A Monadic Framework for Subcontinuations

Friday 29th April, 2005

R. Kent Dybvig
Indiana University

Simon Peyton Jones
Microsoft Research

Amr Sabry*
Indiana University

Abstract. Functional and delimited continuations are more expressive than traditional abortive continuations and they apparently seem to require a framework beyond traditional continuation or monadic semantics. We show that this is not the case: standard continuation semantics is sufficient to explain directly the common control operators for delimited continuations. This implies a monadic framework for typed and encapsulated functional and delimited continuations which we design and implement as a Haskell library.

Keywords: callcc, composable continuation, control delimiter, delimited continuation, encapsulation, Haskell, monad, prompt, reset, shift

1. Introduction

Continuation-passing style (CPS) and its generalization 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" (Felleisen, 1988; Felleisen et al., 1988; Johnson and Duggan, 1988) and "beyond monads" (Wadler, 1994). These control operators permit the manipulation of *delimited continuations* that represent only part of the remainder of a computation, and they also support the *composition* of continuations, even though such operations are not directly supported by standard continuation models (Strachey and Wadsworth, 1974). Delimited continuations are also referred to as *subcontinuations* (Hieb et al., 1994), since they represent the remainder of a subcomputation rather than of a computation as a whole.

Without the unifying frameworks of continuation semantics or monads, it is difficult to understand, compare, implement, and reason about

* 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.

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 generalize 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 first give an abstract and expressive continuation semantics for delimited continuations (Section 3), using a technique first used by Moreau and Queinnec. We simplify this semantics in two ways to produce a novel continuation semantics that 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 allows us to give a *typed* account of the subcontinuation operators that makes explicit where control effects can occur, and where they cannot (Section 5). 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 an ordinary, 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 6). 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.

We also present a properly tail recursive Scheme implementation of our operators in Appendix C.

(Variables)	$x, \ldots$	
(Expressions)	e	$::= x \mid \lambda x.e \mid e\ e$
		$\mid$ *newPrompt* $\mid$ *pushPrompt* $e\ e$
		$\mid$ *withSubCont* $e\ e \mid$ *pushSubCont* $e\ e$

Figure 1. Call-by-value λ-calculus with control

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*. Figure 1 shows the syntax of our operators, extending a conventional, call-by-value λ-calculus. We give their semantics formally in Section 3, but intuitively they 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, *pushPrompt* and *pushSubCont* must be treated as syntactic constructs since they exhibit a non-standard evaluation order.

These operators are essentially identical to ones proposed by Gunter et al. (1995). The only difference aside from minor syntactic details is that our operators do not require captured subcontinuations to be represented as functions.

2.2. Relationship with Existing Operators

While our operators can be used directly, the primary intent is that they be used as building blocks to form higher level control operators, including existing operators from the literature. To provide some intuition about our operators, we compare them with existing operators and show how they can be used to express those 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. To model traditional continuations, we assume the existence of a top-level prompt available as the constant p_0 and define a *withCont* operator to manipulate the entire continuation via this prompt.

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

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

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

where:

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

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 $(\mathit{reifyA}\ k)$. When applied to a value v, this functional representation aborts its context, reinstates k, and returns v to k.

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

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

Like continuations reified by *callcc*, a continuation reified by $\mathcal{C}$ aborts the current continuation when it is invoked. In contrast, the operator $\mathcal{F}$ (Felleisen et al., 1987a) also captures and aborts the entire continuation, but the reified continuation is functional, or composable,

as with our subcontinuations. It can be modeled with a non-aborting *reify* operator:

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

where:

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

When prompts appear other than at top level, they serve as control delimiters (Felleisen et al., 1987b; Felleisen, 1988; Danvy and Filinski, 1990) and allow programs to capture and abort a subcontinuation, *i.e.*, a continuation representing *part of* the remainder of the computation rather than *all* of it. The first control delimiter to be introduced was Felleisen's # (prompt), which delimits, *i.e.*, marks the base of, the continuation captured and aborted by $\mathcal{F}$ (Felleisen et al., 1987a). 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 is invoked. For example, *reset* and *shift* (Danvy and Filinski, 1990) are similar to # and $\mathcal{F}$, but *shift* both leaves behind the prompt when a subcontinuation is captured and includes the prompt in the captured subcontinuation.

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

- $^{-}\mathcal{F}^{-}$ does not leave the prompt behind or include it in the subcontinuation; this is like *cupto* (Gunter et al., 1995) and *withSubCont*.
- $^{-}\mathcal{F}^{+}$ does not leave the prompt behind, but does include it in the subcontinuation; this is like a *spawn* controller (Hieb and Dybvig, 1990).
- $^{+}\mathcal{F}^{-}$ leaves the prompt behind, but does not include it in the subcontinuation; this is the delimited $\mathcal{F}$ operator (Felleisen et al., 1987b).
- $^{+}\mathcal{F}^{+}$ leaves the prompt behind and includes it in the subcontinuation; this is the *shift* operator (Danvy and Filinski, 1990).

