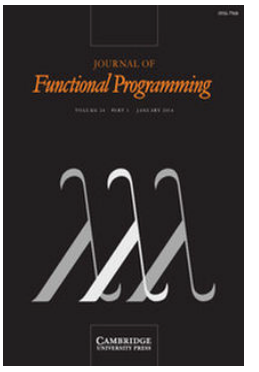

Delimited control and computational effects

PAUL DOWNEN and ZENA M. ARIOLA

Journal of Functional Programming / Volume 24 / Issue 01 / January 2014, pp 1 - 55
DOI: 10.1017/S0956796813000312, Published online: 22 January 2014

JFP **24** (1): 1–55, 2014.
doi:10.1017/S0956796813000312 First published online 22 January 2014

Delimited control and computational effects

PAUL DOWNEN and ZENA M. ARIOLA
University of Oregon, Eugene, OR, USA
(*e-mail:* {pdownen,ariola}@cs.uoregon.edu)

Abstract

We give a framework for 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. For each language of interest, we fully characterize its semantics in terms of a reduction semantics, operational semantics, continuation-passing style transform, and abstract machine. Furthermore, the control operations are expressed in terms of fine-grained primitives that can be used to build well-known, higher-level control operators. In order to illustrate the expressive power provided by various languages, we show how other computational effects can be encoded in terms of these control operators.

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 first appeared in the Scheme programming language as well as Smalltalk, SML, and OCaml. Today call/cc style support for control effects is available in programming languages such as Ruby, Stackless Python, and Squeak. The 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 with 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. There are several formulations of delimited control that differ in how the prompt is managed, including the control and prompt operators presented by Felleisen and Sitaram (1988, 1990a) and shift and reset presented by Danvy & Filinski (1989). Delimited control is expressive enough to simulate mutable state. In fact, Filinski (1994, 1999) showed that

the combination of shift and reset is enough to give a direct style encoding for any effect written in monadic style.

Extending delimited control with multiple prompts that can be referred to by name is a key part to building modular control abstractions that do not interfere with one another. Flatt *et al.* (2007) added delimited control to the PLT Scheme implementation, and in the interest of reducing the number of primitives, they define exception handling control operators in terms of delimited control. However, using only a single prompt for delimiting both primitive delimited control operators and scope of exception handlers leads to interference between the two.

Example 1
In the following program from Flatt *et al.* (2007):

$$\mathsf{reset}(\mathsf{raise}\,0)\,\mathbf{handle}\,n \Rightarrow n+1$$

The expression $\mathsf{raise}\,0$ would be 'caught' by the reset, rather than by the exception handler, giving the wrong result 0.

The solution used by Flatt *et al.* (2007) is to use multiple named prompts. When both reset and **handle** set differently named prompts, and raise refers to the prompt set by **handle**, the interference is avoided and we get the correct result 1. However, by using multiple prompts, the interaction between different high-level control operators is more intricate.

Example 2
Consider a more complicated program that uses both exception handling and a short-circuiting abort operator which jumps to the nearest reset:

$$(\mathsf{reset}[(\mathsf{abort}(\mathsf{raise}\,0) * 5)\,\mathbf{handle}\,n \Rightarrow n+1])$$
$$\mathbf{handle}\,n \Rightarrow n+2$$

Does abort first evaluate $\mathsf{raise}\,0$ before jumping to the reset, thus raising the exception inside the innermost handler and returning the result 1? Or instead does abort first jump to the reset and then evaluate $\mathsf{raise}\,0$, thereby skipping the innermost handler and returning the result 2? Or something in-between? The answer depends on fundamental choices made during the definition of these operators. Therefore, to understand how these different effects interact with one another, we need to understand the behavior of delimited control with multiple named prompts.

Dybvig *et al.* (2007) 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 (2010). In addition, Kiselyov *et al.* (2006) give a language that combines both delimited control and dynamic variables, showing that the two effects interact in subtle ways.

The goal of this paper is to provide a framework for delimited control and its extensions. In line with this goal, we develop a fine-grained reduction theory for delimited control with multiple prompts. Ariola *et al.* (2009) have formalized abortive and delimited control 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 delimited control with multiple dynamic 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 as follows:

- 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, in the sense of Felleisen (1991). Each extension enables direct encodings of additional, useful language constructs, and arises as a natural extension of a less expressive or intermediate language.
- A reduction theory for control with multiple dynamic prompts that is sound with respect to the *continuation passing style* (CPS) semantics and complete 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.[1] In Sections 2, 3, 4, 6, 7, and 8, we define our languages of interest. We start with the λ-calculus (in Section 2), and extend it with control ($\lambda\mu$ in Section 3) and then with delimited control ($\lambda\mu\widehat{\mathsf{tp}}$ in Section 4). Then we branch out in two separate directions, extending $\lambda\mu\widehat{\mathsf{tp}}$ with multiple static prompts ($\lambda\widehat{\mu}$ in Section 6) and also a single dynamic prompt ($\lambda\mu\widehat{\mathsf{tp}}_0$ in Section 7). Finally, we bring $\lambda\widehat{\mu}$ and $\lambda\mu\widehat{\mathsf{tp}}_0$ together, giving us a language of delimited control with multiple dynamic prompts ($\lambda\widehat{\mu}_0$ in Section 8).

We present the semantics of each language in four different ways: as a *CPS transformation* from the source language to a target calculus, such as the pure λ-calculus, then as a set of *reduction rules*, and finally as an *operational semantics* and *abstract machine*. The CPS transformation implements an evaluator for the language written in the λ-calculus, and is used as our primary reference point for the definition of 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 restriction on the reduction rules and equivalent small-step evaluator for the CPS transformation. The abstract machine captures the same essential steps as the CPS transform, since it is derived by defunctionalizing the CPS transform (Reynolds 1972; Danvy 2004). We however present both, since some readers may find the abstract machine easier to follow.

[1] This is a revised and expanded version of 'A Systematic Approach to Delimited Control with Multiple Prompts,' which appeared in the 21st European Symposium on Programming, Tallinn, Estonia (Downen & Ariola 2012).

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

Fig. 1. The syntax of the pure λ-calculus.

$$\beta_v : \quad (\lambda x.t)\ V \to t\{V/x\}$$

Fig. 2. Call-by-value β reduction in the pure λ-calculus.

To motivate this development, we demonstrate the expressiveness of the various languages of control by encoding other well-known computational effects. In Section 3, we give an encoding of Scheme's call/cc in Parigot's $\lambda\mu$-calculus. In Section 9.1, we show how the delimited control operators shift and reset can be encoded in $\lambda\mu\widehat{\mathsf{tp}}$, and furthermore, how delimited control acts as a 'universal' effect (Filinski 1994, 1999). Finally, we demonstrate the utility of delimited control with multiple named prompts in Sections 9.2 and 9.4 by encoding exception handling of multiple exceptions and state with multiple cells.

In Sections 5, 7, and 8, we present intermediate languages which are used as stepping stones for defining CPS transformations of our primary languages, and provide a good framework for designing extensions during the development to delimited control with multiple named prompts.

2 Lambda calculus: λ

Syntax. The syntax of λ-calculus terms includes variables (x), function abstraction ($\lambda x.t$), and function application ($t_1\ t_2$) as shown in Figure 1. Unless otherwise specified, we let the set of *Values* be variables and function abstractions: $V ::= x \mid \lambda x.t$.

Reduction. In this paper, we will focus on the call-by-value setting, which restricts substitution to values, as described by the β_v reduction rule given in Figure 2. We allow this reduction rule to be applied anywhere inside a λ-calculus term, giving us a *reduction semantics* for the λ-calculus.

For presentational purposes, we also use let-bindings, which are defined as syntactic sugar in terms of ordinary functions and application:

$$(\mathbf{let}\, x = t\, \mathbf{in}\, u) = (\lambda x.u)\ t$$

Since we are using call-by-value function application, this standard notation captures the *sequencing* behavior of the call-by-value λ-calculus. In the expression $\mathbf{let}\, x = t\, \mathbf{in}\, u$, first we evaluate t since it has priority. Then, only once t has been reduced to a value V can we substitute V for x and continue evaluation of u.

Operational semantics. In order to eliminate the non-deterministic choice of which reduction to perform, we can restrict reduction by selecting a specific, *standard* reduction for every term, giving us an *operational semantics*. One way of defining which reduction is the standard one is by giving an *evaluation context* which points out the location of the standard reduction. For the call-by-value λ-calculus, we have the set of evaluation contexts

$$E \in EvCxt ::= \Box \mid E\ t \mid V\ E$$

$$E[(\lambda x.t)\ V] \mapsto E[t\{V/x\}]$$

Fig. 3. Call-by-value evaluation contexts and operational semantics for the pure λ-calculus.

$$\mathscr{C}_\lambda[\![V]\!] = \lambda k.k\ \mathscr{C}_\lambda[\![V]\!]^V \qquad \mathscr{C}_\lambda[\![x]\!]^V = x$$
$$\mathscr{C}_\lambda[\![t_1\ t_2]\!] = \lambda k.\mathscr{C}_\lambda[\![t_1]\!]\lambda f.\mathscr{C}_\lambda[\![t_2]\!]\lambda s.f\ s\ k \qquad \mathscr{C}_\lambda[\![\lambda x.t]\!]^V = \lambda x.\mathscr{C}_\lambda[\![t]\!]$$

Fig. 4. Call-by-value CPS transform of the pure λ-calculus.

given in Figure 3. The operational semantics for the λ-calculus then restricts evaluation so that reduction, as described by the β_v rule in Figure 2, is only allowed in an evaluation context.

Notation: From now on, we use $\twoheadrightarrow$ and $\mapsto\!\!\!\!\twoheadrightarrow$ to denote the reflexive, transitive closure of $\rightarrow$ and $\mapsto$, respectively.

CPS transform. An alternative way of presenting the semantics is to perform a translation that hard-wires the evaluation strategy into the term itself. The translation is called a CPS transform; it splits a term into the current work to be done and the rest of the computation, which is called a *continuation*. In Figure 4 we give a transform based on Plotkin's (1975) call-by-value CPS transform of the λ-calculus. 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 t_1 is evaluated, and its value is bound to f. Second, the argument t_2 is evaluated and its value is bound to s. 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. This initial continuation k_ι is initialized as the identity function: $k_\iota = \lambda x.x$.

