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 [8, 9] 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 [6, 10, 15], and used in the design of libraries like for creating web servers [10].

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 [3, 4, 7, 12, 21]. 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 *captures* 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 *continuation* that acts as a functional representation of the context. Taken together, the delimiter serves to pose a limit on the range of influence that the control operator can exert over the future course of the program. In other words, if we have the delimiter, $\#$, and control operator, $\mathcal{F}$, then an expression like

$$\#(2 \times (\mathcal{F}\ (\lambda k.M)) \leq 10)$$

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

$$\mathbf{if}\ \#(2 \times (\mathcal{F}\ (\lambda k.M)) \leq 10)\ \mathbf{then}\ red\ \mathbf{else}\ blue$$

the control operator can still only see the context $2 \times \square \leq 10$.

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 [6] as summarized in Figure 1:[1]

$$\begin{aligned}
\langle E[\mathcal{S}\ V]\rangle &\mapsto \langle V\ (\lambda x.\langle E[x]\rangle)\rangle \\
\#(E[\mathcal{F}\ V]) &\mapsto \#(V\ (\lambda x.E[x])) \\
\langle E[\mathcal{S}_0\ V]\rangle_0 &\mapsto V\ (\lambda x.\langle E[x]\rangle_0) \\
\#_0(E[\mathcal{F}_0\ V]) &\mapsto V\ (\lambda x.E[x])
\end{aligned}$$

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

- $+\mathcal{F}+$ Both delimiters are present, giving us the shift and reset operators ($\mathcal{S}$ and $\langle\_\rangle$) of Danvy and Filinski [3, 4].
- $+\mathcal{F}-$ Only the surrounding delimiter is present, giving us the control and prompt operators ($\mathcal{F}$ and $\#$) of Felleisen [7, 21].
- $-\mathcal{F}+$ Only the delimiter in the created continuation is present, giving us the shift_0 and reset_0 operators [18, 20] ($\mathcal{S}_0$ and $\langle\_\rangle_0$).
- $-\mathcal{F}-$ Neither delimiter is present, giving us $\mathsf{control}_0$ and prompt_0 ($\mathcal{F}_0$ and $\#_0$), similar to *cupto* [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]:

$$\begin{aligned}
&traverse\ xs = \langle visit\ xs\rangle \\
&\quad \mathbf{where}\ visit\ [] \qquad\quad = [] \\
&\qquad\qquad visit\ (x :: xs) = visit\ (\mathcal{S}(\lambda k.x :: (k\ xs)))
\end{aligned}$$

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

[1] V stands for a value and E stands for an undelimited evaluation context.

can see similar differences between shift and shift_0. For example, consider the continuation swapping function in terms of shift, where we capture two continuations and then apply them in the reverse order:

$$swap\ x = \mathcal{S}(\lambda k_1.\mathcal{S}(\lambda k_2.k_1\ (k_2\ x)))$$

The net effect of this function is to just yield x, so that evaluating $\langle\langle(swap\ 1) \times 2\rangle + 10\rangle$ gives 12. If instead we replace the shift in $swap$ with shift_0, we end up swapping the nearest two evaluation contexts delimited by reset_0, so that evaluating $\langle\langle(swap\ 1) \times 2\rangle_0 + 10\rangle_0$ 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 *continuation-passing style* (CPS) transformation in the ordinary λ-calculus [3, 4]. This has enabled developments of high-level tools like an equational theory [13] and type system [3] for reasoning about programs. More recently, shift_0 and reset_0 have joined in this study with a similarly simple CPS transformation [18], equational theory [17], and type system [18]. On the other hand, implementations of delimited control in Racket [10], Haskell [6], and OCaml [15] have been based on control and prompt, or $\mathsf{control}_0$ and prompt_0 style of operators.[2] So it seems that we prefer to study the $*\mathcal{F}+$ family of delimited control, but favor implementing the $*\mathcal{F}-$ family. To bridge the gap, there have been several different encodings of the $*\mathcal{F}-$ family of operators into the $*\mathcal{F}+$ family. Shan [20] gives an encoding of control into shift (as well as encodings of shift_0 and $\mathsf{control}_0$) using a recursive type of continuation. Kiselyov [14] gives an alternative encoding of control in terms of shift using a sum type of two cases: a request for control or a returned value. Biernacki *et al.* [2] gives an abstract machine for control, from which a direct implementation using shift can be derived.

