Delimited control with multiple prompts in theory and practice

Paul Downen Zena M. Ariola
University of Oregon {pdownen,ariola}@cs.uoregon.edu

1. Proposal
The versatile and expressive capabilities of delimited control and composable continuations have gained it attention in both the theory and practice of functional programming with effects. On the more theoretical side, delimited control may be used as a basis for explaining and understanding a wide variety of other computational effects, like mutable state and exceptions, based on Filinski’s observation that composable continuations can be used to represent any monadic effect. On the more practical side, forms of delimited control have been implemented in real-world programming languages, and used in the design of libraries like for creating web servers.

However, the design space for adding composable continuations to a programming language is vast, and a number of different definitions of delimited control operators have been proposed. This has, in part, caused the theory and practice of delimited control to diverge somewhat from one another: the operators we most often study in theory are typically not the ones we use in practice. In this 30 minute talk, we will consider some of the fundamental, though subtle, differences in delimited control operators that appear in the literature and in programming languages, and some of the efforts to connect these together. We will also look at a common extension of delimited control that occurs in practice — the ability to delimit multiple different operations by name, much like exception handlers for specific subsets of errors — and how it provides another, novel approach for bridging the gap between the different frameworks.

2. A zoo of delimited control
At its most basic, delimited control is defined in two parts: (1) a delimiter that isolates some (potentially effectful) computation in a program, and (2) a control operator that part of the evaluation context (i.e. call-stack, control state, or “next steps in the program”) up to the nearest delimiter and creates a first-class evaluation context (i.e. captures a program, and (2) a control operator that isolates some (potentially effectful) computation in

At its most basic, delimited control is defined in two parts: (1) a delimiter that isolates some (potentially effectful) computation in a program, and (2) a control operator that

### Figure 1. Four different pairs of delimiters and control operators.

represents a closed off sub-computation, where the \( F \) operator is only capable of capturing the evaluation context \( 2 \times \bullet \leq 10 \). That way, the delimiter protects the surrounding context from the control operator, so that even in the larger program

\[
\text{if } \#(2 \times (F (\lambda k. M))) \leq 10 \text{ then red else blue}
\]

the control operator can still only see the context \( 2 \times \bullet \leq 10 \).

However, even within this general idea, we already have some choices as to how the delimiter and control operator interact with one another. After the control operator captures an evaluation context, does it remove the surrounding delimiter that marked the end of the context? Does the control operator create a continuation that includes the delimiter as well (e.g. the function \( \lambda x. \#(2 \times x) \leq 10 \))? Both of these questions can be answered either way, giving us four possibilities as summarized in Figure 1.

- \( +F - \) Only the surrounding delimiter is present, giving us the control operator (\( S \) and \( \langle \cdot \rangle \)) of Danvy and Filinski.
- \( -F + \) Only the delimiter in the created continuation is present, giving us the shift\(_0\) and reset\(_0\) operators \( [18, 20] \) \( (S_0 \) and \( \langle \cdot \rangle_0\)\).
- \( -F - \) Neither delimiter is present, giving us control\(_0\) and prompt\(_0\) \( (F_0 \) and \( \#_0\)), similar to caputo \( [12] \) or withSubCont \( [6] \).

These different between the four formulations of delimited control are not just minor details. The different interactions between the control operator and delimiter can have a major impact on the result of a program. For example, consider the following list traversal function which makes use of shift and reset \( [2] \):

\[
\text{traverse } xs = \langle \text{visit } xs \rangle
\]

\[
\text{where } \text{visit } [] = []
\]

\[
\text{visit } (x :: xs) = \text{visit } (S(\lambda k.x :: (k xs)))
\]

This function behaves like a copying identity function on lists, so that evaluating \( \text{traverse } [1, 2, 3] \) gives back the list \( [1, 2, 3] \). Contrarily, if we just replace the shift and reset in \( \text{traverse} \) with control and prompt, instead we end up with a list reversing function, so that evaluating \( \text{traverse } [1, 2, 3] \) gives back \( [3, 2, 1] \). We

\( V \) stands for a value and \( E \) stands for an undelimited evaluation context.
For example, consider the continuation swapping function in terms of shift.