In all cases, the traditional interface is that the captured subcontinuation is reified as a function. Using our primitives and a single constant prompt #, these operators can be defined as follows:

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

where

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

A natural extension of the framework with a single fixed prompt is to allow multiple prompts. Some proposals generalize the single prompt by allowing hierarchies of prompts and control operators, like reset_n and shift_n (Danvy and Filinski, 1990; Sitaram and Felleisen, 1990).

Other proposals instead allow new prompts to be generated dynamically, like *s*pawn (Hieb and Dybvig, 1990; Hieb et al., 1994). In such systems, the base of each subcontinuation is rooted at a different prompt, and each generated prompt is associated with a function that can be used for accessing the continuation up to that prompt. This is more 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:

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

where we have generalized 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 specialized control operator available to its argument f.

Moreau and Queinnec (1994) 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 behavior in our system with the addition of a *strip* operator as follows.

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

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 behavior 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.

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. 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:

$$\begin{aligned} {}^{+}\mathcal{F}^{-}\ (\lambda k.ke) &= e \qquad \text{if } k \notin e \\ {}^{-}\mathcal{F}^{+}\ (\lambda k.ke) &= e \qquad \text{if } k \notin e \end{aligned}$$

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 it is the one that most easily models the others. While Shan (2004) 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 (1990) to implement hierarchies of # and $\mathcal{F}$.

We have supposed the possible existence of a top-level prompt p_0, which we needed to implement *callcc*, $\mathcal{C}$, and $\mathcal{F}$ without #. We do not insist that the top-level prompt be included in the model, however. It may be preferable *not* to include a top-level prompt, since this gives subprograms possibly undesirable control over the main program, which can easily provide the subprogram with a top-level prompt if desired.

3. Continuation Semantics

In this section, we develop a continuation semantics for the call-by-value λ-calculus embedding of our operators. We proceed in the traditional way, by giving a translation from the *source language* of Figure 1 to a pure, call-by-name lambda calculus *target language* (introduced in Section 3.1). We review the traditional CPS semantics for simple control operators like *callcc* (Section 3.2) and explain why it is insufficient for *delimited* continuations. We then discuss why neither of the two standard approaches to extending the CPS translation for delimited

$$
\begin{array}{lll}
\text{(Variables)} & x, \kappa, \ldots & \\
\text{(Numerals)} & \underline{n} & \\
 & & \\
\text{(Expressions)} & e & ::= x \mid \lambda x.e \mid e\ e \\
 & & \quad \mid \ \mathsf{error} \\
 & & \quad \mid \ \underline{n} \mid e + e \mid \mathsf{num?}\ e \\
 & & \quad \mid \ [] \mid e : e \mid \mathsf{hd}\ e \mid \mathsf{tl}\ e \mid \mathsf{null?}\ e \\
 & & \quad \mid \ \mathsf{true} \mid \mathsf{false} \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e
\end{array}
$$

Figure 2. Extended call-by-name λ-calculus: Syntax

continuations is entirely satisfactory (Section 3.3). Thus motivated, we develop an expressive and abstract CPS semantics in Sections 3.4 and 3.5.

3.1. A CPS Calculus

In order to make the CPS semantics precise, we first introduce a pure calculus to serve as the *target* of the CPS translation. The calculus is a standard call-by-name one whose syntax is given in Figure 2. In addition to the core λ-terms, it includes a constant denoting an error, numbers, lists, and booleans with their associated operations. The semantics of the CPS calculus is standard, and is given by the equivalences in Figure 3.

In the sequel, we take the liberty to use pattern-matching syntax, recursive definitions, and other convenient syntactic sugar which is easily translated to the core calculus.

3.2. 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 Figure 1 (without the control operations for now) and returns an expression in the extended call-by-name calculus of Figure 2. The result of the translation is always a λ-expression that expects a continuation and returns an answer of some arbitrary but fixed type:

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

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:

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

$$
\begin{array}{rcll}
(\lambda x.e)e' & = & e[e'/x] & \\
\lambda x.(\lambda y.e)\ x & = & \lambda y.e & \text{if } x \notin e \\
\underline{n_1} + \underline{n_2} & = & \underline{n_1 + n_2} & \\
 & & & \\
\mathsf{num?}\ n & = & \mathsf{true} & \\
\mathsf{num?}\ (\lambda x.e) & = & \mathsf{false} & \\
\mathsf{num?}\ [] & = & \mathsf{false} & \\
\mathsf{num?}\ (e_1 : e_2) & = & \mathsf{false} & \\
\mathsf{hd}\ (e_1 : e_2) & = & e_1 & \\
\mathsf{tl}\ (e_1 : e_2) & = & e_2 & \\
\mathsf{null?}\ [] & = & \mathsf{true} & \\
\mathsf{null?}\ (e_1 : e_2) & = & \mathsf{false} & \\
\textbf{if}\ \mathsf{true}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & = & e_1 & \\
\textbf{if}\ \mathsf{false}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & = & e_2 & \\
 & & & \\
\mathsf{error}\ e' & = & \mathsf{error} & \\
\mathsf{error} + e & = & \mathsf{error} & \\
n + \mathsf{error} & = & \mathsf{error} & \\
\mathsf{num?}\ \mathsf{error} & = & \mathsf{error} & \\
\mathsf{hd}\ \mathsf{error} & = & \mathsf{error} & \\
\mathsf{tl}\ \mathsf{error} & = & \mathsf{error} & \\
\mathsf{null?}\ \mathsf{error} & = & \mathsf{error} & \\
\textbf{if}\ \mathsf{error}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & = & \mathsf{error} &
\end{array}
$$