3. Multiple prompts and the power of two

An extension to delimited control often found in practice gives the ability to tag or name delimiters and control operations, so that the two only interact when they share the same name. For instance, the control operator $\mathcal{F}^{p_1}$ in the term

$$\#^{p_1}(\#^{p_2}((\mathcal{F}^{p_1}(\lambda k.\ldots)) \times 2) + 10)$$

can capture the context $\#^{p_2}(\Box \times 2) + 10$ up to the prompt p_1, rather than just the context $\Box \times 2$ delimited by the nearest prompt. This 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 binding [16] and call-by-need evaluation [11].

In order to better understand the behavior of delimited control with multiple prompts, Downen and Ariola [5] developed a framework, called the $\lambda\widehat{\mu}_0$-calculus, for expressing such operators. It extends the $\lambda\mu\widehat{\mathsf{tp}}$-calculus [1], a language for expressing shift and reset based on Parigot's $\lambda\mu$-calculus [19], with multiple dynamically-bound continuation variables. The $\lambda\widehat{\mu}_0$-calculus presents a more fine-grained explanation of delimited control with a number of more low-level operations:

- $\mu\alpha.c$: capture the current continuation and substitute it for α.
- $[q]M$: run M with q as the current continuation, so that when M returns a value it is passed to q.
- $\mu_0\widehat{\alpha}.c$: defer the current continuation by binding it to the dynamic variable $\widehat{\alpha}$.

[2] Of note, there are two native implementations of shift and reset: in OchaCaml (an extension of Caml light) and more recently in Scala.

$$\begin{aligned}
\langle M\rangle_0^{\widehat{\alpha}} &= \mu_0\widehat{\alpha}.[\widehat{\alpha}]M \\
\mathcal{S}_0^{\widehat{\alpha}} &= \lambda h.\mu\beta.[\widehat{\alpha}]_0\Delta.h\ (\lambda x.\mu_0\widehat{\alpha}.[\Delta][\beta]x) \\
\#_0^{\widehat{\alpha}}(M) &= \mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mu_0\widehat{\alpha}.[\widehat{\mathsf{tp}}]M \\
\mathcal{F}_0^{\widehat{\alpha}} &= \lambda h.\mu\beta.[\widehat{\alpha}]_0\Delta.h\ (\lambda x.\mu_0\widehat{\mathsf{tp}}.[\Delta][\beta]x)
\end{aligned}$$

Figure 2. Encodings of the $\mathcal{S}_0^{\widehat{\alpha}}/\langle M\rangle_0^{\widehat{\alpha}}$ and $\mathcal{F}_0^{\widehat{\alpha}}/\#_0^{\widehat{\alpha}}$ control operators in the $\lambda\widehat{\mu}_0$-calculus.

- $[\widehat{\alpha}]_0\Delta.M$: lookup the continuation dynamically bound to the nearest $\widehat{\alpha}$, substituting the prefix of more recent bindings in the dynamic environment for Δ, and then run M with the found continuation and the remainder of the dynamic environment.
- $[\Delta]c$: extend the dynamic environment with all the bindings of Δ and then run c.

Intuitively, the previous program using two prompts may be expressed by a similar program in the $\lambda\widehat{\mu}_0$-calculus using two dynamic continuation variables:

$$\mu_0\widehat{\alpha}_1.[\widehat{\alpha}_1]((\mu_0\widehat{\alpha}_2.[\widehat{\alpha}_2]((\mu\beta.[\widehat{\alpha}_1]_0\Delta.\ldots) \times 2)) + 10)$$

In this case, β receives only the current continuation up to the nearest delimiter, $[\widehat{\alpha}_2](\Box \times 2)$. The rest of the captured context, $[\widehat{\alpha}_1]((\mu\widehat{\alpha}_2.\Box) + 10)$, which is hidden behind dynamically bound continuation variables, is found during the lookup of $\widehat{\alpha}$ and substituted for Δ.

The $\lambda\widehat{\mu}_0$-calculus expresses delimited control in a similar style as shift_0: we can remove a surrounding delimiter due to a control effect (as in the dynamic lookup $[\widehat{\alpha}]_0\Delta.M$), but the syntax of the language forces continuations to insert a delimiter to be used (as in $\mu_0\widehat{\beta}.[\alpha]M$). We can therefore give an encoding of shift_0 and reset_0 with multiple prompts in Figure 2, where the named reset_0 delimiter binds the current continuation to a dynamic variable bearing that name, and the higher-order shift_0 up to a specific delimiter is spelled out by a number of smaller operations. To recover the ordinary shift_0 and reset_0, we can limit ourselves to just one dynamic continuation variable, as in the single dynamic $\widehat{\mathsf{tp}}$ of the $\lambda\mu\widehat{\mathsf{tp}}$-calculus.