\[
\text{swap } x = S(\lambda k_1. S(\lambda k_2. k_1 \; (k_2 \; x)))
\]

The net effect of this function is to just yield \(x\), so that evaluating \(\langle\langle \text{swap} \rangle \; 1 \times 2 \rangle\; 10\) gives 12. If instead we replace the shift in \(\text{swap}\) with \(\text{shift}_0\), we end up swapping the nearest two evaluation contexts delimited by \(\text{reset}_0\), so that evaluating \(\langle\langle \text{swap} \rangle \; 1 \times 2 \rangle\; 10\) results in 22 because we double after adding 10.

In general, shift and reset have been widely studied delimited control operators, in part due to the fact that they are defined by a simple \textit{continuation-passing style} (CPS) transformation in the ordinary \(\lambda\)-calculus \(\lambda^c\). This has enabled developments of high-level tools like an equational theory \(\lambda^c\) and type system \(\lambda^c\) for reasoning about programs. More recently, \(\text{shift}_0\) and \(\text{reset}_0\) have joined in this study with a similarly simple CPS transformation \(\lambda^c\), equational theory \(\lambda^c\), and type system \(\lambda^c\). On the other hand, implementations of delimited control in Racket \[10\], Haskell \[6\], and OCaml \[15\] have been based on control and prompt, or control and prompt style of operators\(\hat{0}\). So it seems that we prefer to study the \(\lambda^c\) family of delimited control, but favor implementing the \(\lambda^c\) family of operators.\(\hat{0}\)

An extension to delimited control often found in practice gives the ability to tag or name delimiters and control operations, so that the \(\lambda^c\) is in the continuation they create: instead of inserting an empty continuation. The trick used in this encoding bears a striking resemblance to Kiselyov \[14\] representation of \(\lambda^c\) using a sum type of two cases: a request for control. Here, we find that a framework of multiple prompts, Downen and Ariola \[5\] developed a dynamic binding \(\lambda\) and call-by-need evaluation \(\lambda\). The extension of delimited control is shared by the major implementations of delimited control in Racket \[10\], Haskell \[6\], and OCaml \[15\], and has been used to implement other effects like dynamic continuation variables, is found during the lookup of a delimiter, and the higher-order \(\text{shift}_0\) up to a specific delimiter is spelled out by a number of smaller operations. To recover the ordinary \(\text{shift}_0\) and \(\text{reset}_0\), we can limit ourselves to just one dynamic continuation variable, as in the single dynamic \(\text{tp}\) of the \(\mu\text{tp}\)-calculus.

The \(\lambda^c\) is also fully capable of expressing operators like control and \(\text{control}_0\) as well, by making use of at least two dynamic continuation variables. The intuition is that we can isolate one dynamic continuation variable, say \(\text{tp}\), for the purpose of returning and propagating values only. Then, the other dynamic variable(s) may be used for delimited control effects as before. The encoding of the multi-promt \(\text{control}_0\) and \(\text{promtp}_0\) are also given in Figure 6. Notice that the only difference between \(\lambda^c\) and \(\mu\text{tp}\) is in the continuation they create: instead of inserting an \(\alpha\) delimiter, the \(\lambda^c\) continuation “returns” to its calling context by using \(\text{tp}\). The encoding of \(\#\) also makes use of \(\text{tp}\), by binding \(\alpha\) to the “empty” continuation \(\text{tp}\) and then evaluating \(M\) in that empty continuation. The trick used in this encoding bears a striking resemblance to Kiselyov \[14\], and \(\alpha\) and the remainder of the dynamic environment. The net effect of this function is to just yield \(x\), substituting the prefix of more recent bindings in the dynamic environment for \(\Delta\), and then run \(M\) with the found continuation and the remainder of the dynamic environment.

