A Systematic Approach to Delimited Control with Multiple Prompts

Paul Downen and Zena M. Ariola

University of Oregon
{pdownen,ariola}@cs.uoregon.edu

Abstract. We formalize delimited control with multiple prompts, in the style of Parigot's $\lambda\mu$-calculus, through a series of incremental extensions by starting with the pure λ-calculus. Each language inherits the semantics and reduction theory of its parent, giving a systematic way to describe each level of control.

Keywords: Delimited control, dynamic variables, shift, reset, multiple prompts.

1 Introduction

Control operators have become an integral part of modern programming languages. In particular, the flexible abstraction of continuation-based control is becoming more mainstream in high-level languages. The classic control operator is call-with-current-continuation, or call/cc, which has appeared in languages such as Scheme and Ruby. call/cc allows the programmer to capture the surrounding context of an expression, creating a continuation that serves as a return point to "the rest of the program" from where call/cc was called. This style of control abstraction is called *abortive*, since invoking a continuation captured by call/cc aborts the computation currently in progress, and immediately returns to the context stored in the continuation. Even though call/cc is a very flexible control operator, it has limits. For example, call/cc alone is not enough to simulate mutable state in an otherwise state-free language.

Compared to abortive control, delimited control provides a more powerful abstraction. The difference of delimited control is that the continuation behaves like a normal function, so that multiple continuations may be composed together. In addition, the scope of the control operator can be managed by setting a *prompt*, limiting the context that can be captured. The shift and reset operators, as presented by Danvy and Filinski [5], are expressive enough to simulate mutable state. In fact, Filinski [11,12] showed that the combination of shift and reset is enough to give a direct style encoding for any effect written in monadic style, as well as several layered effects.

An interesting extension of delimited control is the addition of multiple prompts that can each delimit a different portion of the context. Dybvig, Peyton Jones, and Sabry [8] define a general framework for delimited control in the presence of multiple prompts, in which higher-level control operators may be defined. They provide an operational semantics and a monadic translation into a pure λ-calculus extended with stacks, as well as an implementation of the monadic effect in Haskell. A direct implementation of delimited control with multiple prompts in OCaml is given by Kiselyov [14]. In addition, Kiselyov, Shan, and Sabry [15] give a language that combines both delimited control and dynamic variables, showing that the two effects interact in subtle ways. Garcia *et al.* [13] showed that delimited control with multiple prompts can represent call-by-need evaluation.

The goal of this paper is to provide a reduction theory for delimited control with multiple prompts. Ariola *et al.* have formalized abortive and delimited control [2] in the style of Parigot's call-by-value $\lambda\mu$, leading to a calculus called $\lambda\mu\widehat{\mathsf{tp}}$. We use $\lambda\mu\widehat{\mathsf{tp}}$ as a reference point since it has a well-understood reduction theory that directly expresses the operational semantics. By extending $\lambda\mu\widehat{\mathsf{tp}}$ with multiple prompts, we clearly delineate the reduction of delimited control with multiple prompts in a way that is not apparent in the usual presentations based on operational semantics. Our approach is to build up to the expressive power of shift and shift_0 with multiple prompts in incremental steps, while using intermediate languages as stepping stones. We start with the pure λ-calculus and make small extensions to each language that are compatible with the previous semantics. Separate concerns, such as binding and capture, are explicitly apparent in the syntax of the language. The end result is a calculus that expresses delimited control with multiple prompts, which arises naturally from the representation of the semantics. Our contributions are:

- A better understanding of the dynamic nature of the prompt, in the context of delimited control with a single prompt. We express this in terms of an intermediate language with one dynamic variable that avoids recursive bindings.
- A set of small, incremental extensions of $\lambda\mu\widehat{\mathsf{tp}}$, providing more expressive languages that are compatible with the existing semantics. Each extension enables direct encodings of additional, useful language constructs, and arises as a natural extension of a less expressive language or intermediate language.
- A reduction theory for control with multiple prompts that is sound with respect to the *continuation passing style* (CPS) semantics and expressive enough to lead to the final answer. This reduction theory is compatible with the one of $\lambda\mu\widehat{\mathsf{tp}}$.

The overall strategy of the paper is as follows. In Sections 2, 3, 4, 6, 7, and 9, we define our languages of interest. We start with the λ-calculus (in 2), and extend it with control ($\lambda\mu$ in 3) and then with delimited control ($\lambda\mu\widehat{\mathsf{tp}}$ in 4). Then, we branch out in two separate directions, extending $\lambda\mu\widehat{\mathsf{tp}}$ with multiple prompts ($\lambda\widehat{\mu}$ in 6) and also transparent prompts ($\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}$ in 7). Finally, we bring $\lambda\widehat{\mu}$ and $\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}$ together, giving us a language of delimited control and multiple prompts ($\lambda\widehat{\mu}^{\uparrow}$ in 9). We present the semantics of the new languages in three different ways: first as a *CPS transformation* from the source language to the pure λ-calculus, then as a set of *reduction rules*, and finally as an *operational semantics*. The CPS transformation implements a big-step evaluator for the language written in the λ-calculus, and is used as our primary reference point for the definition of the semantics. The reduction rules are a set of local program transformations in the source language that correspond to reductions performed in the CPS transformed program. The operational semantics arise as both a restriction on the reduction rules and as the equivalent small-step evaluator for the CPS transformation, and is derived by defunctionalizing the continuation of the CPS [19,4]. We wrap up these sections with a discussion on expressiveness by encoding control operators in the language. In Sections 5 and 8 we present two intermediate languages which are used as stepping stones for defining the CPS transformations of our primary languages, and provide a good framework for designing extensions.

2 Lambda Calculus: λ

The syntax of λ-calculus includes variables, function abstraction, and function application. Unless otherwise specified, we let the set of *Values* be $V ::= x \mid \lambda x.t$.

$$t \in Term ::= V \mid t_1\ t_2 \qquad\qquad V \in Value ::= x \mid \lambda x.t$$

In this paper we are going to focus on the call-by-value setting, which restricts substitution to values, as described by the β_v reduction rule: $(\lambda x.t)\ V \to t\{V/x\}$. An alternative way of presenting the semantics is to perform a translation which hard-wires the evaluation strategy into the term itself. The transformation is called *continuation passing style* (CPS); it splits a program into the current work to be done and the rest of the computation, which is called a *continuation*. The call-by-value CPS transform of the λ-calculus is defined as follows:

$$\mathcal{C}_\lambda[\![x]\!]k = k\ x \quad \mathcal{C}_\lambda[\![\lambda x.t]\!]k = k\ \lambda x.\mathcal{C}_\lambda[\![t]\!] \quad \mathcal{C}_\lambda[\![t_1\ t_2]\!]k = \mathcal{C}_\lambda[\![t_1]\!]\lambda f.\mathcal{C}_\lambda[\![t_2]\!]\lambda s.f\ s\ k$$

Variables and functions are both values, so during evaluation they are just passed to the current continuation. The only non-value case, where actual computation occurs, is in the function application step. First, the function is evaluated, and its value is bound to f in the top continuation. Second, the argument is evaluated and its value is bound to s in the next continuation. Finally, with values for both terms, the function value is applied to the argument value, and the computation continues with the original continuation k.

In the output of this transformation, terms are maps from continuations, k, to final answers. Continuations, then, are maps from values to final answers. This means that the CPS translation of a term does not execute by itself, it must be given some initial continuation in order to begin the process of evaluation. Following the sequent calculus tradition, we add the counterpart of this initial continuation to the syntax, which explicitly marks the *top-level*, or final return point of the whole program. We name this continuation $*$ and specify that running a term consists of coupling that term with $*$, written as $[*]t$, which we call a command. Operationally, the command $[*]t$ is interpreted as evaluating the term t in the empty context. We extend the syntax of our call-by-value calculus with two new syntactic categories:

$$c \in Command ::= [q]t \qquad q \in CoTerm ::= * \qquad t \in Term ::= V \mid t_1\ t_2$$

We also extend our previous CPS transform $\mathcal{C}_\lambda$ with clauses for commands and the constant $*$.

$$\mathcal{C}_\lambda[\![[q]t]\!] = \mathcal{C}_\lambda[\![t]\!]\ \mathcal{C}_\lambda[\![q]\!] \qquad\qquad \mathcal{C}_\lambda[\![*]\!] = \lambda x.x$$

The interpretation of the command $[q]t$ is to evaluate the term t in the context q, which means to pass the continuation represented by q to the term. The initial continuation $*$ just returns the value it is given without modifying it.