However, the $\lambda\widehat{\mu}_0$-calculus is also fully capable of expressing operators like control and $\mathsf{control}_0$ as well, by making use of at least *two* dynamic continuation variables. The intuition is that we isolate one dynamic continuation variable, say $\widehat{\mathsf{tp}}$, 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-prompt $\mathsf{control}_0$ and prompt_0 are also given in Figure 2. Notice that the only difference between $\mathcal{S}_0^{\widehat{\alpha}}$ and $\mathcal{F}_0^{\widehat{\alpha}}$ is in the continuation they create: instead of inserting an $\widehat{\alpha}$ delimiter, the $\mathcal{F}_0^{\widehat{\alpha}}$ continuation "returns" to its calling context by using $\widehat{\mathsf{tp}}$. The encoding of $\#_0^{\widehat{\alpha}}$ also makes use of $\widehat{\mathsf{tp}}$, by binding $\widehat{\alpha}$ to the "empty" continuation $[\widehat{\mathsf{tp}}]\Box$ and then evaluating M in that empty continuation. The trick used in this encoding bears a striking resemblance to Kiselyov [14] representation of control, where there are two possible return values: an ordinary return and a request for control. Here, we find that a framework of multi-prompt delimited control already has this facility built-in: we can always isolate one prompt as the special purpose "return" prompt. This means that shift_0 is capable of encoding $\mathsf{control}_0$ by using two prompts. Additionally, in the presence of multiple prompts we can focus on operators like shift and shift_0 while still providing the capabilities of control.

4. Acknowledgements

Paul Downen and Zena M. Ariola have been supported by NSF grant CCF-0917329.

References

[1] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A type-theoretic foundation of delimited continuations. *Higher-Order and Symbolic Computation*, 22(3):233–273, 2009.

[2] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. *A dynamic continuation-passing style for dynamic delimited continuations*. BRICS, Department of Computer Science, Univ., 2005.

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

[4] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160. ACM Press, 1990.

[5] Paul Downen and Zena M. Ariola. Delimited control and computational effects. *Journal of Functional Programming*, pages 1–55, 2014.

[6] R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(06):687–730, 2007.

[7] Matthias Felleisen. The theory and practice of first-class prompts. In *Principles of Programming Languages '88*, pages 180–190, 1988.

[8] Andrzej Filinski. Representing monads. In *Principles of Programming Languages '94*, pages 446–457. ACM, 1994.

[9] Andrzej Filinski. Representing layered monads. In *Principles of Programming Languages '99*, pages 175–188, 1999.

[10] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, volume 1, pages 165–176, 2007.

[11] Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. In *Proceedings of POPL '09*, pages 153–164, New York, NY, USA, 2009. ACM.

[12] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Functional Programming Languages and Computer Architecture '95*, pages 12–23, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7.

[13] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 177–188, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7.

[14] Oleg Kiselyov. How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Technical report, Technical Report 611, Computer Science Department, Indiana University, Bloomington, Indiana, 2005.

[15] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely: System description. *Functional and Logic Programming*, pages 304–320, 2010.

[16] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited dynamic binding. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 26–37, New York, NY, USA, 2006. ACM.

[17] Marek Materzok. Axiomatizing subtyped delimited continuations. In *CSL*, pages 521–539, 2013.

[18] Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 81–93, New York, NY, USA, 2011. ACM.

[19] Michel Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning*, pages 190–201. Springer, 1992.

[20] C. Shan. Shift to control. In *Workshop on Scheme and Functional Programming*, page 99, 2004.

[21] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

A. On marked stacks and dynamic environments

Perhaps the closest framework to the $\lambda\widehat{\mu}_0$-calculus [5] is the Monadic Framework [6], which provides operations in the style of the $\mathsf{control}_0$ operator with multiple prompt_0s. Both frameworks present the semantics of delimited control using both a *continuation* (representing an evaluation context of the pure, call-by-value λ-calculus) and a *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 $\lambda\widehat{\mu}_0$-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:

$$MetaContinuation = [DynVar * Continuation]$$

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 $\widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\alpha}_3$ to the continuations k_1, k_2, k_3 (the environment grows to the right, so $\widehat{\alpha}_1 \mapsto k_1$ is the most recent binding)

$$\begin{aligned} &[\widehat{\alpha}_3 \mapsto k_3, \widehat{\alpha}_2 \mapsto k_2, \widehat{\alpha}_1 \mapsto k_1](\widehat{\alpha}_2) \\ &= ([\widehat{\alpha}_3 \mapsto k_3], k_2, [\widehat{\alpha}_1 \mapsto k_1]) \end{aligned}$$

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 ($[\widehat{\alpha}_1 \mapsto k_1]$), (2) the continuation bound to the dynamic variable $\widehat{\alpha}_2$ (k_2), and (3) the remaining dynamic environment containing *older* bindings ($[\widehat{\alpha}_3 \mapsto k_3]$).