Figure 3. Extended call-by-name λ-calculus: Semantics

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 $\mathcal{F}$ in the absence of # is also straightforward:

$$\mathcal{P}[\![\mathcal{F}e]\!] \ = \ \lambda\kappa.\mathcal{P}[\![e]\!]\ (\lambda f.f\ (\lambda x.\lambda\kappa'.\kappa'(\kappa\ x))\ \kappa_0)$$

In this case, f is invoked in the initial continuation, effectively aborting the current continuation, and the encapsulating function does not drop the current continuation but rather composes it with the captured continuation.

Handling even a single prompt # is not so straightforward. What we need 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 is a richer representation of the continuation that supports two operations: $\kappa^{\#}_{\uparrow}$ representing the portion of κ above the prompt, and $\kappa^{\#}_{\downarrow}$ representing the portion of κ below the prompt.

3.3. Traditional Solutions

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

1. Abstract continuation semantics (Felleisen et al., 1988). This approach develops an *algebra of contexts* that is expressive enough to support the required operations on continuations. From the algebra, two representations for continuations are derived: one as a sequence of frames, and the other as objects with two methods for invoking and updating the continuation. The operations $\kappa^{\#}_{\uparrow}$ and $\kappa^{\#}_{\downarrow}$ 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 their two sequences.

2. Metacontinuations (Danvy and Filinski, 1990). 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 nontail 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 (Danvy and Filinski, 1990), 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 not sufficiently abstract for our purposes, due to its over-constrained representation of continuations. Although a common representation of continuations in an implementation is indeed as a stack of frames, exposing this fine granularity in the semantics suggests that an implementation must loop through these frames individually (Gasbichler and Sperber, 2002), even though prompts may be many frames apart. 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.

3.4. Representing Metacontinuations

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 as abstract as possible. We do this by adopting features of both of the traditional approaches to modeling 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 (1994) in a semantics for *marker* and *call/pc*.

Before giving the full CPS translation, we first discuss the representation of partial continuations and metacontinuations. A partial continuation is represented in the standard way for CPS semantics, *i.e.*, as a function mapping values to answers. A metacontinuation is represented as a list, where each element is either a numeral (representing a unique prompt name) or a partial continuation.

When a metacontinuation (represented as a list) is applied to a value, the result should be a final observable answer. In our case, this is slightly more complicated in order to deal properly with the generation of new prompts, without using global side-effects. To accomplish this, we assume a global supply of names (represented as numerals) that is threaded through along with the metacontinuation. Applying a metacontinuation is defined by cases on the sequence representing the metacontinuation:

$$\begin{aligned} \mathcal{K}([], v) &= \lambda p.v \\ \mathcal{K}(p : \gamma, v) &= \mathcal{K}(\gamma, v) \\ \mathcal{K}(\kappa : \gamma, v) &= \kappa\ v\ \gamma \end{aligned}$$

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

Figure 4. CPS translation of call-by-value calculus with control

If the sequence is empty, then we are done: we simply ignore the prompt supply and return the value v as the final answer. If the sequence starts with a prompt p, then the value is returned through the prompt, which is popped, and the next segment of the metacontinuation is inspected. Finally, if the sequence starts with a partial continuation κ, then κ is given the value v and the rest of the sequence as its metacontinuation.

With the list representation, a metacontinuation can easily be split at an arbitrary prompt, and two metacontinuations can easily be composed. Composition is achieved via the function ++, which appends two lists. The operations that split the metacontinuation are defined below:

$$
\begin{array}{rcl@{\qquad}rcl}
[]^p_\uparrow & = & \mathsf{error} & []^p_\downarrow & = & \mathsf{error} \\
(p:\gamma)^p_\uparrow & = & [] & (p:\gamma)^p_\downarrow & = & \gamma \\
(p':\gamma)^p_\uparrow & = & p':\gamma^p_\uparrow & (p':\gamma)^p_\downarrow & = & \gamma^p_\downarrow \quad \text{where } p \neq p' \\
(\kappa:\gamma)^p_\uparrow & = & \kappa:\gamma^p_\uparrow & (\kappa:\gamma)^p_\downarrow & = & \gamma^p_\downarrow
\end{array}
$$

3.5. An Expressive but Abstract CPS Semantics

A CPS translation for the call-by-value λ-calculus embedding of our operators is given in Figure 4. The translation of an expression e from Figure 1 is $\mathcal{P}[\![e]\!]\kappa_0\ []\ \underline{0}$ where κ_0 is the initial partial continuation, $[]$ is the initial empty metacontinuation, and $\underline{0}$ is the first generated prompt name. The initial partial continuation κ_0 takes a value and a metacontinuation and applies the metacontinuation to the value: $\lambda v.\lambda\gamma.\mathcal{K}(\gamma, v)$.

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

Figure 5. CPS translation of call-by-value calculus with control (η-reduced)

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

where, in the target language:

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

Figure 6. Factoring the control operations

So far, this 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 push on it a bit harder.