3 Lambda Calculus with Control: Parigot's $\lambda\mu$

Felleisen [9] extended the call-by-value lambda calculus with continuation abstraction. This allows a term to store its evaluation context as a special function and to reinstall this context by invoking that function. The function representing a continuation never returns to the call site. Here, we instead follow Parigot's approach [18] because it provides a reduction theory which more accurately simulates the operational semantics [1]. In Parigot's $\lambda\mu$, continuations are not functions. Similarly to the the top-level, continuations belong to a separate syntactic category of co-terms. Intuitively, terms are producers of values, whereas continuations are consumers of values. The invocation of a continuation is a command. The syntax of $\lambda\mu$ extends the class of terms and co-terms as follows:

$$c \in Command ::= [q]t \quad t \in Term ::= V \mid t_1\ t_2 \mid \mu\alpha.c \quad q \in CoTerm ::= \alpha \mid *$$

We define the CPS semantics of $\lambda\mu$ by extending $\mathcal{C}_\lambda$ for the new syntax:

$$\mathcal{C}_{\lambda\mu}[\![\mu\alpha.c]\!]k = (\lambda\alpha.\mathcal{C}_{\lambda\mu}[\![c]\!])\ k \qquad\qquad \mathcal{C}_{\lambda\mu}[\![\alpha]\!] = \alpha$$

The reduction semantics is then given by the following reduction rules:

$$(\lambda x.t)\ V \to t\{V/x\} \qquad E_1[\mu\alpha.c] \to c\{[\alpha](E_1[t])/[\alpha]t\} \qquad [q]\mu\alpha.c \to c\{q/\alpha\}$$

Where the one-step evaluation context E_1 is defined as: $E_1 ::= \Box\ t \mid V\ \Box$. The term $\mu\alpha.c$ propagates its evaluation context piece-by-piece to each invocation of α in c, until it reaches the top of its surrounding command. The rule makes use of a new notion of substitution, called *structural substitution*; $c\{[\alpha](E_1[t])/[\alpha]t\}$ should be read as: substitute each occurrence of $[\alpha]t$ in command c with $[\alpha](E_1[t])$. When iterated, these two rules perform the big-step capturing reduction that substitutes the entire evaluation context up to the top of the command. The operational semantics of $\lambda\mu$ is:

$$[*]E[(\lambda x.t)\ V] \mapsto [*]E[t\{V/x\}] \qquad\qquad [*]E[\mu\alpha.c] \mapsto c\{[*]E[t]/[\alpha]t\}$$

Where the evaluation context is: $E ::= \Box \mid E\ t \mid V\ E$. The operational semantics is sound and complete with respect to the CPS transform: $\mathcal{C}_{\lambda\mu}[\![[*]t]\!] \mapsto\!\!\!\!\to V$ iff $[*]t \mapsto\!\!\!\!\to [*]V$.

Expressiveness. Parigot's $\lambda\mu$ equipped with the top-level constant $*$ gives us the ability to express the $\mathsf{call/cc}$ ($\mathcal{K}$) and the abort ($\mathcal{A}$) control operators. One can also express Felleisen's $\mathcal{C}$ operator, which is definable in terms of $\mathsf{call/cc}$ and abort.

$$\begin{array}{c} \mathcal{K} = \lambda h.\mu\alpha.[\alpha]h\ (\lambda x.\mu\_.[\alpha]x) \qquad \mathcal{A}\ t = \mu\_.[*]t \\ \mathcal{C} = \lambda h.\mathcal{K}\ (\lambda k.\mathcal{A}\ (h\ k)) = \lambda h.\mu\alpha.[*]h\ (\lambda x.\mu\_.[\alpha]x) \end{array} \tag{1}$$

4 Delimited Control: $\lambda\mu\widehat{\mathsf{tp}}$

Delimiting control means temporarily re-defining the top-level in a program, limiting the extent to which the evaluation context may be captured. Examples of delimited control are the shift ($\mathcal{S}$) and reset ($\#$) operators given in the seminal paper of Danvy and Filinski [5]. Felleisen [10,9] also extended his control theory with a reset operator which he calls prompt. The prompt operator is shown to be necessary in providing a fully abstract model of λ-calculus [20].

In [2], it is shown that delimited control can be explained by replacing the top-level constant $*$ with the rebindable dynamic continuation variable $\widehat{\mathsf{tp}}$. The syntax of $\lambda\mu\widehat{\mathsf{tp}}$ is:

$$c \in Comm. ::= [q]t \quad t \in Term ::= V \mid t_1\ t_2 \mid \mu q.c \quad q \in CoTerm ::= \alpha \mid \widehat{\mathsf{tp}} \tag{2}$$

The dynamic nature of $\widehat{\mathsf{tp}}$ is due to the fact that in a function like $\lambda x.\mu\_.[\widehat{\mathsf{tp}}]x$, the binding of $\widehat{\mathsf{tp}}$ is taken from the environment active at the call site and not in the environment active when the function is defined. This dynamic nature is captured by adding the following reduction rule to the reduction theory of $\lambda\mu$:

$$\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]V \to V$$

4.1 Continuation Passing Style ($\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$)

We extend the $\mathcal{C}_{\lambda\mu}$ transform to give $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}$, the CPS transform for $\lambda\mu\widehat{\mathsf{tp}}$.

$$\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\mu\widehat{\mathsf{tp}}.c]\!]k = k\ (\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!]) \qquad \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!] = \lambda x.x$$

Here, $\widehat{\mathsf{tp}}$ takes the place of the old constant $*$. However, now we also have a binding form for $\widehat{\mathsf{tp}}$. When $\widehat{\mathsf{tp}}$ is bound over a command, the current continuation is set aside and that command is run to completion. Then, when the command has produced an answer value, that value is fed to the original continuation and that context is restored. Unfortunately, the above translation of $\mu\widehat{\mathsf{tp}}.c$ is not in CPS form, since the term $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!]$ is an application instead of a value. One can remedy the situation by taking the output from $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}$ and running it through the CPS transform $\mathcal{C}_\lambda$ [5]. The composition of the two CPS transforms gives us $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$, a double CPS transform. There is no change to the clauses inherited from $\mathcal{C}_{\lambda\mu}$ since they were already in full CPS form. The only difference is in the translation of $\widehat{\mathsf{tp}}$:

$$\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\mu\widehat{\mathsf{tp}}.c]\!]k = \lambda\gamma.\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!]\lambda x.k\ x\ \gamma \qquad \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!] = \lambda x.\lambda\gamma.\gamma\ x$$

The CPS transform of a term is now a function requiring both a continuation k and a meta-continuation γ. In addition, continuations now take both a value and a meta-continuation as parameters. Here, the initial value for the meta-continuation is γ_ι which is initialized to $\lambda x.x$.

Notice that we are now in the same situation as we were with the pure λ-calculus. The CPS translation of both terms and commands take an extra argument, but this fact is not reflected in the syntax of $\lambda\mu\widehat{\mathsf{tp}}$. To reconcile the difference between the CPS transform and the source language, we extend the syntax of $\lambda\mu\widehat{\mathsf{tp}}$ in the same way we extended the pure λ-calculus. We add a second-order command, or meta-command, which explicitly names the meta-continuation of the underlying first-order command. Since we can only mark the initial meta-continuation of a command, we add the constant $\circledast$, which is the meta-top-level of the program. Thus, we extend the syntax of $\lambda\mu\widehat{\mathsf{tp}}$ given in (2) with meta-commands:

$$q^2 \in CoTerm^2 ::= \circledast \qquad c^2 \in Command^2 ::= [q^2]c$$

The double CPS translation of meta-commands and the meta-top-level $\circledast$ follow the same pattern as commands and the top-level in the pure λ-calculus:

$$\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[q^2]c]\!] = \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!]\ \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![q^2]\!] \qquad \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\circledast]\!] = \lambda x.x$$

The standard way to evaluate the CPS form of term t in this system is to provide the initial continuation $\lambda x.\lambda\gamma.\gamma\ x$ and the initial meta-continuation $\lambda x.x$, which translates to evaluating the meta-command $[\circledast][\widehat{\mathsf{tp}}]t$. If the meta-command is reduced to $[\circledast][\widehat{\mathsf{tp}}]V$, then the value V is the final answer.

Expressiveness. The rebindable top-level is the additional power that allows us to encode shift ($\mathcal{S}$) and reset ($\#$) in $\lambda\mu\widehat{\mathsf{tp}}$:

$$\#t = \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]t \qquad \mathcal{S} = \lambda h.\mu\alpha.[\widehat{\mathsf{tp}}]h\ (\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x)$$

The above encoding resembles Filinski's encoding [11] of $\mathcal{S}$ and $\#$ in terms of Felleisen's $\mathcal{C}$ and $\#$ operators. One can also encode a slightly different abort operator, $\mathcal{A}^{\widehat{\mathsf{tp}}}$, which aborts up to the nearest binding of $\widehat{\mathsf{tp}}$. This operator is expressible in terms of shift alone.

$$\mathcal{A}^{\widehat{\mathsf{tp}}}\ t = \mathcal{S}\ \lambda\_.t = \mu\_.[\widehat{\mathsf{tp}}]t \tag{3}$$

The behavior of this operator is different from the original abort, in that it does not exit the program completely, but only removes the context up to the nearest binding of $\widehat{\mathsf{tp}}$.

Unbound $\widehat{\mathsf{tp}}$. It is important to note that in the above definition of $\lambda\mu\widehat{\mathsf{tp}}$, the $\widehat{\mathsf{tp}}$ variable is always bound throughout the entire execution of the program. In a sense, the meta-continuation, which is responsible for giving the current binding for $\widehat{\mathsf{tp}}$, already comes with $\widehat{\mathsf{tp}}$ bound to the *true* top-level of the program. Next, we analyze the impact of this choice.

Example 1. The following example shows the successful evaluation of a meta-command with an unbound use of $\widehat{\mathsf{tp}}$, which is equivalent to the shift expression $\mathcal{S}\lambda\_.9 = \mu\alpha.[\widehat{\mathsf{tp}}](\lambda\_.9)\ (\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x)$.

$$[\circledast][\widehat{\mathsf{tp}}]\mu\alpha.[\widehat{\mathsf{tp}}]((\lambda\_.9)\ (\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x)) \twoheadrightarrow [\circledast][\widehat{\mathsf{tp}}]9$$
$$\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\circledast][\widehat{\mathsf{tp}}]9]\!] = \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!]\ 9\ \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\circledast]\!] = \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\circledast]\!]\ 9 = 9$$

Alternative Initial Conditions. But what if we want to begin evaluation with $\widehat{\mathsf{tp}}$ initially unbound? To do this, we will need to add the true top-level of the program, $*$, back to our grammar along with a different meta-top-level in which $\widehat{\mathsf{tp}}$ is considered unbound. Our syntax of $\lambda\mu\widehat{\mathsf{tp}}$ becomes:

$$\begin{aligned} c^2 \in \mathit{Command}^2 &::= [q^2]c & t \in \mathit{Term} &::= V \mid t_1\ t_2 \mid \mu\widetilde{\alpha}.c \\ c \in \mathit{Command} &::= [q]t & q \in \mathit{CoTerm} &::= \widetilde{\alpha} \mid * \\ q^2 \in \mathit{CoTerm}^2 &::= \bullet & \widetilde{\alpha} \in \mathit{CoVar} &::= \alpha \mid \widehat{\mathsf{tp}} \end{aligned}$$

Note that we now have both notions of abort as defined in (1) and (3). $\mathcal{A}^{\widehat{\mathsf{tp}}}$ removes the context up to the nearest binding of $\widehat{\mathsf{tp}}$, whereas $\mathcal{A}$ removes the context of the entire rest of the program.