Explicit top-level. 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 co-term $*$ and specify that running a term consists of coupling that term with $*$, written as $[*]t$, which we call a *command*. This can be done in an initial phase that precedes evaluation similar to a final phase that turns the final state of a program into a final answer. Operationally, the command $[*]t$ is interpreted as evaluating the term t until a value is reached, which is taken as the final answer,

$$[*]t \text{ initial program} \qquad [*]V \text{ final answer}$$

For our purposes, a program in the λ-calculus may be any command and does not need to be closed, but may contain free variables. We extend the syntax of the pure λ-calculus

$$t \in \mathit{Term} ::= V \mid t_1\ t_2 \qquad c \in \mathit{Command} ::= [q]t$$
$$V \in \mathit{Value} ::= x \mid \lambda x.t \qquad q \in \mathit{CoTerm} ::= *$$

Fig. 5. The λ-calculus with an explicit top-level.

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

$$\mathscr{C}_\lambda[\![V]\!] = \lambda k.k\ \mathscr{C}_\lambda[\![V]\!]^V \qquad \mathscr{C}_\lambda[\![x]\!]^V = x$$
$$\mathscr{C}_\lambda[\![t_1\ t_2]\!] = \lambda k.\mathscr{C}_\lambda[\![t_1]\!]\lambda f.\mathscr{C}_\lambda[\![t_2]\!]\lambda s.f\ s\ k \qquad \mathscr{C}_\lambda[\![\lambda x.t]\!]^V = \lambda x.\mathscr{C}_\lambda[\![t]\!]$$

Fig. 6. Call-by-value CPS transform for the λ-calculus with an explicit top-level.

in Figure 1 with two new syntactic categories, giving us the syntax shown in Figure 5. We also extend our CPS transform $\mathscr{C}_\lambda$ from Figure 4 with clauses for commands and the constant $*$ as shown in Figure 6. The interpretation of the command $[q]t$ is to evaluate the term t with the co-term q, which means to pass the continuation represented by q to the term. The co-term $*$ stands for the initial continuation that just returns the value it is given without modifying it.

Abstract machine. To give a complete account of the various ways of describing the semantics of the λ-calculus, we also present an abstract machine, which is in between the operational semantics and CPS transform in the sense of Danvy's (2004) inter-derivation of evaluators. The states of the abstract machine for the λ-calculus are described in Figure 7. Rather than using a top-down style for evaluation contexts, the machine keeps track of a *stack* of *frames*, ending with a co-term q, where each frame is a single step of a call-by-value evaluation context E. The notation $F^*[F]$ evokes the connection with the operation that plugs a frame into a top-down evaluation context, $[*](E[F])$, except that here it is syntax for pushing the frame F on top of the stack F^*. For presentational purposes, we will avoid explicitly handling the bindings of static variables, and instead use substitution during execution of the machine.

The steps of the abstract machine are given in Figure 8. The purpose of the refocus phase in the machine is to find the next reduction to perform, the apply phase fills a context with a value, and the reduce phase performs the reduction step. Also note how this abstract machine can be viewed as a defunctionalized form of the $\mathscr{C}_\lambda$ transform, using Danvy's (2004) technique of deriving abstract machines from CPS transforms (Reynolds 1972). The refocus steps implement the definition of $\mathscr{C}_\lambda[\![c]\!]$ and β-reductions of the form $\mathscr{C}_\lambda[\![t]\!]k$, and the apply steps implement β-reduction of a continuation applied to a value. Finally, the reduce step is a representation of the CPS program $(\lambda x.\mathscr{C}_\lambda[\![t]\!])\ \mathscr{C}_\lambda[\![V]\!]^V\ k$ that results from the translation of $\mathscr{C}_\lambda[\![(\lambda x.t)\ V]\!]k$.

Correctness. Having seen several presentations of the semantics of the call-by-value λ-calculus, we would like some assurance that they are compatible so that they all agree on the meaning of the λ-calculus terms. The operational semantics was derived as a restriction on the reduction theory, where reduction may only occur in one specific place. Likewise, the abstract machine can be derived as the defunctionalized version of the CPS transform

$$S ::= \langle c \rangle_{\mathsf{refocus}} \mid \langle t, F^* \rangle_{\mathsf{refocus}} \mid \langle F^*, V \rangle_{\mathsf{apply}} \mid \langle t, F^* \rangle_{\mathsf{reduce}} \mid \langle V \rangle_{\mathsf{done}}$$

$$F \in \mathit{Frame} ::= \Box\, t \mid V\, \Box \qquad\qquad F^* \in \mathit{Stack} ::= q \mid F^*[F]$$

Fig. 7. States and evaluation stacks of the call-by-value λ-calculus abstract machine.

$$\begin{array}{rr}
\langle [*]t \rangle_{\mathsf{refocus}} \rightsquigarrow \langle t, * \rangle_{\mathsf{refocus}} & \langle F^*[\Box\, t], V \rangle_{\mathsf{apply}} \rightsquigarrow \langle t, F^*[V\, \Box] \rangle_{\mathsf{refocus}} \\
\langle t\ t', F^* \rangle_{\mathsf{refocus}} \rightsquigarrow \langle t, F^*[\Box\, t'] \rangle_{\mathsf{refocus}} & \langle F^*[V'\, \Box], V \rangle_{\mathsf{apply}} \rightsquigarrow \langle V'\ V, F^* \rangle_{\mathsf{reduce}} \\
\langle V, F^* \rangle_{\mathsf{refocus}} \rightsquigarrow \langle F^*, V \rangle_{\mathsf{apply}} & \langle *, V \rangle_{\mathsf{apply}} \rightsquigarrow \langle V \rangle_{\mathsf{done}}
\end{array}$$

$$\langle (\lambda x.t)\ V, F^* \rangle_{\mathsf{reduce}} \rightsquigarrow \langle t\{V/x\}, F^* \rangle_{\mathsf{refocus}}$$

Fig. 8. Call-by-value abstract machine for the λ-calculus.

(Reynolds 1972; Danvy 2004) so that the continuations are represented as concrete data structures and their behavior is implemented by the apply steps. Therefore, we connect these two separate worlds by relating the reduction theory to the CPS transform. The first check of correctness is *soundness* of the call-by-value λ-calculus reduction theory with respect to the $\mathscr{C}_\lambda$ transform, establishing that the reductions are not too strong. This means that every reduction in the source language is translated to an equality in the target language.

Theorem 2.1 (*Soundness*)
If $M \twoheadrightarrow M'$, then $\mathscr{C}_\lambda[\![M]\!] =_\beta \mathscr{C}_\lambda[\![M']\!]$.

We would also like some guarantee that the opposite property holds, and that the reductions are strong enough. Therefore, we also consider the notion of *operational completeness*, meaning that if the CPS transform is capable of finding a final answer (a value in the CPS λ-calculus), then the operational semantics reaches the same final answer (up to β-equality in the CPS λ-calculus).

Theorem 2.2 (*Evaluation*)
If $\mathscr{C}_\lambda[\![c]\!] =_\beta V$, then there is a final answer c' such that $c \mapsto\!\!\!\!\twoheadrightarrow c'$ and $\mathscr{C}_\lambda[\![c']\!] =_\beta V$.

A final answer in the call-by-value λ-calculus, as defined in Figure 5, is a command of the form $[*]V$. Furthermore, the operational semantics is a restriction on the reduction theory.

Theorem 2.3
If $c \mapsto\!\!\!\!\twoheadrightarrow c'$, then $c \twoheadrightarrow c'$.

Therefore, the reduction theory is also strong enough to find the same final answer as the operational semantics and the CPS transform. This property holds for all the languages of control to follow. For further discussion and proofs of these theorems, see Section A in the appendix (online only), available as supplementary material at `http://dx.doi.org/10.1017/S0956796813000312`.

3 Lambda calculus with control: Parigot's $\lambda\mu$

Felleisen *et al.* (1987; 1992) 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

$$t \in \mathit{Term} ::= V \mid t_1\ t_2 \mid \mu\alpha.c \qquad c \in \mathit{Command} ::= [q]t$$
$$V \in \mathit{Value} ::= x \mid \lambda x.t \qquad q \in \mathit{CoTerm} ::= \alpha \mid *$$

Fig. 9. The syntax of the $\lambda\mu$-calculus.

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

Fig. 10. Call-by-value reduction theory of the $\lambda\mu$-calculus.

reinstall this context by invoking that function. The function representing a continuation never returns to the call site. Here we instead follow Parigot's (1992) approach because it provides a reduction theory which more accurately simulates the operational semantics (Ariola & Herbelin 2008). Unlike the continuations constructed by call/cc in Scheme, in Parigot's $\lambda\mu$, continuations are not functions. Similar to the top-level, continuations belong to a separate syntactic category of co-terms, and the invocation of a continuation is a command. Intuitively, terms are producers of values, whereas continuations are consumers of values.

Syntax. The syntax of $\lambda\mu$ extends the λ-calculus terms and co-terms from Figure 5 as shown in Figure 9. In addition to pure λ-calculus terms, we also have a μ-abstraction, which captures the evaluation context of a term and binds it to a co-variable α in a command c. The syntax of co-terms is also extended to include co-variables α, which allow us to plug a term into the evaluation context bound to α.

Reduction. The reduction semantics of $\lambda\mu$ is given by the rules shown in Figure 10. Reduction depends on the notion of evaluation context, which is shown by the presence of the call-by-value frame F, as previously defined in Figure 7, in the reduction rules. These rules are fine-grained due to the fact that only a single evaluation context frame is manipulated at a time rather than a large, unbounded context. 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*; $c\{[\alpha](F[t])/[\alpha]t\}$ should be read as: substitute each occurrence of $[\alpha]t$ in command c with $[\alpha](F[t])$.

Operational semantics. When iterated, these two reductions for μ-abstractions perform the big-step capturing reduction that substitutes the entire evaluation context up to the top of the command, giving us an operational semantics for $\lambda\mu$ shown in Figure 11. Since computation in the $\lambda\mu$-calculus operates on commands, the operational rules are given in terms of D, *complete* evaluation contexts that are terminated by a co-term q. These contexts are 'complete' in the sense that they represent the entire future of the program. Unlike ordinary evaluation contexts, a complete evaluation context cannot be extended from the outside. For example, given E and a term t to plug into it, we can observe the result of a

$$E ::= \Box \mid E\ t \mid V\ E \qquad\qquad D ::= [q]E$$

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

Fig. 11. Call-by-value evaluation contexts and operational semantics for the $\lambda\mu$-calculus.

$$\mathscr{C}_{\lambda\mu}[\![[q]t]\!] = \mathscr{C}_{\lambda\mu}[\![t]\!]\ \mathscr{C}_{\lambda\mu}[\![q]\!] \qquad\qquad \mathscr{C}_{\lambda\mu}[\![*]\!] = \lambda x.x$$
$$\mathscr{C}_{\lambda\mu}[\![\alpha]\!] = \alpha$$
$$\mathscr{C}_{\lambda\mu}[\![V]\!] = \lambda k.k\ \mathscr{C}_{\lambda\mu}[\![V]\!]^V$$
$$\mathscr{C}_{\lambda\mu}[\![t_1\ t_2]\!] = \lambda k.\mathscr{C}_{\lambda\mu}[\![t_1]\!]\lambda f.\mathscr{C}_{\lambda\mu}[\![t_2]\!]\lambda s.f\ s\ k \qquad \mathscr{C}_{\lambda\mu}[\![x]\!]^V = x$$
$$\mathscr{C}_{\lambda\mu}[\![\mu\alpha.c]\!] = \lambda k.(\lambda\alpha.\mathscr{C}_{\lambda\mu}[\![c]\!])\ k \qquad \mathscr{C}_{\lambda\mu}[\![\lambda x.t]\!]^V = \lambda x.\mathscr{C}_{\lambda\mu}[\![t]\!]$$

Fig. 12. Call-by-value CPS transform for the $\lambda\mu$-calculus.

function call $(E[t]\ u)$. However, with a D, $D[t]$ forms a complete command which does not return a value to a calling context.