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:

$$MetaContinuation = [Prompt + Continuation]$$

Fundamentally, we have two different operations on this meta-continuation: sending a value to the next available continuation, and splitting[3] 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:

$$send\ x\ [k_3, p_3, p_2, k_2, k_1, p_1] = k_1\ x\ [k_3, p_3, p_2, k_2]$$

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

$$split\ p_2\ [k_3, p_3, p_2, k_2, k_1, p_1] = ([k_3, p_3], [k_2, k_1, p_1])$$

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 ($[k_3, p_3]$).

[3] The Monadic Framework [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:

$$[\widehat{\alpha}_3 \mapsto k_3, \widehat{\alpha}_2 \mapsto k_2, \widehat{\alpha}_1 \mapsto k_1] = [k_3, \widehat{\alpha}_3, k_2, \widehat{\alpha}_2, k_1, \widehat{\alpha}_1]$$

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, *send* and *split*, in terms of just variable lookup. The key is to choose a single dynamic continuation variable, say $\widehat{\mathsf{tp}}$, 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 $\widehat{\mathsf{tp}}$, and every prompt with the "empty" continuation $k_{\widehat{\mathsf{tp}}}$ which just passes the value it receives to the next continuation bound to $\widehat{\mathsf{tp}}$ in the environment:

$$k_{\widehat{\mathsf{tp}}}\ x\ \gamma = k\ x\ \gamma' \qquad \textbf{where}\ (\gamma', k, \_) = \gamma(\widehat{\mathsf{tp}})$$

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

$$\begin{aligned}&[k_3, p_3, p_2, k_2, k_1, p_1]\\ &= [\widehat{\mathsf{tp}} \mapsto k_3, p_3 \mapsto k_{\widehat{\mathsf{tp}}}, p_2 \mapsto k_{\widehat{\mathsf{tp}}}, \widehat{\mathsf{tp}} \mapsto k_2, \widehat{\mathsf{tp}} \mapsto k_1, p_1 \mapsto k_{\widehat{\mathsf{tp}}}]\end{aligned}$$

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

$$\begin{aligned}&\mathit{send}\ x\ \gamma = k\ x\ \gamma' &&\textbf{where}\ (\gamma', k, \_) = \gamma(\widehat{\mathsf{tp}})\\ &\mathit{split}\ p\ \gamma = (\gamma_1, \gamma_2) &&\textbf{where}\ (\gamma_2, \_, \gamma_1) = \gamma(p)\end{aligned}$$

The analogy between marked stacks and dynamic environments helps to explain the encoding of $\mathcal{F}_0^{\widehat{\alpha}}$ and $\#_0^{\widehat{\alpha}}$ from Figure 2. First, let's begin with an abstract machine description of $\mathcal{F}_0^{\widehat{\alpha}}$ and $\#_0^{\widehat{\alpha}}$, similar to the semantics given by the Monadic Framework [6]. We'll use the simplified configuration, $\langle M, E, D\rangle$, consisting of a term M, an ordinary call-by-value λ-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:[4]

$$\langle \#_0^{\widehat{\alpha}}(M), E, D\rangle \rightsquigarrow \langle M, \Box, D : E : \widehat{\alpha}\rangle$$

$$\begin{aligned}&\langle \mathcal{F}_0^{\widehat{\alpha}}\ V, E, D\rangle \rightsquigarrow \langle V\ f, \Box, D_2\rangle \quad \textbf{where}\\ &\quad (D_2, D_1) = \mathit{split}\ \widehat{\alpha}\ D\\ &\quad \langle f\ V, E', D'\rangle \rightsquigarrow \langle V, E, (D' : E') +\!\!+ D_1\rangle\end{aligned}$$

$$\begin{aligned}\langle V, \Box, E : D\rangle &\rightsquigarrow \langle V, E, D\rangle\\ \langle V, \Box, \widehat{\alpha} : D\rangle &\rightsquigarrow \langle V, \Box, D\rangle\end{aligned}$$

[4] Here, $\#_0^{\widehat{\alpha}}(M)$ behaves like $\mathit{pushPrompt}\ \widehat{\alpha}\ M$ from the Monadic Framework, and $\mathcal{F}_0^{\widehat{\alpha}}\ V$ is similar to $\mathit{withSubCont}\ \widehat{\alpha}\ V$ that wraps up the captured continuation into a call-by-value function that immediately invokes it with $\mathsf{pushSubCont}$.