The $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$ transform is extended with clauses for the new top-level and meta-top-level. The meaning of the constant $*$ is easy to define, but $\bullet$ is more tricky.

$$\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![*]\!] = \lambda x.\lambda\gamma.x \qquad\qquad \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\bullet]\!] = \gamma_0\ \textbf{where}\ \gamma_0 \text{ free}$$

When $*$ is invoked with a value, the program immediately exits with that value as a final answer. The meta-continuation is thrown away because the current binding of $\widehat{\mathsf{tp}}$ is not needed. If the $\widehat{\mathsf{tp}}$ continuation is given a value without being bound, then the program gets stuck; since $\widehat{\mathsf{tp}}$ was not defined there is not enough information to continue. We need to map this stuck state down to the target language of $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$: the pure λ-calculus. A natural way to do this is to make $\bullet$, the meta-top-level in which $\widehat{\mathsf{tp}}$ is unbound, a free variable. Then, invoking an unbound $\widehat{\mathsf{tp}}$ with a value is translated to a stuck term.

The reduction semantics of $\lambda\mu\widehat{\mathsf{tp}}$ is extended with one more rule to reduce an invocation of $*$ under a binding for $\widehat{\mathsf{tp}}$.

$$\mu\widehat{\mathsf{tp}}.[*]V \to \mu\_.[*]V$$

The meaning of $[*]V$ is to throw away the bindings of $\widehat{\mathsf{tp}}$ in γ and return with the value V as the final answer. Therefore, we can throw away an adjacent binding of $\widehat{\mathsf{tp}}$ by turning it into an abort.

Example 2. Let's revisit the previous example using $*$ and $\bullet$ to initialize execution instead of $\widehat{\mathsf{tp}}$ and $\circledast$.

$$[\bullet][*]\mu\alpha.[\widehat{\mathsf{tp}}]((\lambda\_.9)\ (\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x)) \twoheadrightarrow [\bullet][\widehat{\mathsf{tp}}]9$$
$$\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\bullet][\widehat{\mathsf{tp}}]9]\!] = \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!]\ 9\ \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\bullet]\!] = \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\bullet]\!]\ 9 = \gamma_0\ 9$$

Since $\widehat{\mathsf{tp}}$ was not initialized we get an error, represented by the stuck term $\gamma_0\ 9$.

The reduction rules of $\lambda\mu\widehat{\mathsf{tp}}$ are sound and complete with respect to $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}$.

Theorem 1. *If* $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\bullet][*]t]\!] \twoheadrightarrow V$ *then* $[\bullet][*]t \twoheadrightarrow [\bullet][*]V$.
If $M \to M'$ *then* $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![M]\!] = \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![M']\!]$.

Where the meta-syntactic variable M ranges over terms, commands, and meta-commands. Here and throughout the paper, equality between terms in the λ-calculus are up to $\beta\eta$ reduction.

Even though we replaced $\circledast$ with $\bullet$ in our language, we haven't actually lost anything. We can regain the original initial conditions by providing a binding for $\widehat{\mathsf{tp}}$ at the top of the program.

Theorem 2. $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\bullet][*]\mu\widehat{\mathsf{tp}}.c]\!] = \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\circledast]c]\!]$

5 Intermediate Languages of Dynamic Binding: $\lambda\widehat{\mathsf{tp}}$, $\lambda\widehat{\mathsf{tp}}^b$

Ariola *et al.* [2] showed how the CPS of $\lambda\mu\widehat{\mathsf{tp}}$ can be factored into a state-passing transformation to $\lambda\mu$ extended with subtraction combined with a translation to λ-calculus with pairs. In order to better understand the dynamic nature of the prompt binding, we investigate an alternative decomposition. We start by translating away the control effects from $\lambda\mu\widehat{\mathsf{tp}}$ ($\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}$), leaving behind the dynamic binding of $\widehat{\mathsf{tp}}$. We then translate away the dynamic binding by first adopting a typical environment passing translation ($\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}$). This however leads to an incorrect interpretation of the dynamic nature of $\widehat{\mathsf{tp}}$. We thus propose another way of translating the dynamic binding that models the behavior of the prompt ($\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^b}$).

5.1 Translating Control ($\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}$)

We start with a CPS transform from $\lambda\mu\widehat{\mathsf{tp}}$ to an intermediate language with one dynamic variable, $\lambda\widehat{\mathsf{tp}}$, with the following syntax:

$$c \in Closure ::= [e]t \qquad t \in Term ::= V \mid t_1\ t_2 \mid \widehat{\mathsf{tp}}$$
$$\widetilde{x} \in Var ::= x \mid \widehat{\mathsf{tp}} \qquad V \in Value ::= x \mid \lambda\widetilde{x}.t$$

Where e is the empty *Environment* $\bullet$. The $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}$ transform defines the call-by-value application and the context capturing behavior of $\mu\widetilde{\alpha}.c$ while using the dynamic variable in $\lambda\widehat{\mathsf{tp}}$ to manage the binding of $\widehat{\mathsf{tp}}$.

$$\begin{aligned}
\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![[q^2]c]\!] &= [\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![q^2]\!]]\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!] & \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![x]\!]k &= k\ x\\
\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![[q]t]\!] &= \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t]\!]\ \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![q]\!] & \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\lambda x.t]\!]k &= k\ \lambda x.\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t]\!]\\
\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\alpha]\!] &= \alpha & \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t_1\ t_2]\!]k &= \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t_1]\!]\lambda f.\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t_2]\!]\lambda s.f\ s\ k\\
\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!] &= \lambda x.\widehat{\mathsf{tp}}\ x & \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\mu\widetilde{\alpha}.c]\!]k &= (\lambda\widetilde{\alpha}.\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!])\ k\\
\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![*]\!] &= \lambda x.x & \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\bullet]\!] &= \bullet
\end{aligned}$$

Note that $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!]$ is η-expanded. Otherwise, in the translation of $[\widehat{\mathsf{tp}}]\mu\alpha.c$ one would obtain $(\lambda\alpha.\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!])\ \widehat{\mathsf{tp}}$. Since $\widehat{\mathsf{tp}}$ is not a value, the dynamic binding would be looked up when α is defined, instead of when it is called. To better understand the reason consider the following example.

Example 3. In $[*]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mu\alpha.[\alpha]((\mu\widehat{\mathsf{tp}}.[\alpha]I)\ x)$, notice that α is invoked with a value under a rebinding of $\widehat{\mathsf{tp}}$. The renaming of $\widehat{\mathsf{tp}}$ for α is captured by the more recent binding, as shown by the reduction:

$$[*]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mu\alpha.[\alpha]((\mu\widehat{\mathsf{tp}}.[\alpha]I)\ x) \to [*]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]((\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]I)\ V)$$

If we instead adopt the transform $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!] = \widehat{\mathsf{tp}}$ then we would have to bind α to the current value of $\widehat{\mathsf{tp}}$, which is $*$.

5.2 Translating Dynamic Binding ($\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}$)

For a first attempt at defining the dynamic binding of $\widehat{\mathsf{tp}}$, we try a simple environment-passing style transform, $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}$, where the environment is just the value currently bound to $\widehat{\mathsf{tp}}$. In the case that $\widehat{\mathsf{tp}}$ isn't bound, as in the initial environment $\bullet$, we use the free variable γ_0. That is, we have $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![\bullet]\!] = \gamma_0$. The rest of the transform is:

$$\begin{aligned}
\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![[e]t]\!] &= \mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![t]\!]\ \mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![e]\!] & \mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![\lambda x.t]\!]\gamma &= \lambda x.\lambda\gamma'.\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![t]\!]\gamma'\\
\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![x]\!]\gamma &= x & \mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![\lambda\widehat{\mathsf{tp}}.t]\!]\gamma &= \lambda v.\lambda\gamma'.\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![t]\!]v\\
\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!]\gamma &= \gamma & \mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![t_1\ t_2]\!]\gamma &= (\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![t_1]\!]\gamma)\ (\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![t_2]\!]\gamma)\ \gamma
\end{aligned}$$

This transform is equivalent to a simplified version of Moreau's calculus of dynamic binding [17] with only one dynamic variable.

Unfortunately, this definition of dynamic binding does not properly capture the meaning of the rebindable top-level since it creates vicious cycles, as shown in the reduction of $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}} \circ \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]x]\!]\gamma$:

$$(\lambda v.\lambda\gamma'.v\ x\ v)\ (\lambda y.\lambda\gamma'.\gamma'\ y\ \gamma')\ \gamma \to (\lambda y.\lambda\gamma'.\gamma'\ y\ \gamma')\ x\ (\lambda y.\lambda\gamma'.\gamma'\ y\ \gamma') \to \ldots$$

This does not match the reductions of $\lambda\mu\widehat{\mathsf{tp}}$, since one has: $[\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]x \to [\widehat{\mathsf{tp}}]x$. In Moreau's [17] framework, this corresponds to the reduction:

$$\mathbf{dlet}\,\widehat{\mathsf{tp}} = (\lambda y.\widehat{\mathsf{tp}}\ y)\,\mathbf{in}\,\widehat{\mathsf{tp}}\ x \twoheadrightarrow \mathbf{dlet}\,\widehat{\mathsf{tp}} = (\lambda y.\widehat{\mathsf{tp}}\ y)\,\mathbf{in}(\lambda y.\widehat{\mathsf{tp}}\ y)\ x \twoheadrightarrow \ldots$$

Remark 1. One can understand the dynamic abstraction $\lambda\widehat{\mathsf{tp}}.t$ in terms of a static abstraction and dynamic let, as $\lambda v.\, \mathbf{dlet}\, \widehat{\mathsf{tp}} = v\, \mathbf{in}\, t$, where the transform of **dlet** is $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![\mathbf{dlet}\, \widehat{\mathsf{tp}} = v\, \mathbf{in}\, t]\!]\gamma = \mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![t]\!]v$.