\[
\langle M \rangle^c = \mu_0 \bar{\alpha} . [\bar{\alpha}] \; M
\]

\[
S^c_0 = \lambda h . \mu \beta . [\bar{\alpha}] \; \Delta . h \; (\lambda x . \mu_0 \bar{\alpha} . [\Delta] [\beta] \; x)
\]

\[
\#^c_0 (M) = \mu_0 [\text{tp}] . [\bar{\alpha}] \; (\text{tp}) \; M
\]

\[
F^c_0 = \lambda h . \mu \beta . [\bar{\alpha}] \; \Delta . h \; (\lambda x . \mu_0 [\text{tp}] . [\Delta] [\beta] \; x)
\]

\[\text{Figure 2. Encodings of the } S^c_0/\langle M \rangle^c \text{ and } F^c_0/\#^c_0 \text{ control operators in the } \lambda^c\text{-calculus.}\]

\[\mu_0 \bar{\alpha} . [\bar{\alpha}] (\mu_0 \bar{\alpha} . [\bar{\alpha}] (\mu_\beta . [\bar{\alpha}] [\Delta] \; \ldots) \times 2) + 10)\]

In this case, \(\beta\) receives only the current continuation up to the nearest delimiter, \(\alpha\), but then \(\alpha\) can focus on operators like \(\text{shift}\) and \(\text{reset}\) while still providing the capabilities of control.
4. Acknowledgements
Paul Downen and Zena M. Ariola have been supported by NSF grant CCF-0917329.

References

A. On marked stacks and dynamic environments
Perhaps the closest framework to the \(\lambda_{\mu}\)-calculus \(\textbf{5}\) is the Monadic Framework \(\textbf{6}\), which provides operations in the style of the \(\text{control}_0\) operator with multiple prompts. Both frameworks present the semantics of delimited control using both a \textit{continuation} (representing an evaluation context of the pure, call-by-value \(\lambda\)-calculus) and a \textit{meta-continuation} (which manages the delimiting effect of control in a program by storing continuations that are hidden behind a delimiter). The primary difference between the two systems is in the treatment and representation of meta-continuations.

In the \(\lambda_{\mu}\)-calculus, control delimiters are achieved by using dynamically bound continuation variables, and so the meta-continuation is represented as a dynamic environment. In other words, the type of the meta-continuation can be thought of as a list associating dynamic variables to continuations:

\[
\text{MetaContinuation} = [\text{DynVar} \times \text{Continuation}]
\]

Since the meta-continuation is a particular kind of dynamic environment, the only operation we have on the meta-continuation is to lookup a particular variable. For example, we would see the following result for dynamic variable lookup in a particular environment that binds the variables \(\alpha_1, \alpha_2, \alpha_3\) to the continuations \(k_1, k_2, k_3\) (the environment grows to the right, so \(\alpha_1 \mapsto k_1\) is the most recent binding):

\[
\begin{align*}
\alpha_3 & \mapsto k_3, \alpha_2 \mapsto k_2, \alpha_1 \mapsto k_1(\alpha_2) \\
& = ([\alpha_3 \mapsto k_3, k_2, (\alpha_1 \mapsto k_1)])
\end{align*}
\]

where the environment has been partitioned into three parts: (1) the prefix of the environment containing bindings that are more recent than the one we are looking for (\(\alpha_1 \mapsto k_1\)), (2) the continuation bound to the dynamic variable \(\alpha_2\) (\(k_2\)), and (3) the remaining dynamic environment containing older bindings (\(\alpha_3 \mapsto k_3\)).

In the Monadic Framework, control delimiters are achieved by storing a stack of both ordinary continuations and prompt markers. In other words, the type of the meta-continuation can be thought of as a list of prompt markers and continuations in an arbitrary arrangement:

\[
\text{MetaContinuation} = [\text{Prompt} + \text{Continuation}]
\]

Fundamentally, we have two different operations on this meta-continuation: sending a value to the next available continuation, and splitting the meta-continuation at a specified prompt. Sending a value involves skipping past prompts in the meta-continuation until a continuation is found. For example, in a particular stack \([k_3, p_3, p_2, k_2, k_1, p_1]\), of the prompts \(p_1, p_2, p_3\) and continuations \(k_1, k_2, k_3\) (the stack here grows to the right, so \(p_1\) is the most recent) we would have:

\[
\text{send } x [k_3, p_3, p_2, k_2, k_1, p_1] = k_1 \ x [k_3, p_3, p_2, k_2]
\]