CPS transform. We define the CPS transform of $\lambda\mu$ by extending $\mathscr{C}_\lambda$ for definitions of the new syntax, giving us the $\mathscr{C}_{\lambda\mu}$ transform in Figure 12. The only change from the $\mathscr{C}_\lambda$ transform from Figure 6 is the translation of a μ-abstraction as a function that binds its current continuation and the translation of a co-variable α as an ordinary λ-calculus variable in the target program. Note that complete evaluation contexts D from a source program correspond exactly with continuations in the CPS transform, represented as functions in the λ-calculus.

Abstract machine. Extending the λ-calculus abstract machine only requires that we add additional refocus step and reduce step that implement the μ-abstraction. The steps of the machine are given in Figure 13. As before, F^* is a sequence of frames ending with a co-term: either a co-variable α or $*$. Note that during execution, this machine steps outside the syntax of $\lambda\mu$ in the step that substitutes a stack F^* for a co-variable α. Therefore, during execution and in final answers, the set of co-terms q is extended to also include F^* in addition to co-variables α.

Correctness. As an extension of the pure, call-by-value λ-calculus, we have the same notion of correctness of the $\lambda\mu$ reduction theory with respect to the $\mathscr{C}_{\lambda\mu}$ CPS transform.

Theorem 3.1 (*Soundness*)
If $M \twoheadrightarrow M'$, then $\mathscr{C}_{\lambda\mu}[\![M]\!] =_\beta \mathscr{C}_{\lambda\mu}[\![M']\!]$.

Theorem 3.2 (*Evaluation*)
If $\mathscr{C}_{\lambda\mu}[\![c]\!] =_\beta V$, then there is a final answer c' such that $c \twoheadrightarrow c'$ and $\mathscr{C}_{\lambda\mu}[\![c']\!] =_\beta V$.

Note that since the $\lambda\mu$-calculus allows for control effects, terms may return a result to the same location more than once. Therefore, we emphasize that the answers given by the $\lambda\mu$-calculus must be *final*, meaning that there can be no possible use of control effects to

$$\langle [q]t\rangle_{\mathsf{refocus}} \rightsquigarrow \langle t,q\rangle_{\mathsf{refocus}} \qquad \langle F^*[\Box\ t],V\rangle_{\mathsf{apply}} \rightsquigarrow \langle t,F^*[V\ \Box]\rangle_{\mathsf{refocus}}$$
$$\langle t\ t',F^*\rangle_{\mathsf{refocus}} \rightsquigarrow \langle t,F^*[\Box\ t']\rangle_{\mathsf{refocus}} \qquad \langle F^*[V'\ \Box],V\rangle_{\mathsf{apply}} \rightsquigarrow \langle V'\ V,F^*\rangle_{\mathsf{reduce}}$$
$$\langle \mu\alpha.c,F^*\rangle_{\mathsf{refocus}} \rightsquigarrow \langle \mu\alpha.c,F^*\rangle_{\mathsf{reduce}} \qquad \langle *,V\rangle_{\mathsf{apply}} \rightsquigarrow \langle V\rangle_{\mathsf{done}}$$
$$\langle V,F^*\rangle_{\mathsf{refocus}} \rightsquigarrow \langle F^*,V\rangle_{\mathsf{apply}}$$

$$\langle (\lambda x.t)\ V,F^*\rangle_{\mathsf{reduce}} \rightsquigarrow \langle t\{V/x\},F^*\rangle_{\mathsf{refocus}}$$
$$\langle \mu\alpha.c,F^*\rangle_{\mathsf{reduce}} \rightsquigarrow \langle c\{F^*/\alpha\}\rangle_{\mathsf{refocus}}$$

Fig. 13. Abstract machine for call-by-value Parigot's $\lambda\mu$ calculus.

$$\begin{aligned}\mathscr{K} &= \lambda h.\mu\alpha.[\alpha]h\ (\lambda x.\mu\_.[\alpha]x)\\ \mathscr{A}\ t &= \mu\_.[*]t\\ \mathscr{C} &= \lambda h.\mathscr{K}\ (\lambda k.\mathscr{A}\ (h\ k))\\ &= \lambda h.\mu\alpha.[*]h\ (\lambda x.\mu\_.[\alpha]x)\end{aligned}$$

Fig. 14. Encodings of the $\mathscr{K}$, $\mathscr{A}$, and $\mathscr{C}$ operators in $\lambda\mu$.

change the result of the program. A final answer of the $\lambda\mu$-calculus is still a command of the form $[*]V$. For further discussion and proofs of these theorems, see Sections A.1 and A.2 in the appendix.

Expressiveness. Even though we have a CPS transform that relates all $\lambda\mu$ terms to pure λ-calculus terms, intuitively the $\lambda\mu$-calculus is more expressive than the λ-calculus. When speaking of 'expressiveness,' we refer to the ability to encode operations or computational behavior as a local, compositional macro-expansion, without the need for a global transformation of the entire program, as described by Felleisen (1991). Parigot's $\lambda\mu$ gives us the ability to express the call/cc ($\mathscr{K}$) control operator from the Scheme programming language as shown in Figure 14. In addition, equipping $\lambda\mu$ with the top-level constant $*$ allows us to express the abort ($\mathscr{A}$) operator and Felleisen's $\mathscr{C}$ operator, which is definable in terms of call/cc and abort.

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

Delimiting control means temporarily redefining the top-level in a program, limiting the extent to which the evaluation context may be captured. Examples of delimited control operators are Felleisen's $\mathscr{F}$ and prompt operators (Felleisen & Friedman 1987; Felleisen 1988) as well as the shift and reset operators given in the seminal paper of Danvy & Filinski (1989). The prompt operator is shown to be necessary in providing a fully abstract model of λ-calculus by Sitaram & Felleisen (1990b).

Syntax. From Ariola *et al.* (2009), it is shown that delimited control can be explained by replacing the top-level constant $*$ with the re-bindable dynamic continuation variable $\widehat{\mathsf{tp}}$. The syntax of $\lambda\mu\widehat{\mathsf{tp}}$ is given in Figure 15. The only change from the $\lambda\mu$-calculus presented in Figure 9 is that the top-level constant $*$ is replaced with $\widehat{\mathsf{tp}}$, and the new μ-abstraction $\mu\widehat{\mathsf{tp}}.c$ is added, which binds $\widehat{\mathsf{tp}}$ in c.

$$t \in \mathit{Term} ::= V \mid t_1\ t_2 \mid \mu q.c \qquad c \in \mathit{Command} ::= [q]t$$
$$V \in \mathit{Value} ::= x \mid \lambda x.t \qquad q \in \mathit{CoTerm} ::= \alpha \mid \widehat{\mathsf{tp}}$$

Fig. 15. The syntax of the $\lambda\mu\widehat{\mathsf{tp}}$-calculus.

$$\begin{aligned}(\lambda x.t)\ V &\to t\{V/x\}\\ F[\mu\alpha.c] &\to \mu\alpha.c\{[\alpha](F[t])/[\alpha]t\}\\ [q]\mu\alpha.c &\to c\{q/\alpha\}\\ \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]V &\to V\end{aligned}$$

Fig. 16. Call-by-value reduction theory of the $\lambda\mu\widehat{\mathsf{tp}}$-calculus.

Reduction. 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 in Figure 16 by adding the following reduction rule to the reduction theory of $\lambda\mu$ given in Figure 10:

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

Note that the renaming rule

$$[q]\mu\alpha.c \to c\{q/\alpha\}$$

allows a μ-abstraction to bind α to the dynamic $\widehat{\mathsf{tp}}$ in addition to other static co-variables. When $\widehat{\mathsf{tp}}$ is substituted for α in a command, it may be *captured* by nearer bindings for $\widehat{\mathsf{tp}}$, as in the following example:

$$[\widehat{\mathsf{tp}}]\mu\alpha.[\alpha](1+\mu\widehat{\mathsf{tp}}.[\alpha]2) \to [\widehat{\mathsf{tp}}](1+\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]2) \to [\widehat{\mathsf{tp}}](1+2) \to [\widehat{\mathsf{tp}}]3$$

CPS transform. We extend the $\mathscr{C}_{\lambda\mu}$ transform from Figure 12 to give $\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}$, the CPS transform for $\lambda\mu\widehat{\mathsf{tp}}$ shown in Figure 17. 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, the 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, since the term $\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!]$ is an application instead of a value. One can remedy the situation by taking the output from $\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}$ and running it through the CPS transform $\mathscr{C}_{\lambda}$ (Danvy & Filinski 1989). The composition of the two CPS transforms gives us $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$, a double CPS transform. There is no change to the clauses inherited from $\mathscr{C}_{\lambda\mu}$ since they were already in full CPS form. The only difference is in the translation of $\widehat{\mathsf{tp}}$ as shown in Figure 18. 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 value and meta-continuation as parameters. Here the initial values for the continuation, k_ι, and meta-continuation, γ_ι, are:

$$k_\iota = \lambda x.\lambda\gamma.\gamma\,x \qquad\qquad \gamma_\iota = \lambda x.x$$

$$\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![[q]t]\!] = \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t]\!]\ \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![q]\!] \qquad \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\alpha]\!] = \alpha$$

$$\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!] = \lambda x.x$$

$$\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![V]\!] = \lambda k.k\ \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![V]\!]^V$$

$$\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t_1\ t_2]\!] = \lambda k.\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t_1]\!]\lambda f.\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t_2]\!]\lambda s.f\ s\ k$$

$$\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![x]\!]^V = x$$

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

$$\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\lambda x.t]\!]^V = \lambda x.\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t]\!]$$

$$\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\mu\widehat{\mathsf{tp}}.c]\!] = \lambda k.k\ \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!]$$

Fig. 17. Call-by-value pseudo-CPS transform of the $\lambda\mu\widehat{\mathsf{tp}}$-calculus.