5.3 Backtracking the Environment ($\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^b}$)

We see vicious cycles arise because dynamic binding allows for self-reference. In order to evaluate the application $\widehat{\mathsf{tp}}\ V$, we (1) lookup the value f most recently bound to $\widehat{\mathsf{tp}}$, and (2) evaluate $f\ V$ in the current environment where f is still bound. The root of our problem is in step (2). Instead, we want to evaluate $f\ V$ in a different environment where that same f isn't bound. In particular, we want to backtrack to the environment that was active just before f was bound to $\widehat{\mathsf{tp}}$. To do this, we restrict the grammar of $\lambda\widehat{\mathsf{tp}}$ so that $\widehat{\mathsf{tp}}$ can only be used as an immediate application, giving us $\lambda\widehat{\mathsf{tp}}^b$.

$$t \in Term ::= V \mid t_1\ t_2 \mid \widehat{\mathsf{tp}}\ t$$

We then modify $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}$ to match the restricted grammar. In particular, we change dynamic binding and application of $\widehat{\mathsf{tp}}$ to backtrack to a previous environment.

$$\begin{aligned}
\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^b}[\![\lambda\widehat{\mathsf{tp}}.t]\!]\gamma &= \lambda v.\lambda\gamma'.\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^b}[\![t]\!]\gamma'[\widehat{\mathsf{tp}} \mapsto v] &\qquad \gamma[\widehat{\mathsf{tp}} \mapsto v] &= \lambda x.v\ x\ \gamma \\
\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^b}[\![\widehat{\mathsf{tp}}\ t]\!]\gamma &= \gamma(\widehat{\mathsf{tp}})\ (\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^b}[\![t]\!]\gamma) & \gamma(\widehat{\mathsf{tp}}) &= \gamma
\end{aligned}$$

When we bind a value v to $\widehat{\mathsf{tp}}$, we wrap v in a function that will return to the previous dynamic environment when applied. Since values bound to $\widehat{\mathsf{tp}}$ are equipped with their own environment, we do not need to pass the current dynamic environment to the application $\widehat{\mathsf{tp}}\ V$. Compare the new translation of $\widehat{\mathsf{tp}}\ V$ with the original one: $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}\ V]\!]\gamma = \gamma\ V\ \gamma$. Notice that the restriction on how $\widehat{\mathsf{tp}}$ is used allows us to eliminate the self-application of the environment γ. With the new backtracking definition of dynamic binding, we no longer create the same cycle as before in the reduction of $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^b} \circ \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]x]\!]\gamma$:

$$(\lambda v.\lambda\gamma'.v\ x\ \gamma')\ (\lambda y.\lambda\gamma'.\gamma'\ y)\ \gamma \rightarrow (\lambda y.\lambda\gamma'.\gamma'\ y)\ x\ \gamma \rightarrow \gamma\ x = [\![\widehat{\mathsf{tp}}\ x]\!]\gamma$$

When we compose the two phases together, we get the derived translation $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^b} \circ \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}$, which is exactly the same as our original translation $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$.

Theorem 3. $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^b} \circ \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}} = \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$

Remark 2. Note that the definition of $*$ in $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$ is exactly the environment-passing style translation of the initial continuation $\lambda x.x$. The backtracking behavior we present here is also necessary to express exceptions with dynamic variables. A similar encoding was given by Moreau[17] using an abort operator to reinstall the right environment.

6 Control with Multiple Prompts: $\lambda\widehat{\mu}$ via $\widehat{\lambda}^b$

We want to extend $\lambda\mu\widehat{\mathsf{tp}}$ with a multiple prompts so that binding prompt $\widehat{\alpha}$ does not interfere with prompt $\widehat{\beta}$ and vice versa. This is different from the nested definition of resets in the CPS hierarchy [7]. Unfortunately, this means that we cannot use the iterated layered CPS approach to define our prompts. However, now that we have factored the transform for $\lambda\mu\widehat{\mathsf{tp}}$ into two passes that flow through an intermediate language with dynamic binding, it is easy to extend the calculus to have multiple prompts by simply using an intermediate language with multiple dynamic variables, $\widehat{\lambda}^b$, whose syntax is:

$$\begin{aligned} c \in Closure &::= [e]t & t \in Term &::= V \mid t_1\ t_2 \mid \widehat{x}\ t \\ \widetilde{x} \in Var &::= x \mid \widehat{x} & V \in Value &::= x \mid \lambda\widetilde{x}.t \end{aligned}$$

Where e is the empty environment $\bullet$. The definition of $\widehat{\lambda}^b$ uses the same environment-passing style translation as $\lambda\widehat{\mathsf{tp}}^b$. The only thing that needs to change from $\lambda\widehat{\mathsf{tp}}^b$ to $\widehat{\lambda}^b$ is dynamic binding and lookup. Now that there is more than one variable, we may have to search through the environment for the variable that we want.

$$\gamma(\widehat{x}) = \gamma\ \ulcorner\widehat{x}\urcorner \qquad \gamma[\widehat{x} \mapsto v] = \lambda p.\, \mathbf{if}\, p \equiv \ulcorner\widehat{x}\urcorner\, \mathbf{then}\, \lambda x.v\ x\ \gamma\, \mathbf{else}\, \gamma\ p$$

Here the quotation brackets, $\ulcorner\cdot\urcorner$, reify the dynamic variables into terms in the target language. These terms must all be distinct and have decidable equality.

The language of control with multiple prompts, $\lambda\widehat{\mu}$, is a simple extension of $\lambda\mu\widehat{\mathsf{tp}}$ with multiple dynamic top-level binders.

$$\begin{aligned} c^2 \in Command^2 &::= [q^2]c & t \in Term &::= V \mid t_1\ t_2 \mid \mu\widetilde{\alpha}.c \\ c \in Command &::= [q]t & q \in CoTerm &::= \widetilde{\alpha} \mid * \\ q^2 \in CoTerm^2 &::= \bullet & \widetilde{\alpha} \in CoVar &::= \alpha \mid \widehat{\alpha} \end{aligned}$$

The semantics of $\lambda\widehat{\mu}$ is just the composed transform $\mathcal{D}_{\widehat{\lambda}^b} \circ \mathcal{C}_{\lambda\widehat{\mu}}$, exactly as in Section 5.3 except that multiple dynamic variables are used by $\mathcal{C}_{\lambda\widehat{\mu}}$, with one unique variable for each different prompt. The reduction rules for multiple prompts are a generalization of the reduction rules for single prompt $\widehat{\mathsf{tp}}$.

$$\mu\widehat{\alpha}.[\widehat{\alpha}]V \to V \qquad \mu\widehat{\alpha}.[\widehat{\beta}]V \to \mu\_.[\widehat{\beta}]V \qquad \mu\widehat{\alpha}.[*]V \to \mu\_.[*]V$$

Where $\widehat{\alpha} \neq \widehat{\beta}$. Just like how invocation of $*$ throws away the dynamic environment, invocation of the prompt $\widehat{\beta}$ will throw away portions of its dynamic environment until the correct binding is found. Then the usual η-reduction of prompts is available to resolve the invocation of the prompt.

To define the operational semantics for $\lambda\widehat{\mu}$, we first give the evaluation contexts for $\lambda\widehat{\mu}$, which can be derived from a defunctionalization [19,4] of the continuation and environment used in the $\mathcal{D}_{\widehat{\lambda}^b} \circ \mathcal{C}_{\lambda\widehat{\mu}}$ transform.

$$E ::= \square \mid E\ t \mid V\ E \quad F ::= [q]E \quad E^2 ::= \square \mid F[\mu\widehat{\alpha}.E^2] \quad F^2 ::= [q^2]E^2$$

The context E is just the standard call-by-value evaluation context for the pure λ-calculus. The meta-context E^2 drills down through any number of dynamic bindings for continuation variables. Both F and F^2 are convenient shorthand for a (meta-)context embedded in a (meta-)command, and correspond exactly with the continuation and meta-continuation in the CPS transform for $\lambda\widehat{\mu}$.

The operational rules are derived from the fine-grained reduction rules using the call-by-value contexts for $\lambda\widehat{\mu}$ to restrict where they may apply.

$$\begin{array}{ll} F^2[F[(\lambda x.t)\ V]] \mapsto F^2[F[t\{V/x\}]] & F^2[F[\mu\alpha.c]] \mapsto F^2[c\{F[t]/[\alpha]t\}] \\ F^2[F[\mu\widehat{\alpha}.E^2_{\widehat{\alpha}}[[\widehat{\alpha}]V]]] \mapsto F^2[F[V]] & [q^2]E^2[[*]V] \mapsto [q^2][*]V \end{array}$$

Where $E^2_{\widehat{\alpha}}$ does not contain a binding for $\widehat{\alpha}$. The reduction rules for $\lambda\widehat{\mu}$ are sound with respect to the transform $\mathcal{D}_{\widehat{\lambda}^b} \circ \mathcal{C}_{\lambda\widehat{\mu}}$, and the operational semantics is complete with respect to the transform. Also, since the operational semantics are just a coarse, limited form of the reduction rules, it follows that the reductions are complete with respect to the operational semantics and that the operational semantics is sound with respect to the transform.