Figure 5 simplifies the CPS translation by η-reducing the equations in Figure 4 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 5 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 metacontinuation and next prompt are accessed in a monad-like way, separate from the everyday handling of control.

Figure 6 takes the simplification one step further. The handling of core terms is as in Figure 5, 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 thereby becomes completely independent of the meta-

continuation and next prompt and deals with the control operators only superficially. A practical consequence of this observation is that any program that makes use of delimited continuations can be evaluated simply by rewriting the program (just once!) using a completely standard CPS-conversion algorithm and by supplying additional "hidden" arguments—the metacontinuation and next prompt—and suitable implementations of our operators that manipulate those arguments. 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.

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 the default parameter-passing mechanism would evaluate e_2 *before* invoking the *pushPrompt* operation. Since the whole point of *pushPrompt* is to introduce a prompt to which control operations in e_2 can refer, evaluating those control operations before pushing the prompt defeats the purpose of the operation. One solution is to treat the control operators as syntactic constructs, as we have done so far. Another is to use thunks to manually delay and force the evaluation of e_2 at the appropriate times (see for example the embedding of `reset` in ML by Filinski (1994)). As shown in Appendix C, 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 behavior of the control operators can be expressed by defining them as constants, just as we did in Section 3.5. 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 in the next section to introduce a monadic typing discipline to track and encapsulate the control effects.

By separating the issues related to the order of evaluation from the semantics of control operators, we gain better understanding of

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

Figure 7. Monadic metalanguage: Syntax

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

Figure 8. Monadic translation of call-by-value calculus with control

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 5.3 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 7. 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 we defer the type issues until Section 5.

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

Figure 9. CPS translation of monadic metalanguage

The monadic translation is in Figure 8. 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.

4.2. Semantics of the Monadic Metalanguage

The monadic metalanguage is now translated to the CPS calculus of Section 3.1. The translation $\mathcal{M}$ is given in Figure 9. In the presentation of the translation, we have grouped the term constructors in four groups. The first group consists of the pure λ-calculus constructors whose semantics knows nothing about continuations or metacontinuations. The second group is the standard monadic constructors *return* and $\gg\!=$ which are given the standard definitions for the CPS monad (Moggi, 1991), *i.e.*, they manipulate a concrete representation of the continuation but know nothing about the metacontinuation. The third group consists of the control operators other than *newPrompt*. The semantics of these control operators manipulate the continuation (but not its representation) and manipulate a concrete representation of the metacontinuation. Finally the semantics of the last control operator *newPrompt* refers to the continuation, the metacontinuation, and the counter used to generate unique names.

4.3. Relating the CPS and Monadic Semantics

The semantics are equivalent in the sense of the following proposition.

PROPOSITION 4.1. *For any expression e defined in Figure 1, we have* $\mathcal{P}[\![e]\!] = \mathcal{M}[\![\mathcal{T}[\![e]\!]]\!]$ *in the CPS calculus.*

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

- $e = x$.
  The left-hand side is $\lambda\kappa.\kappa\ x$.
  The right-hand side is $\mathcal{M}[\![\mathcal{T}[\![x]\!]]\!] = \mathcal{M}[\![return\ x]\!] = \lambda\kappa.\kappa\ x$.

- $e = \lambda x.e'$
  The left-hand side is $\lambda\kappa.\kappa\ (\lambda x.\lambda\kappa'.\mathcal{P}[\![e']\!]\ \kappa')$.
  The right-hand side is:

$$\begin{aligned}\mathcal{M}[\![return\ (\lambda x.\mathcal{T}[\![e']\!])]\!] &= \lambda\kappa.\kappa\ (\mathcal{M}[\![\lambda x.\mathcal{T}[\![e']\!]]\!])\\ &= \lambda\kappa.\kappa\ (\lambda x.\mathcal{M}[\![\mathcal{T}[\![e']\!]]\!])\\ &= \lambda\kappa.\kappa\ (\lambda x.\lambda\kappa'.\mathcal{P}[\![e']\!]\ \kappa')\end{aligned}$$

- $e = e_1 e_2$
  The left-hand side is $\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda f.\mathcal{P}[\![e_2]\!]\ (\lambda a.f\ a\ \kappa))$.
  The right-hand side is:

$$\begin{aligned}&\mathcal{M}[\![\mathcal{T}[\![e_1]\!] \gg\!= \lambda f.\mathcal{T}[\![e_2]\!] \gg\!= \lambda a.f\ a]\!]\\ =\ &\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda f.\mathcal{M}[\![\mathcal{T}[\![e_2]\!] \gg\!= \lambda a.f\ a]\!]\ \kappa)\\ =\ &\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda f.\mathcal{M}[\![\mathcal{T}[\![e_2]\!]]\!]\ (\lambda a.\mathcal{M}[\![f\ a]\!]\ \kappa))\\ =\ &\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda f.\mathcal{P}[\![e_2]\!]\ (\lambda a.f\ a\ \kappa))\end{aligned}$$

- $e = newPrompt$
  The left-hand side is $\lambda\kappa.\lambda\gamma.\lambda p.\kappa\ p\ \gamma\ (p + \underline{1})$.
  The right-hand side is $\mathcal{M}[\![newPrompt]\!] = \lambda\kappa.\lambda\gamma.\lambda p.\kappa\ p\ \gamma\ (p + \underline{1})$