$$\begin{aligned}E &::= \Box \mid E\ t \mid V\ E\\ D &::= \Box \mid E[\#_0^{\widehat{\alpha}}(D)]\\ D_{\widehat{\alpha}} &::= \Box \mid E[\#_0^{\widehat{\beta}}(D_{\widehat{\alpha}})] \qquad \textbf{where}\ \widehat{\beta} \neq \widehat{\alpha}\end{aligned}$$

$$\begin{aligned}D[E[(\lambda x.M)\ V]] &\mapsto D[E[M\{V/x\}]]\\ D[E[\#_0^{\widehat{\alpha}}(V)]] &\mapsto D[E[V]]\\ D[E[\#_0^{\widehat{\alpha}}(D'_{\widehat{\alpha}}[E'[\mathcal{F}_0^{\widehat{\alpha}}\ V]])]] &\mapsto D[E[V\ (\lambda x.D'_{\widehat{\alpha}}[E'[x]])]]\end{aligned}$$

Figure 3. Call-by-value evaluation contexts and operational semantics of the $\mathcal{F}_0^{\widehat{\alpha}}$ and $\#_0^{\widehat{\alpha}}$ control operators.

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

$$\langle \#_0^{\widehat{\alpha}}(M), E, D\rangle \rightsquigarrow \langle M, [\widehat{\mathsf{tp}}]\Box, D[\widehat{\mathsf{tp}} \mapsto E][\widehat{\alpha} \mapsto [\widehat{\mathsf{tp}}]\Box]\rangle$$

$$\begin{aligned}&\langle \mathcal{F}_0^{\widehat{\alpha}}\ V, E, D\rangle \rightsquigarrow \langle V\ f, [\widehat{\mathsf{tp}}]\Box, D_2\rangle \quad \textbf{where}\\ &(D_2, [\widehat{\mathsf{tp}}]\Box, D_1) = D(\widehat{\alpha})\\ &\quad \langle f\ V, E', D'\rangle \rightsquigarrow \langle V, E, D'[\widehat{\mathsf{tp}} \mapsto E'] +\!\!+ D_1\rangle\end{aligned}$$

$$\begin{aligned}&\langle V, [\widehat{\mathsf{tp}}]\Box, D\rangle \rightsquigarrow \langle V, E, D'\rangle \quad \textbf{where}\\ &\quad (D', E, \_) = D(\widehat{\mathsf{tp}})\end{aligned}$$

Notice that, like the encoding of $\#_0^{\widehat{\alpha}}$ in Figure 2, the $\#_0^{\widehat{\alpha}}(M)$ operation binds $\widehat{\mathsf{tp}}$ to the current evaluation context and $\widehat{\alpha}$ to the "empty" context $[\widehat{\mathsf{tp}}]\Box$, and then runs M in the empty context. The $\mathcal{F}_0^{\widehat{\alpha}}$ operator looks up the binding of $\widehat{\alpha}$ in the current environment to create the continuation f, and when f is called it binds its calling context to $\widehat{\mathsf{tp}}$ before further extending the environment. Again, notice that in both the encoding of $\mathcal{F}_0^{\widehat{\alpha}}$ and the machine above, the dynamic continuation variable that we look up is *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 $\widehat{\mathsf{tp}}$.

Using the encodings in Figure 2 and the semantics for the $\lambda\widehat{\mu}_0$-calculus [5], the derived operational semantics for $\mathsf{control}_0$ with multiple prompt_0s given in Figure 3 shows that the encodings give the intended behavior. It's important that the chosen $\widehat{\mathsf{tp}}$ dynamic variable is never used as "prompt," so that $\#_0^{\widehat{\mathsf{tp}}}$ and $\mathcal{F}_0^{\widehat{\mathsf{tp}}}$ are disallowed. In other words, the chosen $\widehat{\mathsf{tp}}$ variable is hidden from the view of the programmer. This way, any intermediate binding of $\widehat{\mathsf{tp}}$ in a program is effectively invisible to every $\mathcal{F}_0^{\widehat{\alpha}}$ operation and can be ignored, supporting the idea that the function created by $\mathcal{F}_0^{\widehat{\alpha}}$ 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 $\widehat{\mathsf{tp}}$ for returning values — to produce the single-prompt $\mathsf{control}$ operator. With this restriction, the semantics in Figure 3 simplifies to the rule for $\mathcal{F}_0$ and $\#_0$ in Figure 1.