Theorem 4. *If $M \to M'$ then $\mathcal{D}_{\widehat{\lambda}^b} \circ \mathcal{C}_{\lambda\widehat{\mu}}[\![M]\!] = \mathcal{D}_{\widehat{\lambda}^b} \circ \mathcal{C}_{\lambda\widehat{\mu}}[\![M']\!]$. If $\mathcal{D}_{\widehat{\lambda}^b} \circ \mathcal{C}_{\lambda\widehat{\mu}}[\![[\bullet][*]t]\!] \mapsto\!\!\!\!\to V$ then $[\bullet][*]t \mapsto\!\!\!\!\to [\bullet][*]V$. If $c^2 \mapsto c'^2$ then $c^2 \twoheadrightarrow c'^2$.*

Expressiveness. With multiple prompts, we get the ability to set multiple points in the program that we can abort to at will, giving us the multi-prompt reset ($\#^{\widehat{\alpha}}$) and abort ($\mathcal{A}^{\widehat{\alpha}}$) operators.

$$\#^{\widehat{\alpha}}t = \mu\widehat{\alpha}.[\widehat{\alpha}]t \qquad \mathcal{A}^{\widehat{\alpha}}t = \mu\_.[\widehat{\alpha}]t$$

We can also encode exception handling with multiple independent exceptions.

$$\textbf{raise}\ e\ t = (\lambda x.\mathcal{A}^{\widehat{e}}\,\mathsf{Exn}\,x)\ t$$
$$t\,\textbf{handle}\,e\ x \Rightarrow u = \textbf{case}\ \#^{\widehat{e}}\,\mathsf{OK}\,t\,\textbf{of}\ \mathsf{OK}\,x \Rightarrow x \mid \mathsf{Exn}\,x \Rightarrow u$$

The expression **raise** $e\ t$ evaluates t and then aborts to the dynamically nearest handler for e with an exception. The handling expression t **handle** $e\ x \Rightarrow u$ attempts to evaluate t. If t successfully results in a value (represented as $\mathsf{OK}\,v$), then value v is returned. Otherwise, if an exception for e is raised (represented as $\mathsf{Exn}\,v$), then u is evaluated with the raised value v bound to x.

7 Delimited Control with Transparent Prompts: $\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}$

We now take a break from $\lambda\widehat{\mu}$ and multiple prompts, and return to $\lambda\mu\widehat{\mathsf{tp}}$ in order to examine an alternate extension. Another important delimited control operator to consider is shift$_0$ ($\mathcal{S}_0$) [16]. The difference between shift and shift$_0$ is that when shift captures its immediate context, it leaves the nearest delimiting reset in place, whereas shift$_0$ removes the nearest reset after capturing its context. As discussed previously in Section 4, shift and reset have encodings in $\lambda\mu\widehat{\mathsf{tp}}$. However, to capture the additional behavior of shift_0 we need to extend $\lambda\mu\widehat{\mathsf{tp}}$ with the ability to render the binding of a prompt transparent, making it immediately disappear and letting underlying terms see through to their surrounding context. The new command $\uparrow^{\widehat{\mathsf{tp}}} t$ represents lifting the unevaluated term t through the most recent binding of $\widehat{\mathsf{tp}}$ and embedding the term in that context. The syntax of $\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}$ is:

$$
\begin{aligned}
c^2 \in \mathit{Command}^2 &::= [q^2]c & t \in \mathit{Term} &::= V \mid t_1\ t_2 \mid \mu\widetilde{\alpha}.c \\
c \in \mathit{Command} &::= [q]t \mid \ \uparrow^{\widehat{\mathsf{tp}}} t & q \in \mathit{CoTerm} &::= \widetilde{\alpha} \mid * \\
q^2 \in \mathit{CoTerm}^2 &::= \bullet & \widetilde{\alpha} \in \mathit{CoVar} &::= \alpha \mid \widehat{\mathsf{tp}}
\end{aligned}
$$

We define a CPS for $\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}$ in the style of Materzok and Biernacki's [16] definition of shift_0. This is an extension of the basic $\mathcal{C}_{\lambda\mu}$ transform.

$$
\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![\mu\widehat{\mathsf{tp}}.c]\!]k = \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![c]\!]\ k \quad \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![\uparrow^{\widehat{\mathsf{tp}}} t]\!] = \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![t]\!] \quad \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![\widehat{\mathsf{tp}}]\!] = \lambda x.\lambda k.k\ x
$$

Note that the translation of $\mu\widehat{\mathsf{tp}}.c$ in $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}$ is different from the one in $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}}$. Rather than always running the command to completion, and then passing the result to the bound continuation k, we pass k as an extra argument to the command. A continuation bound to $\widehat{\mathsf{tp}}$ is set aside and carried along in the command as an extra argument. Then, in the case of $[\widehat{\mathsf{tp}}]V$, the list of extra arguments is accessed and V is returned to the most recent one. On the other hand, in the case of $\uparrow^{\widehat{\mathsf{tp}}} t$, the continuation most recently bound to $\widehat{\mathsf{tp}}$ is accessed and used to evaluate t.

For the purpose of comparison with $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$, we run the output of $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}$ through the CPS transform $\mathcal{C}_\lambda$, which gives us the double CPS transform $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}$.

$$
\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![\mu\widehat{\mathsf{tp}}.c]\!]k = \lambda\gamma.\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![c]\!]\lambda t.t\ k\ \gamma
$$

$$
\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![\uparrow^{\widehat{\mathsf{tp}}} t]\!]\gamma = \gamma\ \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![t]\!] \qquad \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![\widehat{\mathsf{tp}}]\!] = \lambda x.\lambda\gamma.\gamma\ \lambda k.k\ x
$$

The small difference in the binding of $\widehat{\mathsf{tp}}$ becomes immediately apparent in the type of the meta-continuation. In $\lambda\mu\widehat{\mathsf{tp}}$, the meta-continuation takes values to final answers. In $\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}$, on the other hand, the meta-continuation takes *terms* to final answers. The more general type allows the translation of $\uparrow^{\widehat{\mathsf{tp}}} t$ to pass t unevaluated to the meta-continuation. The translation of $[\widehat{\mathsf{tp}}]V$ now has to compensate for this extra generality. When the $\widehat{\mathsf{tp}}$ continuation is given a value x and meta-continuation γ, it wraps that value up in the trivial term that immediately returns x, and passes the new term to γ.

We can also define the transformation in terms of $\lambda\widehat{\mathsf{tp}}^{b}$, as in Section 5. Extending the meta-continuation becomes a binding to the dynamic variable $\widehat{\mathsf{tp}}$, and application of the meta-continuation becomes application of $\widehat{\mathsf{tp}}$.

$$
\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![\mu\widehat{\mathsf{tp}}.c]\!]k = (\lambda\widehat{\mathsf{tp}}.\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![c]\!])\ (\lambda t.t\ k)
$$

$$
\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![\uparrow^{\widehat{\mathsf{tp}}} t]\!] = \widehat{\mathsf{tp}}\ \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![t]\!] \qquad \mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![\widehat{\mathsf{tp}}]\!] = \lambda x.\widehat{\mathsf{tp}}\ \lambda k.k\ x
$$

Notice the difference between the two uses of $\widehat{\mathsf{tp}}$. In $[\widehat{\mathsf{tp}}]t$, resolution of the $\widehat{\mathsf{tp}}$ variable is delayed in a function that is passed to the term t, which may be captured by the time the continuation is used. In contrast, $\uparrow^{\widehat{\mathsf{tp}}} t$, directly applies $\widehat{\mathsf{tp}}$ to a term, immediately resolving the dynamic binding in the current environment.

Reduction of the new command is similar to $[\widehat{\mathsf{tp}}]t$, but with different priorities between the continuation and the term. In $\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]t$, $\widehat{\mathsf{tp}}$ is η-reduced only when t is a value. The opposite occurs with $\mu\widehat{\mathsf{tp}}.\uparrow^{\widehat{\mathsf{tp}}} t$, where $\widehat{\mathsf{tp}}$ is η-reduced immediately, before t can be fully reduced to a value.

$$\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]V \to V \qquad\qquad \mu\widehat{\mathsf{tp}}.\uparrow^{\widehat{\mathsf{tp}}} t \to t$$

As before, operational rules are given by deriving the evaluation context from the continuation and meta-continuation used in $\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}$, restricting where reduction may apply.

$$E ::= \square \mid E\ t \mid T\ E \quad F ::= [q]E \quad E^2 ::= \square \mid F[\mu\widehat{\mathsf{tp}}.E^2] \quad F^2 ::= [q^2]E^2$$

With the evaluation contexts, the operational rules are just a coarse-grained representation of the fine-grained reduction rules.

$$\begin{array}{ll} F^2[F[(\lambda x.t)\ V]] \mapsto F^2[F[t\{V/x\}]] & F^2[F[\mu\alpha.c]] \mapsto F^2[c\{F[t]/[\alpha]t\}] \\ F^2[F[\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]V]] \mapsto F^2[F[V]] & F^2[F[\mu\widehat{\mathsf{tp}}.\uparrow^{\widehat{\mathsf{tp}}} t]] \mapsto F^2[F[t]] \\ [q^2]E^2[[*]V] \mapsto [q^2][*]V & \end{array}$$

The reduction rules and operational semantics are sound and complete with respect to the transform $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}$ as in Section 6.