On the other hand, splitting the same stack at the prompt \(p_2\) gives:

\[
\text{split } p_2 [k_3, p_3, p_2, k_2, k_1, p_1] = ([k_3, p_3], [k_2, k_1, p_1])
\]

where the stack has been partitioned into everything more recent than the prompt we’re splitting (\([k_2, k_1, p_1]\)) and everything older (\([k_3, p_3]\)).

\[\textbf{5}\]The Monadic Framework \(\textbf{6}\) actually uses two separate operations for splitting, returning just the first part and just the second part after the split, but it is equivalent to the presentation here.
Now, let’s consider how the meta-continuations in these two frameworks might relate to one another. On the one hand, it appears straightforward to embed a dynamic environment into a marked stack: just flatten the list and remove the pairing that associates variables to continuations. For example, we have the following embedding of the above dynamic environment:

\[
\text{\{\hat{a}_3 \mapsto k_3, \hat{a}_2 \mapsto k_2, \hat{a}_1 \mapsto k_1\}} = [k_3, \hat{a}_3, k_2, \hat{a}_2, k_1, \hat{a}_1]
\]

In this view, the dynamic environment can be seen like a discipline imposed on the meta-continuation stacks of the Monadic Framework. On the other hand, it is not so obvious how to encoding a free-form stack into a dynamic environment, or how to provide the two different operations, \textit{send} and \textit{split}, in terms of just variable lookup. The key is to choose a single dynamic continuation variable, \(\hat{\theta}\), that represents “returning a value to the next available continuation” and to fill in the gaps in a marked stack. That way, a stack can be converted into an environment by associating every continuation with the variable \(\hat{\theta}\), and every prompt with the “empty” continuation \(k_{\hat{\theta}}\) which just passes the value it receives to the next continuation bound to \(\hat{\theta}\) in the environment:

\[
k_{\hat{\theta}} x \gamma = k x \gamma' \quad \text{where} \quad (\gamma', k, ...) = \gamma(\hat{\theta})
\]

For example, we would have the following embedding of a stack:

\[
[k_3, p_3, p_2, k_2, k_1, p_1] = [\hat{\theta} \mapsto k_3, \hat{\theta} \mapsto k_{\hat{\theta}}, p_2 \mapsto k_{\hat{\theta}}, \hat{\theta} \mapsto k_2, \hat{\theta} \mapsto k_1, p_1 \mapsto k_{\hat{\theta}}]
\]

Now we can achieve both operations on marked stacks just in terms of the single variable lookup operation: \textit{send} looks up \(\hat{\theta}\) and passes a value along to the continuation it finds, thereby discarding the prefix of non-\(\hat{\theta}\) prompts in the way, and \textit{split} performs the usual variable lookup on the chosen prompt, finding the two partitions of the environment along with a trivial empty continuation.

\[
send x \gamma = k x \gamma' \quad \text{where} \quad (\gamma', k, ...) = \gamma(\hat{\theta})
\]

\[
split \gamma = (\gamma_1, \gamma_2) \quad \text{where} \quad (\gamma_2, ..., \gamma_1) = \gamma(p)
\]

The analogy between marked stacks and dynamic environments helps to explain the encoding of \(F_0\) and \(\#\) from Figure 2. First, let’s begin with an abstract machine description of \(F_0\) and \(\#\), similar to the semantics given by the Monadic Framework. We’ll use the simplified configuration, \((M, E, D)\), consisting of a term \(M\), an ordinary call-by-value \(\lambda\)-calculus evaluation context \(E\), and a marked stack of evaluation contexts and prompts \(D\). The two operators can then be described by the following transitions:

\[
\langle \#\rangle (M, E, D) \rightsquigarrow (M, [\hat{\theta}], D : E : \hat{\alpha})
\]

\[
\langle F_0 \rangle (V, E, D) \rightsquigarrow (V f, \hat{\theta}, D_2) \quad \text{where}
\]

\[
(D_2, D_1) = split \hat{\alpha} D
\]

\[
(f V, E', D') \rightsquigarrow (V, E, (D' : E') ++ D_1)
\]