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

Fig. 18. Double CPS transform of $\widehat{\mathsf{tp}}$.

So the standard way to evaluate the CPS form of a term t in this system is to apply the transform of t to k_ι and γ_ι.

Explicit initial conditions. Note that we are now in the same situation as we were with the pure λ-calculus. The CPS translation of both terms and commands takes 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 as 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 Figure 15 with meta-commands as shown in Figure 19. This gives us two special continuation labels in our syntax: $\widehat{\mathsf{tp}}$ represents k_ι in the CPS transform whereas $\circledast$ represents γ_ι. 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 as shown in Figure 19. The standard way to run a term t in this system is to evaluate 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.

$$[\circledast][\widehat{\mathsf{tp}}]t \text{ initial program} \qquad [\circledast][\widehat{\mathsf{tp}}]V \text{ final answer}$$

Similar to the λ-calculus and the $\lambda\mu$-calculus, a program of the $\lambda\mu\widehat{\mathsf{tp}}$-calculus is any meta-command, including ones containing free variables.

Operational semantics. The fact that our double CPS translation is parametrized by two continuations (the ordinary continuation and the meta-continuation) is reflected in the operational semantics for $\lambda\mu\widehat{\mathsf{tp}}$. This can be derived from defunctionalization (Reynolds 1972; Danvy 2004) of the continuation and meta-continuation used in the $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$ transform. We now have two levels of evaluation contexts: ordinary evaluation contexts from the λ-calculus, and contexts dynamically bound to $\widehat{\mathsf{tp}}$, as shown in Figure 20. 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.

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

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

Fig. 19. The $\lambda\mu\widehat{\mathsf{tp}}$-calculus and $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$ extended with meta-commands.

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

$$D^2[D[(\lambda x.t)\ V]] \mapsto D^2[D[t\{V/x\}]]$$
$$D^2[D[\mu\alpha.c]] \mapsto D^2[c\{D[t]/[\alpha]t\}]$$
$$D^2[D[\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]V]] \mapsto D^2[D[V]]$$

Fig. 20. Call-by-value evaluation contexts and operational semantics for the $\lambda\mu\widehat{\mathsf{tp}}$-calculus.

As with the $\lambda\mu$-calculus (Figure 11), the (meta-)contexts D and D^2 represent complete evaluation (meta-)contexts, which are terminated by a (meta-)co-term.

Abstract machine. We extend our previous abstract machine for $\lambda\mu$ to $\lambda\mu\widehat{\mathsf{tp}}$. Note that by including the dynamic $\widehat{\mathsf{tp}}$, our machine also gains an additional meta-context in order to match the CPS transform and operational semantics. The states of the $\lambda\mu\widehat{\mathsf{tp}}$ abstract machine are shown in Figure 21. In the machine, the meta-context is represented as a meta-stack, or *dynamic environment*, which is a sequence of bindings of $\widehat{\mathsf{tp}}$, ending in a meta-co-term q^2. The notation $F^{2*}[\widehat{\mathsf{tp}} \mapsto F^*]$ is chosen to re-enforce the intuition that $\widehat{\mathsf{tp}}$ is bound to some context, and corresponds with a top-down meta-context $D^2[D[\mu\widehat{\mathsf{tp}}.\Box]]$. The steps of the abstract machine are given in Figure 22. This machine is an extension to the abstract machine for $\lambda\mu$ from Figure 13, where the steps explicitly mentioning $\widehat{\mathsf{tp}}$ are new and the others are obtained by extending the states of the $\lambda\mu$ machine with a generic meta-stack F^{2*}.

Correctness. We have the same notion of correctness of the reduction theory with respect to the CPS transform for the $\lambda\mu\widehat{\mathsf{tp}}$-calculus as we had with the $\lambda\mu$-calculus.

Theorem 4.1 (*Soundness*)
If $M \twoheadrightarrow M'$, then $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![M]\!] =_{\beta\eta} \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![M']\!]$.

Theorem 4.2 (*Evaluation*)
If $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![c^2]\!] =_{\beta} V$, then there is a final answer c'^2 such that $c^2 \mapsto\!\!\!\!\twoheadrightarrow c'^2$ and $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![c'^2]\!] =_{\beta} V$.

A final answer of the $\lambda\mu\widehat{\mathsf{tp}}$-calculus is a meta-command of the form $[\circledast][\widehat{\mathsf{tp}}]V$ (or $D^2[[*]V]$ using the alternate initial conditions described in Section 4.1. Note that soundness now requires the use of η equivalence, which is needed to equate the transformations of programs that use $\widehat{\mathsf{tp}}$ in a trivial way to simpler programs from the transformation of the $\lambda\mu$-calculus. For further discussion and proofs of these theorems, see Section A.3 in the appendix.

$$S ::= \langle c^2 \rangle_{\mathsf{refocus}} \mid \langle c, F^{2*} \rangle_{\mathsf{refocus}} \mid \langle t, F^*, F^{2*} \rangle_{\mathsf{refocus}} \mid \langle F^*, V, F^{2*} \rangle_{\mathsf{apply}}$$
$$\mid \langle t, F^*, F^{2*} \rangle_{\mathsf{reduce}} \mid \langle F^{2*}, \widehat{\mathsf{tp}}, V \rangle_{\mathsf{reduce}} \mid \langle V \rangle_{\mathsf{done}}$$

$$F^{2*} \in \mathit{DynEnv} ::= q^2 \mid F^{2*}[\widehat{\mathsf{tp}} \mapsto F^*]$$

Fig. 21. States and evaluation meta-frames of the call-by-value $\lambda\mu\widehat{\mathsf{tp}}$ abstract machine.

$$\langle [q^2]c \rangle_{\mathsf{refocus}} \rightsquigarrow \langle c, q^2 \rangle_{\mathsf{refocus}}$$
$$\langle [q]t, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle t, q, F^{2*} \rangle_{\mathsf{refocus}}$$
$$\langle t\ t', F^*, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle t, F^*[\Box\ t'], F^{2*} \rangle_{\mathsf{refocus}}$$
$$\langle \mu\alpha.c, F^*, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle \mu\alpha.c, F^*, F^{2*} \rangle_{\mathsf{reduce}}$$
$$\langle \mu\widehat{\mathsf{tp}}.c, F^*, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle c, F^{2*}[\widehat{\mathsf{tp}} \mapsto F^*] \rangle_{\mathsf{refocus}}$$
$$\langle V, F^*, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle F^*, V, F^{2*} \rangle_{\mathsf{apply}}$$

$$\langle F^*[\Box\ t], V, F^{2*} \rangle_{\mathsf{apply}} \rightsquigarrow \langle t, F^*[V\ \Box], F^{2*} \rangle_{\mathsf{refocus}}$$
$$\langle F^*[V'\ \Box], V, F^{2*} \rangle_{\mathsf{apply}} \rightsquigarrow \langle V'\ V, F^*, F^{2*} \rangle_{\mathsf{reduce}}$$
$$\langle \widehat{\mathsf{tp}}, V, F^{2*} \rangle_{\mathsf{apply}} \rightsquigarrow \langle F^{2*}, \widehat{\mathsf{tp}}, V \rangle_{\mathsf{reduce}}$$

$$\langle (\lambda x.t)\ V, F^*, F^{2*} \rangle_{\mathsf{reduce}} \rightsquigarrow \langle t\{V/x\}, F^*, F^{2*} \rangle_{\mathsf{refocus}}$$
$$\langle \mu\alpha.c, F^*, F^{2*} \rangle_{\mathsf{reduce}} \rightsquigarrow \langle c\{F^*/\alpha\}, F^{2*} \rangle_{\mathsf{refocus}}$$
$$\langle F^{2*}[\widehat{\mathsf{tp}} \mapsto F^*], \widehat{\mathsf{tp}}, V \rangle_{\mathsf{reduce}} \rightsquigarrow \langle F^*, V, F^{2*} \rangle_{\mathsf{apply}}$$
$$\langle \circledast, \widehat{\mathsf{tp}}, V \rangle_{\mathsf{reduce}} \rightsquigarrow \langle V \rangle_{\mathsf{done}}$$

Fig. 22. Abstract machine for the call-by-value $\lambda\mu\widehat{\mathsf{tp}}$ calculus.

Expressiveness. The rebindable top-level is the additional power that allows us to encode the shift ($\mathscr{S}$) and reset (#) control operators in $\lambda\mu\widehat{\mathsf{tp}}$, using the encodings in Figure 23. These encodings resemble Filinski's (1994) encoding of $\mathscr{S}$ and # in terms of Felleisen's (1991) $\mathscr{C}$ and # operators. We can also encode a different abort operator, $\mathscr{A}^{\widehat{\mathsf{tp}}}$, which aborts up to the nearest binding of $\widehat{\mathsf{tp}}$. This operator is expressible in terms of shift alone, as shown in Figure 23. The behavior of this operator is different from the original abort operator given for the $\lambda\mu$-calculus in Figure 14, in that it does not exit the program completely, but only removes the context up to the nearest binding of $\widehat{\mathsf{tp}}$.

We can derive an operational semantics for shift and reset by their encoding into $\lambda\mu\widehat{\mathsf{tp}}$, shown in Figure 24. Since the operational semantics of $\lambda\mu\widehat{\mathsf{tp}}$ has two levels of evaluation contexts, so too does the semantics for shift and reset. The reset operator hides a binding of the dynamic $\widehat{\mathsf{tp}}$. Therefore, the evaluation meta-contexts D, which are a chain of ordinary evaluation contexts E separated by resets, correspond to the $\lambda\mu\widehat{\mathsf{tp}}$ evaluation meta-contexts E^2. The operational rules for the two control operators are given in Figure 24. This operational semantics for shift and reset corresponds to the semantics given by Biernacka *et al.* (2005) (therein referred to as the reduction semantics). The definition of

$$\#t = \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]t$$
$$\mathscr{S} = \lambda h.\mu\alpha.[\widehat{\mathsf{tp}}]h\ (\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x)$$
$$\mathscr{A}^{\widehat{\mathsf{tp}}}\ t = \mathscr{S}\ \lambda\_.t = \mu\_.[\widehat{\mathsf{tp}}]t$$

Fig. 23. Encodings of the $\mathscr{S}$ and # control operators in $\lambda\mu\widehat{\mathsf{tp}}$.