- $e = pushPrompt\ e_1\ e_2$
  The left-hand side is $\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda p.\lambda\gamma.\mathcal{P}[\![e_2]\!]\ \kappa_0\ (p : \kappa : \gamma))$.
  The right-hand side is:

$$\begin{aligned}&\mathcal{M}[\![\mathcal{T}[\![e_1]\!] \gg\!= \lambda p.pushPrompt\ p\ \mathcal{T}[\![e_2]\!]]\!]\\ =\ &\lambda\kappa.\mathcal{M}[\![\mathcal{T}[\![e_1]\!]]\!]\ (\lambda p.\mathcal{M}[\![pushPrompt\ p\ \mathcal{T}[\![e_2]\!]]\!]\ \kappa)\\ =\ &\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda p.\lambda\gamma.\mathcal{M}[\![\mathcal{T}[\![e_2]\!]]\!]\ \kappa_0\ (p : \kappa : \gamma))\\ =\ &\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda p.\lambda\gamma.\mathcal{P}[\![e_2]\!]\ \kappa_0\ (p : \kappa : \gamma))\end{aligned}$$

- $e = withSubCont\ e_1\ e_2$
  The left-hand side is $\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda p.\mathcal{P}[\![e_2]\!]\ (\lambda f.\lambda\gamma.f\ (\kappa : \gamma^p_\uparrow)\ \kappa_0\ \gamma^p_\downarrow))$.
  The right-hand side is:

$$\begin{aligned}&\mathcal{M}[\![\mathcal{T}[\![e_1]\!] \gg\!= \lambda p.\mathcal{T}[\![e_2]\!] \gg\!= \lambda f.withSubCont\ p\ f]\!]\\ =\ &\lambda\kappa.\mathcal{M}[\![\mathcal{T}[\![e_1]\!]]\!]\ (\lambda p.\mathcal{M}[\![\mathcal{T}[\![e_2]\!] \gg\!= \lambda f.withSubCont\ p\ f]\!]\ \kappa)\\ =\ &\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda p.\mathcal{M}[\![\mathcal{T}[\![e_2]\!]]\!]\ (\lambda f.\mathcal{M}[\![withSubCont\ p\ f]\!]\ \kappa))\\ =\ &\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda p.\mathcal{P}[\![e_2]\!]\ (\lambda f.\lambda\gamma.f\ (\kappa : \gamma^p_\uparrow)\ \kappa_0\ \gamma^p_\downarrow))\end{aligned}$$

– $e = \mathit{pushSubCont}\ e_1\ e_2$
The left-hand side is $\lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda\gamma'.\lambda\gamma.\mathcal{P}[\![e_2]\!]\ \kappa_0\ (\gamma' \texttt{++} (\kappa : \gamma)))$.
The right-hand side is:

$$\begin{aligned}
& \mathcal{M}[\![\mathcal{T}[\![e_1]\!] \gg\!= \lambda s.\mathit{pushSubCont}\ s\ \mathcal{T}[\![e_2]\!]]\!] \\
= \ & \lambda\kappa.\mathcal{M}[\![\mathcal{T}[\![e_1]\!]]\!]\ (\lambda s.\mathcal{M}[\![\mathit{pushSubCont}\ s\ \mathcal{T}[\![e_2]\!]]\!]\ \kappa) \\
= \ & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda s.\lambda\gamma.\mathcal{M}[\![\mathcal{T}[\![e_2]\!]]\!]\ \kappa_0\ (s \texttt{++} (\kappa : \gamma))) \\
= \ & \lambda\kappa.\mathcal{P}[\![e_1]\!]\ (\lambda s.\lambda\gamma.\mathcal{P}[\![e_2]\!]\ \kappa_0\ (s \texttt{++} (\kappa : \gamma)))
\end{aligned}$$

In summary, the CPS semantics of Figure 5 has been teased into two parts that can be studied (and implemented as we see in Section 6) 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. Monadic Types in Haskell

In order to study the monadic types in a concrete setting, we implement the monadic metalanguage of the preceding section in Haskell. This implementation allows us not only to use advanced type features like interfaces, type classes, nested polymorphism, and existentials, but also to provide an executable specification of our control operators.

From the semantic perspective, *i.e.*, based on Figure 9, the monadic metalanguage can be easily mapped to Haskell. Since Haskell is an extended λ-calculus, it directly embodies the pure λ-calculus terms of the monadic metalanguage. Furthermore, Haskell provides direct syntactic support for monadic programming.

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 Haskell definitions for the monadic constructors *return* and $\gg\!=$, and the control operators, which can be done in a Haskell library. 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 6. 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.

5.1. Haskell as an Implementation of the Monadic Metalanguage

Defining the monadic constructors in Haskell is directly achieved by defining an instance of the `Monad` type class. Specifically, we must introduce a type constructor, say `CC`, that describes the notion of effect we are interested in, and make `CC` an instance of the class `Monad` by providing definitions of the two methods `return` and `>>=`. For example, if `e1` and `e2` are expressions whose evaluation may have control effects, we can write:

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

The evaluation of the expression 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`. Haskell provides the following convenient syntactic sugar for the above pattern:

```
do x1 ← e1
   x2 ← e2
   return (x1+x2)
```

Finally we must provide definitions for the control operators. Naturally these operators use the definition of the `CC` monad and in fact they are the only operations that need access to that definition.