Theorem 5. *If $M \to M'$ then $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![M]\!] = \mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![M']\!]$.
If $\mathcal{C}^2_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}[\![[\bullet][*]t]\!] \mapsto\!\!\!\!\rightarrow V$ then $[\bullet][*]t \mapsto\!\!\!\!\rightarrow [\bullet][*]V$. If $c^2 \mapsto c'^2$ then $c^2 \twoheadrightarrow c'^2$.*

Expressiveness. To encode shift_0 ($\mathcal{S}_0$) in $\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}$, we need to use $\uparrow^{\widehat{\mathsf{tp}}}$ to make the nearest binding of $\widehat{\mathsf{tp}}$ transparent to its body. For comparison, we repeat shift's encoding:

$$\#t = \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]t \quad \mathcal{S} = \lambda h.\mu\alpha.[\widehat{\mathsf{tp}}]h\ (\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x) \quad \mathcal{S}_0 = \lambda h.\mu\alpha.\uparrow^{\widehat{\mathsf{tp}}} h\ (\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x)$$

We can derive the operational rules for the three control operators from the operational semantics of $\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}$. The two-part definition of evaluation contexts mirrors Materzok and Biernacki's[16] presentation of $\mathcal{S}_0$ using contexts and trails. The derived rules show that the only difference between shift and shift_0 is the presence or absence of the reset after capture.

$$E ::= \square \mid E\ t \mid V\ E \qquad D ::= \square \mid D[E[\#\square]]$$

$$\begin{array}{ll} D[E[(\lambda x.t)\ V]] \mapsto D[E[t\{V/x\}]] & D[E[\#V]] \mapsto D[E[V]] \\ D[E'[\#E[\mathcal{S}_0\ V]]] \mapsto D[E'[V\ (\lambda x.\#E[x])]] & D[E[\mathcal{S}\ V]] \mapsto D[V\ (\lambda x.\#E[x])] \end{array}$$

8 Intermediate Language of Dynamic Unbinding: $\lambda\widehat{\mathsf{tp}}^{\Leftarrow}$

Recall that in Section 5, we had to ensure that the dynamic binding was non-cyclic in order to properly model prompts. We accomplished this by backtracking to a previous version of the dynamic environment whenever $\widehat{\mathsf{tp}}$ was applied to a value. While the backtracking semantics of $\lambda\widehat{\mathsf{tp}}^{b}$ and $\widehat{\lambda}^{b}$ can also be used to encode $\mathsf{shift_0}$ and multiple prompt abort, it does not scale well beyond that. The only time we can backtrack the environment is when we have a value to pass to a dynamic variable. Instead, we can generalize the effect by allowing a form of dynamic backtracking over a term. We modify $\lambda\widehat{\mathsf{tp}}^{b}$ with the ability to undo a binding over an unevaluated term t, giving us $\lambda\widehat{\mathsf{tp}}^{\Leftarrow}$.

$$\begin{aligned} c \in \mathit{Closure} &::= [e]t & t \in \mathit{Term} &::= V \mid x \Leftarrow \widehat{\mathsf{tp}}\,\mathbf{in}\,t \mid t_1\ t_2 \\ \widetilde{x} \in \mathit{Var} &::= x \mid \widehat{\mathsf{tp}} & V \in \mathit{Value} &::= x \mid \lambda\widetilde{x}.t \end{aligned}$$

Where e is the empty environment $\bullet$. The new term, $x \Leftarrow \widehat{\mathsf{tp}}\,\mathbf{in}\,t$, has the effect of undoing the most recent binding of $\widehat{\mathsf{tp}}$ in the current environment, exposing the previous dynamic environment to the term t while rebinding the value to x. In essence, $x \Leftarrow \widehat{\mathsf{tp}}\,\mathbf{in}\,t$ is the reverse effect of a dynamic binding. The direct application $\widehat{\mathsf{tp}}\ V$ can be expressed notationally by the new term: $f \Leftarrow \widehat{\mathsf{tp}}\,\mathbf{in}\,f\ V$.

The translation of $\lambda\widehat{\mathsf{tp}}^{\Leftarrow}$ is a modification of the basic environment-passing style transform $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^{b}}$. We must change how the environment is represented in order to express the additional effect on the dynamic environment. We could implement $\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^{\Leftarrow}}$ in a concrete way, representing the environment as a list structure to store a history of dynamic bindings.

$$\begin{aligned} \mathcal{D}_{\lambda\widehat{\mathsf{tp}}^{\Leftarrow}}[\![\lambda\widehat{\mathsf{tp}}.t]\!]\gamma &= \lambda v.\lambda\gamma'.\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^{\Leftarrow}}[\![t]\!]\gamma[\widehat{\mathsf{tp}} \mapsto v] & \gamma[\widehat{\mathsf{tp}} \mapsto v](\widehat{\mathsf{tp}}) &= \langle v, \gamma\rangle \\ \mathcal{D}_{\lambda\widehat{\mathsf{tp}}^{\Leftarrow}}[\![x \Leftarrow \widehat{\mathsf{tp}}\,\mathbf{in}\,t]\!]\gamma &= \mathbf{let}\,\langle x, \gamma'\rangle = \gamma(\widehat{\mathsf{tp}})\,\mathbf{in}\,\mathcal{D}^{u}[\![t]\!]\gamma' \end{aligned}$$

Here, binding $\widehat{\mathsf{tp}}$ to a new value v in an environment γ just stores that binding as the most recent one in γ, while looking up the binding of $\widehat{\mathsf{tp}}$ returns both the value as well as the dynamic environment that was previously active before $\widehat{\mathsf{tp}}$ was bound. The term $x \Leftarrow \widehat{\mathsf{tp}}\,\mathbf{in}\,t$ uses the extra information returned by lookup to evaluate t using the previous environment.

By refunctionalizing [6] the concrete list structure of the environment, we get a translation from $\lambda\widehat{\mathsf{tp}}^{\Leftarrow}$ to the pure λ-calculus.

$$\begin{aligned} \mathcal{D}_{\lambda\widehat{\mathsf{tp}}^{\Leftarrow}}[\![\lambda\widehat{\mathsf{tp}}.t]\!]\gamma &= \lambda v.\lambda\gamma'.\mathcal{D}_{\lambda\widehat{\mathsf{tp}}^{\Leftarrow}}[\![t]\!]\gamma[\widehat{\mathsf{tp}} \mapsto v] & \gamma(\widehat{\mathsf{tp}}) &= \gamma \\ \mathcal{D}_{\lambda\widehat{\mathsf{tp}}^{\Leftarrow}}[\![x \Leftarrow \widehat{\mathsf{tp}}\,\mathbf{in}\,t]\!]\gamma &= \gamma(\widehat{\mathsf{tp}})\ \lambda x.\mathcal{D}^{u}[\![t]\!] & \gamma[\widehat{\mathsf{tp}} \mapsto v] &= \lambda q.q\ v\ \gamma \end{aligned}$$

Looking up the current binding of $\widehat{\mathsf{tp}}$ is just an application of the current environment. The two return values are implemented by having lookup take a continuation which accepts both the value bound to $\widehat{\mathsf{tp}}$ as well as the previous environment. With the new syntax for rolling back the dynamic environment, we can translate $\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}$ into $\lambda\widehat{\mathsf{tp}}^{\Leftarrow}$ in a more concise way, where k is bound directly to $\widehat{\mathsf{tp}}$.

$$\mathcal{C}^u_{\lambda\mu\widehat{\mathsf{tp}}^\uparrow}[\![\mu\widehat{\mathsf{tp}}.c]\!]k = (\lambda\widehat{\mathsf{tp}}.\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^\uparrow}[\![c]\!])\ k$$

$$\mathcal{C}^u_{\lambda\mu\widehat{\mathsf{tp}}^\uparrow}[\![\uparrow^{\widehat{\mathsf{tp}}}\ t]\!] = k \Leftarrow \widehat{\mathsf{tp}}\,\mathbf{in}\,\mathcal{C}_{\lambda\mu\widehat{\mathsf{tp}}^\uparrow}[\![t]\!]k \qquad \mathcal{C}^u_{\lambda\mu\widehat{\mathsf{tp}}^\uparrow}[\![\widehat{\mathsf{tp}}]\!] = \lambda x.k \Leftarrow \widehat{\mathsf{tp}}\,\mathbf{in}\,k\ x$$

9 Delimited Control with Multiple Prompts: $\boldsymbol{\lambda\widehat{\mu}^\uparrow}$ via $\boldsymbol{\widehat{\lambda}^{\Leftarrow}}$

With just the simple addition of multiple prompts, we still don't have enough expressive power in $\lambda\widehat{\mu}$ to encode shift and reset with multiple prompts. The dilemma is that in the presence of multiple prompts, a shift up to prompt $\widehat{\alpha}$ not only captures its immediate context up to the nearest reset, but also captures all contexts bound behind non-matching resets until it finds a reset for $\widehat{\alpha}$. The continuation that shift captures will then restore the captured context as well as seamlessly inserting a partial meta-context in place. In order to express the full power of shift with multiple prompts, we will need some way of directly manipulating the meta-context. This is reminiscent of the way shift_0 removes the most recent binding of $\widehat{\mathsf{tp}}$ and exposes that context to an underlying term. So in order to fully express shift with multiple prompts, we need to incorporate both multiple prompt binding from Section 6 as well as transparent prompts from Section 7. In other words we need to merge multiple dynamic variables from $\widehat{\lambda}^b$ in Section 6 with the ability to roll back the dynamic environment from $\lambda\widehat{\mathsf{tp}}^{\Leftarrow}$ in Section 8.

9.1 Dynamic Unbinding with Multiple Variables: $\boldsymbol{\widehat{\lambda}^{\Leftarrow}}$

The shift operator with multiple prompts only captures a prefix of the meta-context, up to the binding of a specific prompt. What we need is a way to roll back the dynamic environment up to a given binding, while also remembering all the information that would otherwise be discarded. That is, we need to extend the dynamic unbinding effect from $x \Leftarrow \widehat{x}\,\mathbf{in}\,t$ to give us both the value that was stored in $\widehat{x}$ as well as a trace of all the changes to the environment after $\widehat{x}$ was bound. This trace is just a prefix of the current environment, and can be used later to replay the changes over a future state of the environment, extending it with all the dynamic bindings that were removed.