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

$$D[E[(\lambda x.t)\ V]] \mapsto D[E[t\{V/x\}]]$$
$$D[E[\mathscr{S}\ V]] \mapsto D[V\ (\lambda x.\#(E[x]))]$$
$$D[E[\#V]] \mapsto D[E[V]]$$

Fig. 24. Call-by-value evaluation contexts and operational semantics for the $\mathscr{S}$ and # operators.

two-tiered evaluation contexts is shared. The largest difference between the two semantics is that here the context captured by the shift operator is represented as a λ-abstraction, whereas in Biernacka *et al.* (2005) the context is a first-class object distinct from the class of ordinary functions. Therefore, ordinary β_v reduction for functions subsumes the step that plugs a value into a captured continuation object. Otherwise the rules in Figure 24 and in Biernacka *et al.* (2005), written $\mapsto_{BBD}$, are the same.

Theorem 4.3
$t \mapsto t'$ if and only if $t \mapsto_{BBD} t'$.

Consider how this operational semantics behaves for 'naked' shifts, that is, a shift that occurs without a surrounding reset. Since D may be empty in the second rule, a naked shift is able to capture its current context E. However, using an alternate presentation of the operational semantics by Ariola *et al.* (2009), a program with a naked shift is stuck. It has been shown by Kameyama & Hasegawa (2003) that the CPS transform for shift and reset[2] validates the axiom $\mathscr{S}(\lambda k.k\ t) = t$ in every (meta-)context. Therefore, according to the CPS transform, naked shifts of the form $\mathscr{S}(\lambda k.k\ t)$ should not get stuck, indicating that the alternate operational semantics given by Ariola *et al.* (2009) does not correspond exactly with the CPS transform for shift and reset.

4.1 Interpreting the top-level

We now understand the behavior of naked shifts that occur without a surrounding reset. Let's look more in depth at how this behavior is simulated in the $\lambda\mu\widehat{\mathsf{tp}}$ calculus.

[2] The encodings of shift and reset given here result in the same CPS transform for shift and reset as given by Danvy & Filinski (1990).

Example 3
Consider the evaluation of the $\lambda\mu\widehat{\mathsf{tp}}$ representation of $\mathscr{S}(\lambda\_.9)$ in the default initial configuration

$$[\circledast][\widehat{\mathsf{tp}}]\mu\alpha.[\widehat{\mathsf{tp}}]((\lambda\_.9)\ (\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x)) \twoheadrightarrow [\circledast][\widehat{\mathsf{tp}}]9$$

The question is, what does the final result of $[\circledast][\widehat{\mathsf{tp}}]9$ actually mean? Does it successfully produce 9 as an answer, or is it stuck because the continuation $\widehat{\mathsf{tp}}$ is not bound?

If we look at the CPS transform of the final result, we see which interpretation is correct,

$$\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\circledast][\widehat{\mathsf{tp}}]9]\!] = \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!]\ 9\ \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\circledast]\!] = \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\circledast]\!]\ 9 = 9$$

So the final meta-command is not stuck after all, but 9 is successfully returned as the answer.

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.

The choice to have the initial program state begin with $\widehat{\mathsf{tp}}$ already bound can be problematic in some situations. When we extend the language with arbitrarily many dynamic co-variables in Section 6, it is less clear what it means to begin execution with an arbitrary number of dynamic bindings. In addition, if delimited control is defined as an SML module, as in Ariola *et al.* (2011), which does not have access to the true top-level continuation, it may not be possible to implement this choice of initial conditions when the module is first instantiated. For these reasons, we consider an alternate choice of initial conditions for the top-level in which $\widehat{\mathsf{tp}}$ is unbound.

Alternative initial conditions. What does it mean for $\widehat{\mathsf{tp}}$ to be unbound? Operationally, when $\widehat{\mathsf{tp}}$ is not bound to a context, attempting to look up $\widehat{\mathsf{tp}}$ in the command $[\widehat{\mathsf{tp}}]V$ causes the program to get stuck. To replicate this behavior in the CPS transform, we can represent the empty meta-context as a free variable, γ_0. That way, when there is an attempted lookup of $\widehat{\mathsf{tp}}$ (by applying γ_0 to a value), the CPS transformed program gets stuck.

Now we run into a different problem. If we provide k_ι and γ_0 as the initial continuation and meta-continuation to our CPS transformed program, respectively, then there is no way for evaluation to terminate normally with an answer instead of getting stuck. Since k_ι and every other closed continuation eventually finish by providing a value to the meta-continuation, and since every meta-continuation eventually ends with the free variable γ_0, then there is no way to end with a final value. What we need is an initial continuation, k_0, that ignores its meta-continuation, and terminates evaluation with a value as the final result. This gives us an alternate set of initial conditions of the continuation and meta-continuation for CPS transformed terms:

$$k_0 = \lambda x.\lambda\gamma.x \qquad\qquad \gamma_0 \text{ free}$$

We can now specify this alternate choice of initial conditions in the syntax of our language as a pair of constants: one for the initial continuation and another for the initial meta-continuation as shown in Figure 25. Here, $\bullet$ represents the empty meta-context, and $*$ represents the true top-level of the program, which ends evaluation with an answer. Note

$$c^2 \in \mathit{Command}^2 ::= [q^2]c$$
$$c \in \mathit{Command} ::= [q]t$$
$$t \in \mathit{Term} ::= V \mid t_1\ t_2 \mid \mu\alpha.c \mid \mu\widehat{\mathsf{tp}}.c \qquad q^2 \in \mathit{CoTerm}^2 ::= \bullet$$
$$V \in \mathit{Value} ::= x \mid \lambda x.t \qquad q \in \mathit{CoTerm} ::= \alpha \mid \widehat{\mathsf{tp}} \mid *$$

Fig. 25. The syntax of the $\lambda\mu\widehat{\mathsf{tp}}$-calculus with empty initial conditions.

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

Fig. 26. Call-by-value CPS transform of the empty initial conditions for $\lambda\mu\widehat{\mathsf{tp}}$.

that we now have both notions of abort as defined in Figures 14 and 23. $\mathscr{A}^{\widehat{\mathsf{tp}}}$ removes the context up to the nearest binding of $\widehat{\mathsf{tp}}$, whereas $\mathscr{A}$ removes the context of the entire rest of the program. Therefore, when we find the command $[*]V$ in the eye of an evaluation meta-context D^2, we know that V must be the final answer since the rest of D^2 is irrelevant,

$$[\bullet][*]t \text{ initial program} \qquad D^2[[*]V] \text{ final answer}$$

The $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$ transform is extended with clauses for the new top-level and meta-top-level. In Figure 26, $*$ is mapped to k_0 and $\bullet$ is mapped to γ_0. 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.

The reduction semantics of $\lambda\mu\widehat{\mathsf{tp}}$ from Figure 16 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 $\widehat{\mathsf{tp}}$ 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 4
Let's revisit the previous example using $*$ and $\bullet$ to initialize execution instead of $\widehat{\mathsf{tp}}$ and $\circledast$.

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

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

Example 5
Consider what happens when a naked shift occurs under these alternate initial conditions. When we evaluate the expression $E[\mathscr{S}\ V]$, we get:

$$[\bullet][*]E[\mu\alpha.[\widehat{\mathsf{tp}}]V\ (\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x)] \twoheadrightarrow [\bullet][\widehat{\mathsf{tp}}]V\ (\lambda x.\mu\widehat{\mathsf{tp}}.[*](E[x]))$$
$$\to [\bullet][\widehat{\mathsf{tp}}]V\ (\lambda x.\mu\_.[*](E[x]))$$

Usually the continuation captured by shift behaves like a function and returns a value to its caller. However, in this case the continuation is abortive, more like the style of call/cc. This shows how changing the initial conditions impacts the behavior of shift.

Let's consider the behavior of the term $\mathscr{S}(\lambda k.k\ 9)$ under the two initial conditions. First, when we evaluate $[\circledast][\widehat{\mathsf{tp}}]\mathscr{S}(\lambda k.k\ 9)$, we get:

$$\begin{aligned}[\circledast][\widehat{\mathsf{tp}}]\mu\alpha.[\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.[\alpha]9 &\to [\circledast][\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]9\\ &\to [\circledast][\widehat{\mathsf{tp}}]9\end{aligned}$$

which returns the answer 9. Instead, when we evaluate $[\bullet][*]\mathscr{S}(\lambda k.k\ 9)$, we get:

$$\begin{aligned}[\bullet][*]\mu\alpha.[\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.[\alpha]9 &\to [\bullet][\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.[*]9\\ &\to [\bullet][\widehat{\mathsf{tp}}]\mu\_.[*]9\\ &\to [\bullet][*]9\end{aligned}$$

which returns the answer 9 after aborting its meta-context.

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 4.4
$\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\bullet][*]\mu\widehat{\mathsf{tp}}.c]\!] \twoheadrightarrow \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\circledast]c]\!]$
Proof

$$\begin{aligned}\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\bullet][*]\mu\widehat{\mathsf{tp}}.c]\!] &= \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\mu\widehat{\mathsf{tp}}.c]\!]\ \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![*]\!]\ \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\bullet]\!]\\ &\twoheadrightarrow \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!]\lambda x.\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![*]\!]\ x\ \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\bullet]\!]\\ &\twoheadrightarrow \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!]\lambda x.x\\ &= \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!]\ \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![\circledast]\!]\\ &= \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\circledast]c]\!]\end{aligned}$$

□

In order to run the abstract machine from Figure 22 with these alternate initial conditions, we need an additional step that finishes the computation when we encounter the co-constant $*$:

$$\langle *, V, F^{2*}\rangle_{\mathsf{apply}} \rightsquigarrow \langle V\rangle_{\mathsf{done}}$$

Note that in this step, the meta-stack F^{2*} may be anything, including delayed evaluation stacks that are bound by $\widehat{\mathsf{tp}}$, and will be discarded since V is the final result of the program. Additionally, we also say that the new machine state $\langle \bullet, \widehat{\mathsf{tp}}, V\rangle_{\mathsf{reduce}}$ is stuck, and cannot produce a final result.

5 Intermediate language of dynamic binding: $\lambda\widehat{\mathsf{tp}}$

Ariola *et al.* (2009) 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

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

Fig. 27. The syntax of the λ-calculus with one dynamic variable $\widehat{\mathsf{tp}}$.

with pairs. In order to better understand the dynamic nature of the prompt binding, we investigate an alternative decomposition which uses dynamic binding as a tool for understanding languages with delimited control. We start by translating away the control effects from $\lambda\mu\widehat{\mathsf{tp}}$ ($\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}$), leaving behind the dynamic binding of $\widehat{\mathsf{tp}}$. We then translate away the dynamic binding by first adopting a typical environment passing translation. 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 ($\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}$).