In other words, we can embed the monadic metalanguage in Haskell by simply defining a *library* which exports the monadic type constructor `CC` and the control operations. In the remainder of this section, we present the interfaces of two such libraries and evaluate them using examples.

5.2. Monad with Fixed Observable Type

We first introduce the simplest types for the monadic library.

```
data CC a          -- Abstract
data Prompt a      -- Abstract
data SubCont a b   -- Abstract
type Obs = ...     -- Arbitrary but fixed

instance Monad CC

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` (which must be an instance of the class `Monad`) and two abstract type constructors for prompts `Prompt` and subcontinuations `SubCont`. Following conventional continuation semantics, the type of observables is of an arbitrary but fixed type `Obs`. The type `CC a` is the type of computations returning a value of type `a` to their continuation. 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`. To execute a complete program the function `runCC` takes a computation which returns a value of the fixed type `Obs` and supplies it with the initial context (initial continuation, metacontinuation, and counter for prompt names) to get the final observable value.

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

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

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

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

5.3. Examples

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

- A λ-expression `\x -> e` extends as far to the right as possible.
- A sequence of monadic computations is usually expressed using `do`-notation but we occasionally use the bind operator `>>=`.
- Whitespace (instead of semi-colons) is often used as a separator of monadic actions with indentation (instead of braces) indicating grouping.
- 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)`.

Thus 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}))
```

We first revisit our examples with the top-level prompt p_0 and `callcc` from Section 2.2. The top-level prompt has type `Prompt Obs` and the definitions of `abort` and `callcc` can 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.

5.4. 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`. 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 intentionally 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* (Hieb and Dybvig, 1990).) As Thielecke (2003) 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 (Launchbury and Peyton Jones, 1995), and our solution is quite similar. We add a *region parameter* `r` to every type constructor and enforce non-interference and localization of control actions by using polymorphism. The refined interface becomes:

```
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`.

5.5. Examples

Encapsulation using `runCC` provides a convenient way to isolate regions of control from each other. If a computation labeled by `r1` pushes a prompt, then a computation labeled 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 localized 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 optimization. The control effect is completely localized 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`.

The system with encapsulation is quite expressive. Wadler (1994) studies several generalizations of the type system with a fixed observable type, and only the most general system he considers (which is more general than a monad) is expressive enough to typecheck the following program:

```
let g = reset (if (shift (lambda (f) f)) then 2 else 3)
in (g True) + (g False)
```

This can be written in our framework without stepping outside the world of monads. The following expression typechecks and evaluates to 5 as desired:

```
data A r = Done Int | Sub (Bool → CC r (A r))

w = runCC (
      do p ← newPrompt
         Sub g ← pushPrompt p $
                  do b ← shift p (λs → return (Sub s))
                     if b
                       then return (Done 2)
                       else return (Done 3)
         Done n1 ← g True
         Done n2 ← g False
         return (n1+n2))
```

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. Example `b1` attempts to use a subcontinuation captured from outside its region which is also invalid, and is rejected by the type checker.

6. Executable Specifications in Haskell

Having implemented the monadic metalanguage in Haskell, we can turn the semantics of Figure 9 into an *executable* (as well as typed) specification. We focus on the more interesting version of the monadic types with regions (implementing the other basic interface is simpler). We also provide three versions of the specification that differ in the details of how the continuation is represented. Providing these three typed specifications has several advantages:

- 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 5 and 9 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 Haskell's built-in `error` mechanism to avoid cluttering the code with yet-another-monad to propagate errors. The implementation also uses several constructs that are not part of Haskell 98, although they have become quite 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.

6.1. 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`.

6.2. 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 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:

```
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)
```

We first give the intuitive meaning of each type parameters 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 see Xi et al. (2003) and Cheney and Hinze (2002) for possible extensions and encodings), 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 (Läufer and Odersky, 1992). It is used here to express the fact that if the control segment takes a value of type `a` to one of type `c`, and the rest of the sequence takes a value of 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`.

– 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 implementation of the control operators described later may push "empty" control segments which correspond to the identity continuation, or worse, sequences of coercions if the control operators are used in certain recursive patterns. In both cases, it is possible to avoid such inefficiencies by using "smart constructor" functions, and this is absolutely essential if we are to maintain proper tail recursion as required for example by the semantics of Scheme. For example, instead of using `PushCO` to construct the sequences, we use a function `pushCO` that recognizes the special inefficient situation in which we push one coercion on top of another and combines the two coercions. For control segments, the situation is a little more subtle since it could be that continuations are represented as functions, or even worse, continuations may have an unknown representation, and hence it is not clear how to identify the identity continuation. It may still be possible however even in those situations to avoid pushing a control segment corresponding to the identity continuation. For example, one can refine the representation of continuations represented as functions to be `Id | NonId (A -> B)` where the identity continuation is readily recognizable as such. It could also be possible depending on the host language to compare continuations of unknown representation for (pointer) equality, as is possible for example in *Chez Scheme* where:

$$(\mathit{call/cc}\ (\mathbf{lambda}\ (k_1)\ (\mathit{call/cc}\ (\mathbf{lambda}\ (k_2)\ (\mathit{eqv?}\ k_1\ k_2)))))$$

evaluates to *true*. As shown in Appendix C, this property can be used to achieve proper tail recursion, *i.e.*, no growth of any values holding control information, including the implementation's stack.

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)
```