We merge both $\widehat{\lambda}$ and $\lambda\widehat{\mathsf{tp}}^{\Leftarrow}$, by combining both multiple dynamic variables and reversal of dynamic binding, giving us $\widehat{\lambda}^{\Leftarrow}$.

$$c \in \mathit{Closure} ::= [e]t \qquad\qquad e \in \mathit{Environment} ::= \bullet$$
$$t \in \mathit{Term} ::= V \mid t_1\ t_2 \mid \langle \Delta, x\rangle \Leftarrow \widehat{x}\,\mathbf{in}\,t \mid [\Delta]t \qquad\qquad \widetilde{x} \in \mathit{Var} ::= x \mid \widehat{x}$$

The new class of variables, Δ, ranges over environment prefixes. Intuitively, the term $\langle \Delta, x\rangle \Leftarrow \widehat{x}\,\mathbf{in}\,t$ undoes the most recent binding of $\widehat{x}$, binding the value stored in $\widehat{x}$ to x while also capturing the prefix of the environment more recent than $\widehat{x}$ into Δ. Then, the term t is evaluated in the dynamic environment that was in place immediately before $\widehat{x}$ was bound. Closure under the prefix, $[\Delta]t$, extends the surrounding environment with all the dynamic bindings stored in Δ.

Like before, the direct application $\widehat{x}\ V$ can be notationally defined with the more general prefix-capturing form: $\langle \_, f\rangle \Leftarrow \widehat{x}\,\mathbf{in}\,f\ V$.

The semantics of $\widehat{\lambda}^{\Leftarrow}$, like $\lambda\widehat{\mathsf{tp}}^{\Leftarrow}$, requires a redefinition of the dynamic environment. When we query the environment, we now must remember the previously active environment as well as the prefix of bindings that were skipped over in order to find the requested variable. Like in Section 8, we first define the new environment in a concretely, using lists to implement environments and prefixes and tuples to return multiple values.

$$\mathcal{D}_{\widehat{\lambda}^{\Leftarrow}}[\![\langle \Delta, x\rangle \Leftarrow \widehat{x}\,\mathbf{in}\,t]\!]\gamma = \mathbf{let}\,\langle \Delta, x, \gamma'\rangle = \gamma(\widehat{x})\,\mathbf{in}\,\mathcal{D}_{\widehat{\lambda}^{\Leftarrow}}[\![t]\!]\gamma'$$
$$\mathcal{D}_{\widehat{\lambda}^{\Leftarrow}}[\![[\Delta]t]\!]\gamma = (\Delta @ \mathcal{D}_{\widehat{\lambda}^{\Leftarrow}}[\![t]\!])\ \gamma$$

$$\begin{aligned} \gamma[\widehat{x} \mapsto v](\widehat{x}) &= \langle [], v, \gamma\rangle \\ \gamma[\widehat{y} \mapsto v](\widehat{x}) &= \mathbf{let}\,\langle \Delta, u, \gamma'\rangle = \gamma(\widehat{x}) \\ &\quad\ \mathbf{in}\,\langle \Delta[\widehat{y} \mapsto v], u, \gamma'\rangle \end{aligned} \qquad \begin{aligned} []@t &= t \\ \Delta[\widehat{x} \mapsto v]@t &= \Delta@(\lambda\gamma.t\ \gamma[\widehat{x} \mapsto v]) \end{aligned}$$

Dynamic variable lookup now builds up the prefix of bindings that are skipped over in order to find the correct variable. This prefix of bindings can then be used elsewhere to extend a term's dynamic environment. Note that when a prefix extends a term, the bindings in that prefix are more recent than the surrounding dynamic environment and are bound in exactly the same order in which they originally occurred.

Taking the concrete implementation, we can derive the pure λ-calculus encoding by refunctionalizing the data structures. The environment prefix is now a function mapping terms to terms which implements the extension operation from before. Multiple return values are emulated by taking a continuation that accepts each of the three return values separately.

$$\mathcal{D}_{\widehat{\lambda}^{\Leftarrow}}[\![\langle \Delta, x\rangle \Leftarrow \widehat{x}\,\mathbf{in}\,t]\!]\gamma = \gamma(\widehat{x})\ \lambda\Delta.\lambda x.\mathcal{D}_{\widehat{\lambda}^{\Leftarrow}}[\![t]\!] \qquad \mathcal{D}_{\widehat{\lambda}^{\Leftarrow}}[\![[\Delta]t]\!]\gamma = \Delta\ \mathcal{D}_{\widehat{\lambda}^{\Leftarrow}}[\![t]\!]\ \gamma$$
$$\gamma(\widehat{x}) = \gamma\ \ulcorner\widehat{x}\urcorner \qquad \begin{aligned} \gamma[\widehat{x} \mapsto v] = \lambda p.\,&\mathbf{if}\,p \equiv \ulcorner\widehat{x}\urcorner\,\mathbf{then}\,\lambda q.q\ (\lambda t.t)\ v\ \gamma \\ &\mathbf{else}\,\lambda q.\gamma\ p\ \lambda\delta.q\ (\lambda t.\delta\ \lambda\gamma'.t\ \gamma'[\widehat{x} \mapsto v]) \end{aligned}$$

9.2 Capture Up to a Prompt: $\boldsymbol{\lambda\widehat{\mu}^{\uparrow}}$

We are now finally ready to define the full calculus with capture up to an arbitrary prompt. $\lambda\widehat{\mu}^{\uparrow}$ extends $\lambda\widehat{\mu}$ with the ability to capture a prefix of the meta-context up to a prompt, and then later extend the current meta-context with that prefix.

$$\begin{aligned} c^2 \in Command^2 &::= [q^2]c & t \in Term &::= V \mid t_1\ t_2 \mid \mu\widetilde{\alpha}.c \\ c \in Command &::= [q]t \mid \mu^2\Delta \uparrow^{\widehat{\alpha}}.t \mid [\Delta]c & q \in CoTerm &::= \widetilde{\alpha} \mid * \\ q^2 \in CoTerm^2 &::= \bullet & \widetilde{\alpha} \in CoVar &::= \alpha \mid \widehat{\alpha} \end{aligned}$$

The command $\mu^2\Delta \uparrow^{\widehat{\alpha}}.t$ captures a portion of its meta-context as Δ, up to the nearest binding of the prompt $\widehat{\alpha}$. Then, that portion of the meta-context is removed and the most recent binding of $\widehat{\alpha}$ becomes unbound. t is then evaluated in the context formerly bound to $\widehat{\alpha}$ and the remaining meta-context.

The CPS translation from $\lambda\widehat{\mu}^{\uparrow}$ to $\widehat{\lambda}^{\Leftarrow}$ is a merging of $\mathcal{C}_{\lambda\widehat{\mu}}$ and $\mathcal{C}^{u}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}$. The new syntactic forms in $\lambda\widehat{\mu}^{\uparrow}$ can be defined in terms of the intermediate language $\widehat{\lambda}^{\Leftarrow}$. Capturing a portion of the meta-context up to $\widehat{\alpha}$ translates to capturing a prefix of the dynamic environment while unbinding $\widehat{\alpha}$, and extending the meta-context becomes extending the dynamic environment. Like in $\mathcal{C}^{u}_{\lambda\mu\widehat{\mathsf{tp}}^{\uparrow}}$, the invocation of a prompt is changed due to the change in the way dynamic variable lookup is performed. The CPS transform for $\lambda\widehat{\mu}^{\uparrow}$ is an extension of the basic $\mathcal{C}_{\lambda\mu}$ transform.

$$\mathcal{C}_{\lambda\widehat{\mu}^{\uparrow}}[\![\mu^2\Delta\uparrow^{\widehat{\alpha}}.t]\!] = \langle\Delta,\alpha\rangle \Leftarrow \widehat{\alpha}\,\mathbf{in}\,\mathcal{C}_{\lambda\widehat{\mu}^{\uparrow}}[\![t]\!]\alpha$$

$$\mathcal{C}_{\lambda\widehat{\mu}^{\uparrow}}[\![[\Delta]c]\!] = [\Delta]\mathcal{C}_{\lambda\widehat{\mu}^{\uparrow}}[\![c]\!] \qquad \mathcal{C}_{\lambda\widehat{\mu}^{\uparrow}}[\![\widehat{\alpha}]\!] = \lambda x.\langle \_,\alpha\rangle \Leftarrow \widehat{\alpha}\,\mathbf{in}\,\alpha\ x$$

The final derived transform shares a resemblance with the one given by Dybvig *et al.* [8]. However, since we only treat $\mathsf{shift}/\mathsf{shift_0}$-like operators, the environment is an ordered list of bindings, rather than an arbitrary marked stack.

The reduction rules for capture up to a prompt must incrementally move a prefix of the meta-context into the underlying term. Rather than move the complete context bound to a prompt all at once, we can use the ordinary μ-abstraction to capture that context and move it inward to where it is needed. By using an ordinary μ-abstraction, we can capture the context formerly bound to $\widehat{\beta}$ one step at a time.

$$\mu\widehat{\alpha}.\mu^2\Delta\uparrow^{\widehat{\alpha}}.t \to t\{c/[\Delta]c\} \qquad \mu\widehat{\beta}.\mu^2\Delta\uparrow^{\widehat{\alpha}}.t \to \mu\beta.\mu^2\Delta\uparrow^{\widehat{\alpha}}.t\{[\Delta][\beta]\mu\widehat{\beta}.c/[\Delta]c\}$$

When under a non-matching prompt $\widehat{\beta}$, $\mu^2\Delta\uparrow^{\widehat{\alpha}}.t$ must take the context currently bound to $\widehat{\beta}$ and rebind it to $\widehat{\beta}$ wherever Δ is invoked in t. This can be done by giving the context a fresh static name with a static μ-abstraction, and binding $\widehat{\beta}$ to that continuation variable inside of Δ. The static μ-abstraction is then able to reduce further, incrementally absorbing its context and filling in the renewed bindings for $\widehat{\beta}$ inside Δ. If instead $\mu^2\Delta\uparrow^{\widehat{\alpha}}.t$ is under a binding of the prompt $\widehat{\alpha}$, then t is placed in the context bound to $\widehat{\alpha}$ and Δ is eliminated in t, since there is no more prefix for it to capture.