It is not surprising to find such a close connection between dynamic binding and delimited control. Kiselyov *et al.* (2006) showed that dynamic binding can be encoded in terms of delimited control with multiple prompts. In a way, they exposed the inherent dynamic binding that naturally occurs in delimited control. However, we are taking the opposite approach: building delimited control on top of a suitable definition of dynamic binding. An advantage of taking this course is that dynamic binding is a much *weaker* effect; it can express far fewer programs than expressed by delimited control. This makes dynamic binding easier to understand in isolation of any control operators, and we can use this understanding as a foothold for more complicated effects.

5.1 Translating control

We start with a CPS transform from $\lambda\mu\widehat{\mathsf{tp}}$ to an intermediate language with one dynamic variable $\widehat{\mathsf{tp}}$, with the syntax given in Figure 27. We introduce the concept of an explicit dynamic environment, $\bullet$, and closure under an environment, $[e]t$, to correspond with meta-co-terms and meta-commands in the $\lambda\mu\widehat{\mathsf{tp}}$-calculus, respectively. Intuitively, the environment $\bullet$ signifies the empty set of dynamic bindings, and provides an explicit notation in the language for the initial conditions for the program. To emphasize the dynamic behavior of $\widehat{\mathsf{tp}}$, we also use a 'dynamic let' binding, **dlet**, in the style of Moreau (1998). The dynamic let can be understood as syntactic sugar in terms of the dynamic function, $\lambda\widehat{\mathsf{tp}}.t$, much in the same way that static let bindings are defined in terms of static functions:

$$(\mathbf{dlet}\,\widehat{\mathsf{tp}} = t\,\mathbf{in}\,u) = (\lambda\widehat{\mathsf{tp}}.u)\ t$$

The $\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}$ transform shown in Figure 28 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 $\widehat{\mathsf{tp}}$. Note that the presence of dynamic variables in the target language lets us give a uniform translation of μ-abstractions, where $\mu\alpha.c$ and $\mu\widehat{\mathsf{tp}}.c$ introduce a static and dynamic variable in the CPS program, respectively.

Note that $\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!]$ is η-expanded. Otherwise in the translation of $[\widehat{\mathsf{tp}}]\mu\alpha.c$ one would obtain $(\lambda\alpha.\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!])\ \widehat{\mathsf{tp}}$. Since $\widehat{\mathsf{tp}}$ is not a value, the dynamic binding would be looked up

$$\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![[q^2]c]\!] = [\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![q^2]\!]]\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!] \qquad \widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\alpha]\!] = \alpha$$

$$\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![[q]t]\!] = \widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t]\!]\ \widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![q]\!] \qquad \widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}]\!] = \lambda x.\widehat{\mathsf{tp}}\ x$$

$$\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![*]\!] = \lambda x.x$$

$$\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![V]\!] = \lambda k.k\ \widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![V]\!]^V \qquad \widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\bullet]\!] = \bullet$$

$$\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t_1\ t_2]\!] = \lambda k.\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t_1]\!]\lambda f.\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t_2]\!]\lambda s.f\ s\ k \qquad \widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![x]\!]^V = x$$

$$\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\mu\tilde{\alpha}.c]\!] = \lambda k.(\lambda\tilde{\alpha}.\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![c]\!])\ k \qquad \widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![\lambda x.t]\!]^V = \lambda x.\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![t]\!]$$

Fig. 28. Call-by-value CPS transform of $\lambda\mu\widehat{\mathsf{tp}}$ using one dynamic variable.

when α is defined, instead of when it is called. To better understand the reason, consider the following example.

Example 6
In $[*]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mu\alpha.[\alpha]((\mu\widehat{\mathsf{tp}}.[\alpha]I)\ z)$, notice that α is invoked with a value under a rebinding of $\widehat{\mathsf{tp}}$. After replacing α with $\widehat{\mathsf{tp}}$, the $\widehat{\mathsf{tp}}$ is captured by the more recent binding, as shown by the reduction:

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

In terms of the CPS transformed program, we have the following reduction:

$$\begin{aligned}
&\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![[*]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mu\alpha.[\alpha]((\mu\widehat{\mathsf{tp}}.[\alpha]I)\ z)]\!] \\
&= \mathbf{dlet}\,\widehat{\mathsf{tp}} = (\lambda x.x)\,\mathbf{in}\,\mathbf{let}\,\alpha = (\lambda y.\widehat{\mathsf{tp}}\ y)\,\mathbf{in}\,\mathbf{dlet}\,\widehat{\mathsf{tp}} = (\lambda f.f\ z\ \alpha)\,\mathbf{in}\,\alpha\ \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![I]\!]^V \\
&\twoheadrightarrow \mathbf{dlet}\,\widehat{\mathsf{tp}} = (\lambda x.x)\,\mathbf{in}\,\mathbf{dlet}\,\widehat{\mathsf{tp}} = (\lambda f.f\ z\ (\lambda y.\widehat{\mathsf{tp}}\ y))\,\mathbf{in}\,\widehat{\mathsf{tp}}\ \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}[\![I]\!]^V
\end{aligned}$$

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

5.2 Simple dynamic binding

For a first attempt at defining the dynamic binding of $\widehat{\mathsf{tp}}$, we try a simple environment-passing style transform, $\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}$, where the environment is just the value currently bound to $\widehat{\mathsf{tp}}$. In the case that $\widehat{\mathsf{tp}}$ is not bound, as in the initial environment $\bullet$, we use the free variable γ_0. That is, we have $\mathscr{D}[\![\bullet]\!] = \gamma_0$. The full transform is given in Figure 29. This transform is equivalent to a simplified version of Moreau's (1998) calculus of dynamic binding with only one dynamic variable.

We would now like to give a transform of the $\lambda\mu\widehat{\mathsf{tp}}$-calculus in terms of $\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}$ and $\mathscr{D}$, giving us the composed transform $\mathscr{D}\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}$. Unfortunately, this definition of dynamic binding does not properly capture the meaning of the re-bindable top-level since it creates vicious cycles, as shown in the reduction of $\mathscr{D}\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]x]\!]\gamma$:

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

$$\mathscr{D}[\![[e]t]\!] = \mathscr{D}[\![t]\!]\ \mathscr{D}[\![e]\!] \qquad \mathscr{D}[\![\bullet]\!] = \gamma_0$$

$$\begin{aligned} \mathscr{D}[\![V]\!] &= \lambda\gamma.\mathscr{D}[\![V]\!]^V & \mathscr{D}[\![x]\!]^V &= x \\ \mathscr{D}[\![\widehat{\mathsf{tp}}]\!] &= \lambda\gamma.\gamma & \mathscr{D}[\![\lambda x.t]\!]^V &= \lambda x.\mathscr{D}[\![t]\!] \\ \mathscr{D}[\![t_1\ t_2]\!] &= \lambda\gamma.(\mathscr{D}[\![t_1]\!]\gamma)\ (\mathscr{D}[\![t_2]\!]\gamma)\ \gamma & \mathscr{D}[\![\lambda\widehat{\mathsf{tp}}.t]\!]^V &= \lambda v.\lambda\gamma.\mathscr{D}[\![t]\!]v \end{aligned}$$

Fig. 29. Simple environment-passing style transform of the λ-calculus with one dynamic variable.

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

Fig. 30. The restricted syntax of the $\lambda\widehat{\mathsf{tp}}$-calculus with one dynamic variable.

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 (1998) 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
Additionally, we can translate the dynamic let binding as:

$$\mathscr{D}[\![\mathbf{dlet}\,\widehat{\mathsf{tp}} = t\,\mathbf{in}\,u]\!] = \lambda\gamma.\mathscr{D}[\![u]\!]\ (\mathscr{D}[\![t]\!]\gamma)$$

Note that dynamic functions and dynamic let bindings are equivalent to one another.

$$\begin{aligned} \mathscr{D}[\![\lambda\widehat{\mathsf{tp}}.t]\!] &= \mathscr{D}[\![\lambda v.\,\mathbf{dlet}\,\widehat{\mathsf{tp}} = v\,\mathbf{in}\,t]\!] \\ \mathscr{D}[\![\mathbf{dlet}\,\widehat{\mathsf{tp}} = t\,\mathbf{in}\,u]\!] &= \mathscr{D}[\![(\lambda\widehat{\mathsf{tp}}.u)\ t]\!] \end{aligned}$$

5.3 Backtracking the environment

We see vicious cycles arise because dynamic binding allows for self-reference. In order to evaluate the application $\widehat{\mathsf{tp}}\ V$, we (1) look up 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 is not 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 the dynamic language, as shown in Figure 30, so that $\widehat{\mathsf{tp}}$ can only be used if it is immediately being applied to something, giving us $\lambda\widehat{\mathsf{tp}}$. We then modify the environment-passing style transform in Figure 29 to match the restricted grammar. In particular, we change the dynamic binding and application of $\widehat{\mathsf{tp}}$ to backtrack to a previous environment as shown in Figure 31. Here the notation $\gamma[\widehat{\mathsf{tp}} \mapsto v]$ means that γ, as representation of the dynamic environment, is extended so that $\widehat{\mathsf{tp}}$ is bound to v, and $\gamma(\widehat{\mathsf{tp}})$ signifies looking up the most recent binding of $\widehat{\mathsf{tp}}$ in γ. In the environment-passing style transform, the environment is represented

$$\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}[\![\lambda\widehat{\mathsf{tp}}.t]\!]^V = \lambda v.\lambda\gamma.\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}[\![t]\!]\ (\gamma[\widehat{\mathsf{tp}} \mapsto v]) \qquad \gamma[\widehat{\mathsf{tp}} \mapsto v] = \lambda x.v\ x\ \gamma$$
$$\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}\ t]\!] = \lambda\gamma.\gamma(\widehat{\mathsf{tp}})\ (\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}[\![t]\!]\gamma) \qquad \gamma(\widehat{\mathsf{tp}}) = \gamma$$

Fig. 31. Backtracking environment-passing style transform of the $\lambda\widehat{\mathsf{tp}}$-calculus.

abstractly as a function that, when invoked, will access the most recent binding of $\widehat{\mathsf{tp}}$. Therefore, looking up the binding of $\widehat{\mathsf{tp}}$ in an environment and applying the value to an argument is defined as just applying the environment to the argument. To extend an existing environment with a new binding of a value v to $\widehat{\mathsf{tp}}$, we create a function that, when applied, will call the function v to the argument in the previous environment. 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 $\widehat{\mathsf{tp}}\ x$ in Figure 31, $\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}[\![\widehat{\mathsf{tp}}\ x]\!]\gamma = \gamma\ x$, with the original translation from Figure 29, $\mathscr{D}[\![\widehat{\mathsf{tp}}\ x]\!]\gamma = \gamma\ x\ \gamma$. Note 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, the composed transform $\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}$ no longer creates the same cycle as before in the reduction of $\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![[\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]x]\!]\gamma$:

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

When we compose the two phases together, we get the derived translation $\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}$, which is the same as our original translation $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$ from Figure 18, up to $\beta\eta$-equivalence in the λ-calculus.