6.3. Continuations as Sequences of Frames

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 (Gasbichler and Sperber, 2002). 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.

```
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`.

6.4. Continuations as Functions

In this second implementation, 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 (1994). These types are however not expressive enough to type the example in Section 5.2. 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 Wadler (1994) and to our interface in Section 5.4.

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:

```
runC :: (Cont r a a → MC r a) → MC r a
runC 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 = runC 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 = runC 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 = runC 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 = runC e
       in me (appendSeq subk (PushSeg k mk))))
```

6.5. Continuations Reified by a Control Operator

This third implementation even more clearly formalizes 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 `runC` and `runCC`:

```
runC :: CPS.M (MC s a) a → MC s a
runC 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 (PushCC 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 = runC 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 = runC 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 = runC 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 = runC e
        in sf sk')))
```

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

7. Conclusions

We have presented a typed monadic framework in which one can define and experiment with arbitrary 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.

Chung-chieh Shan (2004) has also recently shown that subcontinuation control operators can be expressed using standard CPS, by defining the various control operators in terms of *shift* and *reset*, which can then in turn be implemented using CPS. Our method represents the metacontinuation as a sequence and directly implements the most basic of these operators, ${}^{-}\mathcal{F}^{-}$, from which it is easy to implement any of the others. In contrast, Shan's method represents the metacontinuation as a function and implements the least basic of these operators, ${}^{+}\mathcal{F}^{+}$. To implement the more basic operators, he introduces wrappers similar to our underflow wrappers around each prompt to convert, in effect, the metacontinuation into a sort of procedural list representation.

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. This idea is illustrated by translating the most interesting specification to Scheme (Appendix C), thereby giving an implementation of our control operators on top of `callcc`.

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

First, in order to include the control operators in a production language, it is necessary to understand how they interact with other dynamic entities, such as exceptions. The situation is already complicated without prompts, and implementations like SML/NJ provide two variants of `callcc`: one that closes over the current exception handler and one that does not. The implementation does not otherwise promise to satisfy any invariants. In contrast, Scheme includes **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 (Ariola et al., 2004; Nanevski, 2004) suggests that one must move to a type-and-effect system in order guarantee such properties, but such effects can in principle be expressed in the monadic framework (Wadler, 1998). In the case of *shift* and *reset*, Filinski (1999) does indeed propose a type-and-effect system for layered monadic effects that both keeps track of the interactions between control abstractions (making some programs that mix them inappropriately ill-typed), and guarantees statically that well-typed programs 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.

Appendix

A. Implementation of the module `Prompt`

```
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 module CPS

```
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)
```

C. Implementation in Scheme

This appendix contains three implementations of our control operators in Scheme. Section C.1 presents a transliteration of the Haskell code of Section 6.5 to Scheme, using call/cc to implement C. Section C.2 simplifies the code by using call/cc directly and maintaining the meta-continuation as a global variable. Section C.3 modifies the latter to handle tail recursion properly by taking advantage of *Chez Scheme*'s equality property for continuations.

C.1. Transliterating the Haskell Code

The code in this section is a literal translation to Scheme of the Haskell code that appears in Section 6.5. We begin by defining the Seq datatype and associated routines[1].

[1] **define-datatype** is a syntactic abstraction; its definition is not interesting here, so it is omitted to save space. An implementation is available from the authors.

```
(define-datatype Seq
  (EmptyS)
  (PushP p Seq)
  (PushSeg k Seq))

(define (appendSeq seq1 seq2)
  (Seq-case seq1
    [(EmptyS) seq2]
    [(PushP p subk) (PushP p (appendSeq subk seq2))]
    [(PushSeg k subk) (PushSeg k (appendSeq subk seq2))]))

(define (splitSeq p seq)
  (Seq-case seq
    [(EmptyS) (error 'splitSeq "prompt ~s not found on stack" p)]
    [(PushP p* sk)
     (if (not (eqv? p p*))
         (let-values ([(subk sk*) (splitSeq p sk)])
           (values (PushP p* subk) sk*))
         (values (EmptyS) sk))]
    [(PushSeg k sk)
     (let-values ([(subk sk*) (splitSeq p sk)])
       (values (PushSeg k subk) sk*))]))
```

The Haskell implementation expects $\mathcal{C}$ rather than *callcc*, so the Scheme version of `runM` defines C in terms of call/cc. The operator C is defined once at top-level where it is visible to the various control operators but assigned its value once per invocation of runM so that it aborts back to the continuation of the most recent call to runM. The computation to be performed by runM is represented as a thunk.

```
(define C)
(define (runM th)
  ((call/cc
     (lambda (abort)
       (set! C
         (lambda (p)
           (call/cc
             (lambda (k)
               (abort (lambda () (p k)))))))
       (let ([v (th)])
         (C (lambda (k) v)))))))
```

The definitions of runC, runCC, underflow, and appmk mirror their Haskell counterparts.

```
(define (runC th) (runM (lambda () (underflow (th)))))
(define (runCC th) ((runC th) (EmptyS)))
(define (underflow v) (lambda (mk) (appmk mk v)))

(define (appmk mk v)
  (Seq-case mk
    [(EmptyS) v]
    [(PushP _ mk*) (appmk mk* v)]
    [(PushSeg k mk*) ((runM (lambda () (k v))) mk*)]))
```

Prompts are fresh strings. (Any mutable object would suffice.)

