Riecke, J. G. A generalization of exceptions and control in ML-like languages. In *Functional Programming & Computer Architecture* (New York, 1995), ACM Press.

[14] Hieb, R., Dybvig, K., and Anderson, III, C. W. Subcontinuations. *Lisp and Symbolic Computation 7*, 1 (1994), 83–110.

[15] Hieb, R., and Dybvig, R. K. Continuations and Concurrency. In *PPoPP '90, Symposium on Principles and Practice of Parallel Programming* (Seattle, Washington, March 14–16, 1990), vol. 25(3) of *SIGPLAN NOTICES*, ACM Press, pp. 128–136.

[16] Johnson, G. F., and Duggan, D. Stores and partial continuations as first-class objects in a language and its environment. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, 1988), ACM Press, pp. 158–168.

[17] Läufer, K., and Odersky, M. An extension of ML with first-class abstract types. In *Proc. ACM SIGPLAN Workshop on ML and its Applications* (New York, June 1992), ACM.

[18] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *Lisp and Symbolic Computation 8*, 4 (Dec. 1995), 293–341.

[19] Moggi, E. Notions of computation and monads. *Information and Computation 93*, 1 (1991), 55–92.

[20] Moreau, L., and Queinnec, C. Partial continuations as the difference of continuations. A duumvirate of control operators. *Lecture Notes in Computer Science 844* (1994).

[21] Nanevski, A. A modal calculus for named control effects. Unpublished manuscript, 2004.

[22] Shan, C. Shift to control. In *Proceedings of the 5th workshop on Scheme and Functional Programming* (2004), O. Shivers and O. Waddell, Eds., pp. 99–107. Technical report, Computer Science Department, Indiana University, 2004.

[23] Sitaram, D., and Felleisen, M. Control delimiters and their hierarchies. *Lisp and Symbolic Computation 3*, 1 (1990), 67–99.

[24] Strachey, C., and Wadsworth, C. P. Continuations A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory Programming Research Group, 1974.

[25] Thielecke, H. From control effects to typed continuation passing. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-03)* (New York, Jan. 15–17 2003), vol. 38, 1 of *ACM SIGPLAN Notices*, ACM Press, pp. 139–149.

[26] Wadler, P. Monads and composable continuations. *Lisp and Symbolic Computation 7*, 1 (Jan. 1994), 39–56.

[27] Wadler, P. The marriage of effects and monads. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming* (1998), ACM Press, pp. 63–74.

[28] Xi, H., Chen, C., and Chen, G. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-03)* (New York, Jan. 15–17 2003), C. Norris and J. J. B. Fenwick, Eds., vol. 38, 1 of *ACM SIGPLAN Notices*, ACM Press, pp. 224–235.