Theorem 5.1
$\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}[\![M]\!] =_{\beta\eta} \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}[\![M]\!]$

Remark 2
Note that the definition of $*$ in $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}}$ is exactly the environment-passing style translation of the initial continuation $\lambda x.x$. The backtracking behavior that we present here is also necessary to express exceptions with dynamic variables. A similar encoding was given by Moreau (1998) using an abort operator to reinstall the right environment.

6 Delimited control with multiple prompts: $\lambda\widehat{\mu}$

We want to extend $\lambda\mu\widehat{\mathsf{tp}}$ with multiple dynamic co-variables, written by convention with a hat like $\widehat{\alpha}$, so that binding $\widehat{\alpha}$ does not interfere with $\widehat{\beta}$ and *vice versa*. This is different from the nested definition of resets in the CPS hierarchy by Danvy & Filinski (1990). 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.

Syntax. The language of control with multiple prompts, $\lambda\widehat{\mu}$ shown in Figure 32, is a simple extension of $\lambda\mu\widehat{\mathsf{tp}}$ with multiple dynamic top-level binders.

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

Fig. 32. The syntax of the $\lambda\widehat{\mu}$-calculus.

$$
\begin{array}{ll}
\mu\widehat{\alpha}.[\widehat{\alpha}]V \to V & \\
\mu\widehat{\alpha}.[\widehat{\beta}]V \to \mu\_.[\widehat{\beta}]V & \textbf{where}\ \widehat{\alpha} \neq \widehat{\beta} \\
\mu\widehat{\alpha}.[*]V \to \mu\_.[*]V &
\end{array}
$$

Fig. 33. Call-by-value reduction theory for dynamic co-variables in the $\lambda\widehat{\mu}$-calculus.

Reduction. The reduction rules for multiple prompts are a generalization of the reduction rules for single prompt $\widehat{\mathsf{tp}}$. The reduction theory for the $\lambda\widehat{\mu}$-calculus is based on the $\lambda\mu$-calculus theory given in Figure 10 and extended with the additional rules for dynamic co-variables given in Figure 33. Just like how invocation of $*$ throws away the dynamic environment, invocation of $\widehat{\beta}$ will throw away portions of its dynamic environment until the correct binding is found. Then the usual reduction of $\mu\widehat{\alpha}.[\widehat{\alpha}]V$ is available to plug V into the correct context.

Operational semantics. To define the operational semantics for $\lambda\widehat{\mu}$, we extend the evaluation meta-contexts from Figure 20 to allow for bindings of many dynamic co-variables. The set of evaluation (meta-)contexts and operational rules for $\lambda\widehat{\mu}$ is given in Figure 34. Note that in the third rule, the context $E^2_{\widehat{\alpha}}$ does not bind the dynamic co-variable $\widehat{\alpha}$, and is defined as:

$$
E^2_{\widehat{\alpha}} ::= \Box \mid D[\mu\widehat{\beta}.E^2_{\widehat{\alpha}}], \qquad \textbf{where}\ \widehat{\beta} \not\equiv \widehat{\alpha}
$$

CPS transform. The intermediate language of dynamic binding, $\widehat{\lambda}$, is extended with the syntax shown in Figure 35. The definition of $\widehat{\lambda}$ uses the same backtracking environment-passing style translation as $\lambda\widehat{\mathsf{tp}}$ from Figure 31. The only thing that needs to change from $\lambda\widehat{\mathsf{tp}}$ to $\widehat{\lambda}$ 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. The translation of dynamic environments with bindings of more than one dynamic variable is given in Figure 36. As in Figure 31, $\gamma(\widehat{x})$ and $\gamma[\widehat{x} \mapsto v]$ are notations for looking up bindings and extending the environment γ, which is represented in a functionalized form. Additionally, the quotation brackets, $\ulcorner\widehat{x}\urcorner$, reify the dynamic variables of the source language as values in the target language. The details of how dynamic co-variables of $\lambda\widehat{\mu}$ are mapped to values in the λ-calculus are not critical to the transform, except that they must all be distinct and have decidable equality, written as $p \equiv q$.

The CPS transform of $\lambda\widehat{\mu}$ is just the composed transform $\mathscr{D}_{\widehat{\lambda}}\widehat{\mathscr{C}}_{\lambda\widehat{\mu}}$, where $\widehat{\mathscr{C}}_{\lambda\widehat{\mu}}$ is the same as $\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}}$ from Figure 28 except that multiple dynamic variables are used by $\widehat{\mathscr{C}}_{\lambda\widehat{\mu}}$,

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

$$D^2[D[(\lambda x.t)\ V]] \mapsto D^2[D[t\{V/x\}]]$$
$$D^2[D[\mu\alpha.c]] \mapsto D^2[c\{D[t]/[\alpha]t\}]$$
$$D^2[D[\mu\widehat{\alpha}.E^2_{\widehat{\alpha}}[[\widehat{\alpha}]V]]] \mapsto D^2[D[V]]$$

Fig. 34. Call-by-value evaluation contexts and operational semantics of the $\lambda\widehat{\mu}$-calculus.

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

Fig. 35. The syntax of the $\widehat{\lambda}$-calculus.

with one unique dynamic variable for each different dynamic co-variable in the source program.

Abstract machine. We extend the abstract machine for $\lambda\mu\widehat{\mathsf{tp}}$ to include multiple dynamic co-variables. The states of the $\lambda\widehat{\mu}$ abstract machine are given in Figure 37 and the steps are given in Figure 38.

Correctness. As before, we have the same notion of correctness of the reduction theory with respect to the CPS transform for $\lambda\widehat{\mu}$ as we had with $\lambda\mu$ and $\lambda\mu\widehat{\mathsf{tp}}$.

Theorem 6.1 (*Soundness*)
If $M \twoheadrightarrow M'$, then $\mathscr{D}_{\widehat{\lambda}}\widehat{\mathscr{C}}_{\lambda\widehat{\mu}}[\![M]\!] =_{\beta\eta} \mathscr{D}_{\widehat{\lambda}}\widehat{\mathscr{C}}_{\lambda\widehat{\mu}}[\![M']\!]$.

Theorem 6.2 (*Evaluation*)
If $\mathscr{D}_{\widehat{\lambda}}\widehat{\mathscr{C}}_{\lambda\widehat{\mu}}[\![c^2]\!] =_{\beta} V$, then there is a final answer c'^2 such that $c^2 \twoheadrightarrow c'^2$ and $\mathscr{D}_{\widehat{\lambda}}\widehat{\mathscr{C}}_{\lambda\widehat{\mu}}[\![c'^2]\!] =_{\beta} V$.

A final answer of the $\lambda\widehat{\mu}$-calculus is a meta-command of the form $D^2[[*]V]$. For further discussion and proofs of these theorems, see Section A.4 in the appendix.

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 ($\mathscr{A}^{\widehat{\alpha}}$) operators in Figure 39.

7 Delimited control with a dynamic prompt: $\lambda\mu\widehat{\mathsf{tp}}_0$

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 ($\mathscr{S}_0$) (Materzok & Biernacki 2011). 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.

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

Fig. 36. The translation of environments in $\widehat{\lambda}$ with multiple dynamic variables.

$$\begin{aligned} S ::= {} & \langle c^2 \rangle_{\mathsf{refocus}} \mid \langle c, F^{2*} \rangle_{\mathsf{refocus}} \mid \langle t, F^*, F^{2*} \rangle_{\mathsf{refocus}} \\ & \mid \langle F^*, V, F^{2*} \rangle_{\mathsf{apply}} \mid \langle t, F^*, F^{2*} \rangle_{\mathsf{reduce}} \mid \langle F^{2*}, \widehat{\alpha}, V \rangle_{\mathsf{reduce}} \mid \langle F^{2*}, \widehat{\alpha}, V \rangle_{\mathsf{lookup}} \end{aligned}$$

Fig. 37. States of the call-by-value $\lambda\widehat{\mu}$ machine.

Example 7
Consider the behavior of the following program which uses the shift operator twice in a row:

$$\#(1 + (\#(\mathscr{S}\lambda k.\mathscr{S}\lambda q.2))) \twoheadrightarrow \#(1 + (\#(\mathscr{S}\lambda q.2))) \twoheadrightarrow \#(1 + (\#2)) \twoheadrightarrow 3$$

Each time the shift is reduced, its body is evaluated underneath the innermost reset. Instead, consider what happens when we replace shift and reset with shift_0 and reset_0.

$$\#_0(1 + (\#_0(\mathscr{S}_0\lambda k.\mathscr{S}_0\lambda q.2))) \twoheadrightarrow \#_0(1 + (\mathscr{S}_0\lambda q.2)) \twoheadrightarrow 2$$

Note how after the first shift_0 is reduced, its body is evaluated outside the innermost reset_0, exposing the context that was previously hidden by the reset_0.

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 their surrounding context. To that end, we introduce a new form of command, $[\widehat{\mathsf{tp}}]_0 t$, in which the dynamic co-term $\widehat{\mathsf{tp}}$ has priority over the term t, rather than the other way around. In a sense, this is a 'non-strict' variant of the command $[\widehat{\mathsf{tp}}]t$, where invoking the dynamic top-level happens without first evaluating the term t, as is the case in the expression $\mathscr{S}_0\lambda k.t$. In order to represent the semantics of reset_0, we need to impose the usual order of evaluation, where the underlying term is first reduced to a value before clearing the prompt. As we will see, we can regain this behavior due to the sequentializing behavior of strict let-bindings in the language.

Syntax. The syntax of $\lambda\mu\widehat{\mathsf{tp}}_0$ is given in Figure 40. The new command $[\widehat{\mathsf{tp}}]_0 t$ represents lifting the unevaluated term t through the most recent binding of $\widehat{\mathsf{tp}}$ and embedding the term in that context. Correspondingly, the binding of $\widehat{\mathsf{tp}}$, $\mu_0\widehat{\mathsf{tp}}.c$, has been given a different notation to signify that it is waiting for an unevaluated term, rather than a value, to be returned by its underlying command.