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

The **pushPrompt** and **pushSubCont** operators are 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 ...))]))
```

The definitions of the control operators are equivalent to their Haskell counterparts.

```
(define ($pushPrompt p th)
  (C (lambda (k)
       (lambda (mk)
         ((runC th) (PushP p (PushSeg k mk)))))))

(define ($pushSubCont subk th)
  (C (lambda (k)
       (lambda (mk)
         ((runC th) (appendSeq subk (PushSeg k mk)))))))

(define (withSubCont p f)
  (C (lambda (k)
       (lambda (mk)
         (let-values ([(subk mk*) (splitSeq p mk)])
           ((runC (lambda () (f (PushSeg k subk)))) mk*))))))
```

C.2. Simplifying the Code

We can simplify the code above a bit by working with the native call/cc directly and keeping the metacontinuation in a global variable instead of passing it around in store-passing style.

Our new runCC sets up the initial metacontinuation *mk* and the runC procedure. (We have no need of a separate runM procedure, since the "virtual machine" is plain Scheme.)

```
(define mk)
(define runC)

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

runC accepts a thunk and thaws it in a base continuation that encapsulates only the call to underflow.

The definition of underflow does the job of the original appmk, where *mk* is maintained in a global variable via assignments.

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

The control operators also maintain *mk* as a global variable, plus use call/cc rather than C. Passing a thunk to runC, which thaws it in the base continuation, effectively simulates the aborting effect of C.

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

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

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

This implementation effectively generalizes Filinksi's implementation of *shift* and *reset* using SML/NJ's *callcc* and a metacontinuation cell (Filinski, 1994) to our family of control operators, which can easily and efficiently support the other control operators described in Section 2.

C.3. 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 implementations presented above do not treat tail recursion properly, since each **pushSubCont** of s adds a new (empty) subcontinuation onto the metacontinuation and the metacontinuation grows without bound. In order to recognize and avoid this situation, the code must have some way to detect empty subcontinuations, as described in Section 6.2. 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 runC)

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

References

Ariola, Z. M., H. Herbelin, and A. Sabry: 2004, 'A Type-Theoretic Foundation of Continuations and Prompts'. In: *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York.

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

Danvy, O. and A. Filinski: 1990, 'Abstracting Control'. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. New York, pp. 151–160, ACM Press.

Felleisen, M.: 1988, '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*. New York: ACM Press, pp. 180–190.

Felleisen, M., D. P. Friedman, B. Duba, and J. Merrill: 1987a, 'Beyond Continuations'. Technical Report 216, Indiana University Computer Science Department.

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

Felleisen, M., M. Wand, D. P. Friedman, and B. F. Duba: 1988, 'Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps'. In: *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*. New York, pp. 52–62, ACM Press.

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

Filinski, A.: 1996, 'Controlling Effects'. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Technical Report CMU-CS-96-119.

Filinski, A.: 1999, '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*. New York: ACM Press, pp. 175–188.

Gasbichler, M. and M. Sperber: 2002, '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*. pp. 271–282, ACM Press.

Gunter, C. A., D. Rémy, and J. G. Riecke: 1995, 'A Generalization of Exceptions and Control in ML-like Languages'. In: *Functional Programming & Computer Architecture*. New York, ACM Press.

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

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

Johnson, G. F. and D. Duggan: 1988, '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, pp. 158–168, ACM Press.

Läufer, K. and M. Odersky: 1992, 'An Extension of ML with First-Class Abstract Types'. In: *Proc. ACM SIGPLAN Workshop on ML and its Applications*. New York, ACM.

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

Moggi, E.: 1991, 'Notions of Computation and Monads'. *Information and Computation* **93**(1), 55–92.

Moreau, L. and C. Queinnec: 1994, 'Partial Continuations as the Difference of Continuations. A Duumvirate of Control Operators'. *Lecture Notes in Computer Science* **844**.

Nanevski, A.: 2004, 'A Modal Calculus for Named Control Effects'. Unpublished manuscript.

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

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

Strachey, C. and C. P. Wadsworth: 1974, 'Continuations A Mathematical Semantics for Handling Full Jumps'. Technical Monograph PRG-11, Oxford University Computing Laboratory Programming Research Group.

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

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

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

Xi, H., C. Chen, and G. Chen: 2003, 'Guarded recursive datatype constructors'. In: C. Norris and J. J. B. Fenwick (eds.): *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-03)*, Vol. 38, 1 of *ACM SIGPLAN Notices*. New York, pp. 224–235, ACM Press.