The operational semantics for $\lambda\widehat{\mu}^{\uparrow}$ is an extension of the semantics for $\lambda\widehat{\mu}$. The evaluation contexts and operational rules for $\lambda\widehat{\mu}$ hold for $\lambda\widehat{\mu}^{\uparrow}$. We only need to include the additional rule for the command $\mu^2\Delta\uparrow^{\widehat{\alpha}}.t$.

$$E ::= \Box \mid E\ t \mid V\ E \qquad F ::= [q]E \qquad E^2 ::= \Box \mid F[\mu\widehat{\alpha}.E^2] \qquad F^2 ::= [q^2]E^2$$

$$F^2[F[\mu\widehat{\alpha}.E^2_{\widehat{\alpha}}[\mu^2\Delta\uparrow^{\widehat{\alpha}}.t]]] \mapsto F^2[F[t\{E^2_{\widehat{\alpha}}[c]/[\Delta]c\}]]$$

Where $E^2_{\widehat{\alpha}}$ does not contain a binding for $\widehat{\alpha}$. Like with $\lambda\widehat{\mu}$ the reduction rules and operational semantics are sound and complete with respect to the transform $\mathcal{D}_{\widehat{\lambda}^{\Leftarrow}} \circ \mathcal{C}_{\lambda\widehat{\mu}^{\uparrow}}$.

Theorem 6. *If* $M \to M'$ *then* $\mathcal{D}_{\widehat{\lambda}^{\Leftarrow}} \circ \mathcal{C}_{\lambda\widehat{\mu}^{\uparrow}}[\![M]\!] = \mathcal{D}_{\widehat{\lambda}^{\Leftarrow}} \circ \mathcal{C}_{\lambda\widehat{\mu}^{\uparrow}}[\![M']\!]$.
If $\mathcal{D}_{\widehat{\lambda}^{\Leftarrow}} \circ \mathcal{C}_{\lambda\widehat{\mu}^{\uparrow}}[\![[\bullet][*]t]\!] \mapsto\!\!\!\!\to V$ *then* $[\bullet][*]t \mapsto\!\!\!\!\to [\bullet][*]V$. *If* $c^2 \mapsto c'^2$ *then* $c^2 \twoheadrightarrow c'^2$.

Expressiveness. With capture of the dynamic environment up to a given prompt, we can encode the full behavior of both shift and shift_0 with multiple prompts:

$$\mathcal{S}_0^{\widehat{\alpha}} = \lambda h.\mu\beta.\mu^2\Delta\uparrow^{\widehat{\alpha}}.h\ (\lambda x.\mu\widehat{\alpha}.[\Delta][\beta]x) \qquad \#^{\widehat{\alpha}}t = \mu\widehat{\alpha}.[\widehat{\alpha}]t$$
$$\mathcal{S}^{\widehat{\alpha}} = \lambda h.\mu\beta.\mu^2\Delta\uparrow^{\widehat{\alpha}}.\mu\widehat{\alpha}.[\widehat{\alpha}]h\ (\lambda x.\mu\widehat{\alpha}.[\Delta][\beta]x)$$

$\mathcal{S}^{\widehat{\alpha}}\ h$ captures the current context as well as the dynamic prefix up to the most recent binding of the prompt $\widehat{\alpha}$, which is kept in place. Then, h is given a function which, when applied, will evaluate its argument in the captured context and dynamic prefix under a new binding of $\widehat{\alpha}$. $\mathcal{S}_0^{\widehat{\alpha}}$ is like $\mathcal{S}^{\widehat{\alpha}}$ except that after capturing the dynamic prefix, the prompt $\widehat{\alpha}$ is unbound and the context bound to $\widehat{\alpha}$ is exposed to the given function. The only difference in their encodings is that $\mathcal{S}^{\widehat{\alpha}}$ replaces the reset of $\widehat{\alpha}$ after capturing the meta-context, while $\mathcal{S}_0^{\widehat{\alpha}}$ does not.

Using the operational semantics from Section 9.2, we can derive the operational semantics for our encoding of the $\#^{\widehat{\alpha}}$, $\mathcal{S}^{\widehat{\alpha}}$, and $\mathcal{S}_0^{\widehat{\alpha}}$ control operators.

$$E ::= \Box \mid E\ t \mid V\ E \qquad D ::= \Box \mid D[E[\#^{\widehat{\alpha}}\Box]]$$

$$D[E[(\lambda x.t)\ V]] \mapsto D[E[t\{V/x\}]] \qquad D[E[\#^{\widehat{\alpha}}V]] \mapsto D[E[V]]$$
$$D[E[\#^{\widehat{\alpha}}D'[E'[\mathcal{S}^{\widehat{\alpha}}\ V]]]] \mapsto D[E[\#^{\widehat{\alpha}}V\ (\lambda x.\#^{\widehat{\alpha}}D'[E'[x]])]] \quad \textbf{where}\ \#^{\widehat{\alpha}} \notin D'$$
$$D[E[\#^{]}D'[E'[\mathcal{S}_0^{\widehat{\alpha}}\ V]\widehat{\alpha}]] \mapsto D[E[V\ (\lambda x.\#^{\widehat{\alpha}}D'[E'[x]])]] \quad \textbf{where}\ \#^{\widehat{\alpha}} \notin D'$$

10 Conclusion

We have provided a calculus which allows us to study delimited control with multiple prompts. To do this, we used an intermediate language of dynamic binding in order to define the semantics of multiple prompts. Kiselyov *et al.* [15] have also investigated the relationship between dynamic binding and delimited control by giving a language that gives the programmer access to both. Interestingly, their approach is the opposite of ours. They directly define the dynamic binding in terms of delimited control with multiple prompts. On the other hand, we use the conceptually simpler notion of dynamic binding as a stepping stone for understanding delimited control with multiple prompts.

Our interest in delimited control with multiple prompts came from the desire of formalizing a call-by-need abstract machine. Both call-by-value and call-by-name λ-calculi can be presented in the sequent calculus style as abstract machines, where the active redex is always at the top of the term [3]. With call-by-need, however, the active redex can become buried under bindings of delayed terms during evaluation. As discussed by Garcia *et al.* [13], call-by-need can be represented in terms of delimited control with multiple prompts. In that spirit, we want to achieve a deeper understanding of the equational theory of delimited control in the presence of more than one prompt, aiming at formalizing classical lazy evaluation in the sequent setting. As future work, we plan to tackle completeness of the equational theory with respect to the CPS semantics. We are also interested in understanding the type theory that arises from the CPS semantics.

Acknowledgments. We would like to thank Olivier Danvy, Hugo Herbelin, Alexis Saurin, Ian Zerny, and the anonymous referees for their valuable comments and suggestions. Paul Downen and Zena M. Ariola have been supported by NSF grant CCF-0917329. This research has been developed under INRIA Équipe Associée SEMACODE.

References

1. Ariola, Z.M., Herbelin, H.: Control reduction theories: The benefit of structural substitution. J. Funct. Program. 18, 373–419 (2008)
2. Ariola, Z.M., Herbelin, H., Sabry, A.: A type-theoretic foundation of delimited continuations. HOSC 22(3), 233–273 (2009)
3. Curien, P.L., Herbelin, H.: The duality of computation. In: ICFP 2000, pp. 233–243 (2000)
4. Danvy, O.: On evaluation contexts, continuations, and the rest of the computation. In: ACM SIGPLAN Workshop on Continuations, pp. 13–23 (2004)
5. Danvy, O., Filinski, A.: A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Copenhagen, Denmark (1989)
6. Danvy, O., Millikin, K.: Refunctionalization at work. Science of Computer Programming 74(8), 534–549 (2009)
7. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, pp. 151–160. ACM Press (1990)
8. Dybvig, R., Jones, S.P., Sabry, A.: A monadic framework for delimited continuations. Journal of Functional Programming 17(06), 687–730 (2007)
9. Felleisen, M.: The theory and practice of first-class prompts. In: POPL 1988 (1988)
10. Felleisen, M., Friedman, D., Kohlbecker, E.: A syntactic theory of sequential control. Theoretical Computer Science 52(3), 205–237 (1987)
11. Filinski, A.: Representing monads. In: POPL 1994, pp. 446–457 (1994)
12. Filinski, A.: Representing layered monads. In: POPL 1999, pp. 175–188 (1999)
13. Garcia, R., Lumsdaine, A., Sabry, A.: Lazy evaluation and delimited control. In: Proceedings of POPL 2009, pp. 153–164. ACM, New York (2009)
14. Kiselyov, O.: Delimited control in OCaml, abstractly and concretely: System description. Functional and Logic Programming, 304–320 (2010)
15. Kiselyov, O., Shan, C.-c., Sabry, A.: Delimited dynamic binding. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, pp. 26–37. ACM, New York (2006)
16. Materzok, M., Biernacki, D.: Subtyping delimited continuations. In: Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, pp. 81–93. ACM, New York (2011)
17. Moreau, L.: A syntactic theory of dynamic binding. HOSC 11(3), 233–279 (1998)
18. Parigot, M.: $\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 190–201. Springer, Heidelberg (1992)
19. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the 25th ACM National Conference, pp. 717–740 (1972)
20. Sitaram, D., Felleisen, M.: Reasoning with continuations II: Full abstraction for models of control. In: Conference on Lisp and Functional Programming (1990)