\[
(V, \hat{\theta}, E : D) \rightsquigarrow (V, E, D)
\]

\[
(V, [\hat{\theta}], \hat{\alpha} : D) \rightsquigarrow (V, [\hat{\theta}], D)
\]

\[
E ::= \square | E t | V E
\]

\[
D ::= \square | E[^\#\#\#D] | D_0
\]

\[
D_0 ::= \square | D[^\#\#\#\#(D_0)] \quad \text{where} \hat{\beta} \neq \hat{\alpha}
\]

\[
D[E[(\lambda x.M) V]] \rightarrow D[E[M(V/x)]]
\]

\[
D[E[^\#\#\#(V)]] \rightarrow D[E[V]]
\]

\[
D[E[^\#\#\#(D_0', E'[\#\#\#(V)]]]) \rightarrow D[E[V \ (\lambda x.D'_0'[E'[x]])]]
\]

Figure 3. Call-by-value evaluation contexts and operational semantics of the \(F_0\) and \(\#\) control operators.

By encoding the stack into a dynamic environment, we now get the modified machine:

\[
\langle \#\rangle (M, E, D) \rightsquigarrow (M, [\hat{\theta}], D [\hat{\theta}] \mapsto E) [\hat{\alpha}] \mapsto (\hat{\theta} [\hat{\theta}])
\]

\[
\langle F_0 \rangle (V, E, D) \rightsquigarrow (V f, [\hat{\theta}], D_2) \quad \text{where}
\]

\[
(D_2, [\hat{\theta}], D_1) = D(\hat{\alpha})
\]

\[
(f V, E', D') \rightsquigarrow (V, E, D' [\hat{\theta}] \mapsto E') ++ D_1
\]

\[
(V, [\hat{\theta}], D) \mapsto (V, E, D') \quad \text{where}
\]

\[
(D', E, \hat{\alpha}) = D(\hat{\theta})
\]

Notice that, like the encoding of \(\#\) in Figure 2, the \(\#\) operation binds \(\hat{\alpha}\) to the current evaluation context and \(\hat{\theta}\) to the “empty” context \([\hat{\theta}][\hat{\theta}]\), and then runs \(M\) in the empty context. The \(F_0\) operator looks up the binding of \(\hat{\alpha}\) in the current environment to create the continuation \(f\), and when \(f\) is called it binds its calling context to \(\hat{\theta}\) before further extending the environment. Again, notice in both the encoding of \(F_0\) and the machine above, the dynamic continuation variable that we look up is \textit{different} from the one that we use to bind the calling context of the created function. Additionally, the steps which look for an evaluation context in the stack have been replaced with a dynamic lookup of \(\hat{\theta}\).

Using the encodings in Figure 2 and the semantics for the \(\lambda\mu_{IO}\)-calculus \(\#\), the derived operational semantics for control with multiple prompts given in Figure 3 shows that the encodings give the intended behavior. It’s important that the chosen \(\hat{\theta}\) dynamic variable is never used as “prompt,” so that \(\hat{\theta}\) and \(F_0\) are disallowed. In other words, the chosen \(\hat{\theta}\) variable is hidden from the view of the programmer. This way, any intermediate binding of \(\hat{\theta}\) in a program is effectively invisible to every \(F_0\) operation and can be ignored, supporting the idea that the function created by \(F_0\) returns directly to its calling context without introducing a delimiter. We can then restrict ourselves to two dynamic variables—an arbitrary dynamic for creating prompt markers and the hidden \(\hat{\theta}\) for returning values — to produce the single-prompt control operator. With this restriction, the semantics in Figure 3 simplifies to the rule for \(F_0\) and \(\#\) in Figure 4.

---

\(^4\)Here, \(\#\) \((M)\) behaves like \textit{pushPrompt} \(\hat{\alpha}\) \(M\) from the Monadic Framework, and \(F_0\) \(V\) is similar to \textit{withSubCont} \(\hat{\alpha}\) \(V\) that wraps up the captured continuation into a call-by-value function that immediately invokes it with \textit{pushSubCont}.

2014/9/17