Reduction. Reduction of the new command is similar to $[\widehat{\mathsf{tp}}]t$ from $\lambda\mu\widehat{\mathsf{tp}}$, but with different priorities between the continuation and the term. In the $\lambda\mu\widehat{\mathsf{tp}}$-calculus term $\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]t$, $\widehat{\mathsf{tp}}$ is reduced only when t is a value. The opposite occurs with $\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]_0 t$, where $\widehat{\mathsf{tp}}$ is reduced immediately without considering t, as shown in Figure 41. Note that due to the reversed priorities in the command $[\widehat{\mathsf{tp}}]_0 t$, the usual renaming rule for μ does not apply to

$$\begin{aligned}
\langle [q^2]c\rangle_{\mathsf{refocus}} &\rightsquigarrow \langle c, q^2\rangle_{\mathsf{refocus}} \\
\langle [q]t, F^{2*}\rangle_{\mathsf{refocus}} &\rightsquigarrow \langle t, q, F^{2*}\rangle_{\mathsf{refocus}} \\
\langle t\ t', F^*, F^{2*}\rangle_{\mathsf{refocus}} &\rightsquigarrow \langle t, F^*[\Box\ t'], F^{2*}\rangle_{\mathsf{refocus}} \\
\langle \mu\alpha.c, F^*, F^{2*}\rangle_{\mathsf{refocus}} &\rightsquigarrow \langle \mu\alpha.c, F^*, F^{2*}\rangle_{\mathsf{reduce}} \\
\langle \mu\widehat{\alpha}.c, F^*, F^{2*}\rangle_{\mathsf{refocus}} &\rightsquigarrow \langle c, F^{2*}[\widehat{\alpha} \mapsto F^*]\rangle_{\mathsf{refocus}} \\
\langle V, F^*, F^{2*}\rangle_{\mathsf{refocus}} &\rightsquigarrow \langle F^*, V, F^{2*}\rangle_{\mathsf{apply}}
\end{aligned}$$

$$\begin{aligned}
\langle F^*[\Box\ t], V, F^{2*}\rangle_{\mathsf{apply}} &\rightsquigarrow \langle t, F^*[V\ \Box], F^{2*}\rangle_{\mathsf{refocus}} \\
\langle F^*[V'\ \Box], V, F^{2*}\rangle_{\mathsf{apply}} &\rightsquigarrow \langle V'\ V, F^*, F^{2*}\rangle_{\mathsf{reduce}} \\
\langle \widehat{\alpha}, V, F^{2*}\rangle_{\mathsf{apply}} &\rightsquigarrow \langle F^{2*}, \widehat{\alpha}, V\rangle_{\mathsf{lookup}} \\
\langle *, V, F^{2*}\rangle_{\mathsf{apply}} &\rightsquigarrow \langle V\rangle_{\mathsf{done}}
\end{aligned}$$

$$\begin{aligned}
\langle (\lambda x.t)\ V, F^*, F^{2*}\rangle_{\mathsf{reduce}} &\rightsquigarrow \langle t\{V/x\}, F^*, F^{2*}\rangle_{\mathsf{refocus}} \\
\langle \mu\alpha.c, F^*, F^{2*}\rangle_{\mathsf{reduce}} &\rightsquigarrow \langle c\{F^*/\alpha\}, F^{2*}\rangle_{\mathsf{refocus}} \\
\langle F^{2*}[\widehat{\alpha} \mapsto F^*], \widehat{\alpha}, V\rangle_{\mathsf{reduce}} &\rightsquigarrow \langle V, F^*, F^{2*}\rangle_{\mathsf{apply}}
\end{aligned}$$

$$\begin{aligned}
\langle F^{2*}[\widehat{\alpha} \mapsto F^*], \widehat{\alpha}, V\rangle_{\mathsf{lookup}} &\rightsquigarrow \langle F^{2*}[\widehat{\alpha} \mapsto F^*], \widehat{\alpha}, V\rangle_{\mathsf{reduce}} \\
\langle F^{2*}[\widehat{\beta} \mapsto F^*], \widehat{\alpha}, V\rangle_{\mathsf{lookup}} &\rightsquigarrow \langle F^{2*}, \widehat{\alpha}, V\rangle_{\mathsf{lookup}} \qquad \textbf{where } \widehat{\alpha} \neq \widehat{\beta}
\end{aligned}$$

Fig. 38. Abstract machine for the call-by-value $\lambda\widehat{\mu}$ calculus.

$$\begin{aligned}
\#^{\widehat{\alpha}}t &= \mu\widehat{\alpha}.[\widehat{\alpha}]t \\
\mathscr{A}^{\widehat{\alpha}}t &= \mu_.[\widehat{\alpha}]t
\end{aligned}$$

Fig. 39. The multi-prompt $\#^{\widehat{\alpha}}$ and $\mathscr{A}^{\widehat{\alpha}}$ control operators.

this use of $\widehat{\mathsf{tp}}$:

$$[\widehat{\mathsf{tp}}]_0\mu\alpha.c \not\to c\{\widehat{\mathsf{tp}}/\alpha\}$$

Operational semantics. The operational semantics of $\lambda\mu\widehat{\mathsf{tp}}_0$ is a slight variation of the one for $\lambda\mu\widehat{\mathsf{tp}}$. As before, two levels of evaluation contexts are used, where the second level meta-evaluation context is a sequence of complete evaluation contexts D separated by dynamic μ_0-abstractions of $\widehat{\mathsf{tp}}$, as shown in Figure 42. Note that this definition of complete evaluation contexts, D, in $\lambda\mu\widehat{\mathsf{tp}}_0$ differs from the complete evaluation contexts in $\lambda\mu\widehat{\mathsf{tp}}$. In particular, whereas $[\widehat{\mathsf{tp}}]\Box$ is an evaluation context in $\lambda\mu\widehat{\mathsf{tp}}$, the variant $[\widehat{\mathsf{tp}}]_0\Box$ is *not* an evaluation context in $\lambda\mu\widehat{\mathsf{tp}}_0$. This expresses the fact that in the command $[\widehat{\mathsf{tp}}]_0 t$, looking up the binding for $\widehat{\mathsf{tp}}$ has priority over evaluating the term t. With this one difference in mind, the operational rules for $\lambda\mu\widehat{\mathsf{tp}}_0$ are given in Figure 42.

$$c^2 \in \mathit{Command}^2 ::= [q^2]c$$
$$c \in \mathit{Command} ::= [q]t \mid [\widehat{\mathsf{tp}}]_0 t$$
$$t \in \mathit{Term} ::= V \mid t_1\ t_2 \mid \mu\alpha.c \mid \mu_0\widehat{\mathsf{tp}}.c \qquad q^2 \in \mathit{CoTerm}^2 ::= \bullet$$
$$V \in \mathit{Value} ::= x \mid \lambda x.t \qquad q \in \mathit{CoTerm} ::= \alpha \mid *$$

Fig. 40. The syntax of the $\lambda\mu\widehat{\mathsf{tp}}_0$-calculus.

$$\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]_0 t \to t \qquad \mu_0\widehat{\mathsf{tp}}.[*]V \to \mu\_.[*]V$$

Fig. 41. Call-by-value reduction theory for the dynamic $\widehat{\mathsf{tp}}$ in the $\lambda\mu\widehat{\mathsf{tp}}_0$-calculus.

CPS transform. We define a CPS for $\lambda\mu\widehat{\mathsf{tp}}_0$ in the style of Materzok & Biernacki's (2011) definition of shift_0 as shown in Figure 43. This is an extension of the basic $\mathscr{C}_{\lambda\mu}$ transform from Figure 12. Note how the translation of $\mu_0\widehat{\mathsf{tp}}.c$ in $\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}_0}$ is different from the translation of $\mu\widehat{\mathsf{tp}}.c$ in $\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}}$. Rather than running the command to completion and then passing the result to the continuation k, we pass k as an extra continuation to the command c. If the command c has the form $[q]t$, we end up evaluating t with an extra continuation instead of just the one. Contrarily, in the translation of $[\widehat{\mathsf{tp}}]_0 t$, the term t is missing a continuation, thus taking one of the extra continuations as its own.

However, we have a problem with this choice of CPS transform when it comes to translating $*$. In particular, because all the continuations bound by $\mu_0\widehat{\mathsf{tp}}.c$ are given extra arguments to the root command, we cannot translate $*$ as a simple function in the λ-calculus since $*$ needs to discard an arbitrary number of arguments. Therefore, we consider an alternate CPS transform that uses a fixed number of continuations.

In general, the CPS translation in Figure 43 produces programs which have a dynamically growing and shrinking stack of continuations. To make this intuition explicit, we give an alternate CPS transform in Figure 44, where the rest of the transform is the same as $\mathscr{C}_{\lambda\mu}$ from Figure 12. This transform reifies all the extra continuations as a literal stack, similar to transforms given by Shan (2007) and Materzok & Biernacki (2011), giving us the meta-continuation stack γ. Here binding an evaluation context with $\mu_0\widehat{\mathsf{tp}}.c$ corresponds to pushing a continuation onto the meta-continuation stack, whereas accessing the most recent binding of $\widehat{\mathsf{tp}}$ with $[\widehat{\mathsf{tp}}]_0 t$ corresponds to popping the top continuation off the stack and evaluating the term t in that continuation.

As with the transform for $\lambda\mu\widehat{\mathsf{tp}}$ from Figure 28, we can express the CPS transform in terms of a language of dynamic binding as shown in Figure 45. Note that in order to accommodate the new command $[\widehat{\mathsf{tp}}]_0 t$, we need to relax the restrictions on when we can look up the dynamic $\widehat{\mathsf{tp}}$. In $\lambda\mu\widehat{\mathsf{tp}}$, we only ever need to look up $\widehat{\mathsf{tp}}$ when we are applying it to a value. However, in $\lambda\mu\widehat{\mathsf{tp}}_0$ we need to look up the continuation bound to $\widehat{\mathsf{tp}}$ before we are ready to apply the continuation to a value. As before with $\lambda\widehat{\mathsf{tp}}$, looking up the dynamic $\widehat{\mathsf{tp}}$ backtracks the dynamic environment to its previous state before $\widehat{\mathsf{tp}}$ was bound. The syntax of this dynamic language, $\lambda\widehat{\mathsf{tp}}_0$, is given in Figure 46. This is an extension of the previous dynamic language $\lambda\widehat{\mathsf{tp}}$ since we can still apply $\widehat{\mathsf{tp}}$ to a term by first explicitly

$$E^2 ::= \Box \mid D[\mu_0 \widehat{\mathsf{tp}}.E^2] \qquad D^2 ::= [q^2]E^2$$
$$E ::= \Box \mid E\ t \mid V\ E \qquad D ::= [q]E$$

$$D^2[D[(\lambda x.t)\ V]] \mapsto D^2[D[t\{V/x\}]]$$
$$D^2[D[\mu\alpha.c]] \mapsto D^2[c\{D[t]/[\alpha]t\}]$$
$$D^2[D[\mu_0 \widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]_0 t]] \mapsto D^2[D[t]]$$

Fig. 42. Call-by-value evaluation contexts and operational semantics for the $\lambda\mu\widehat{\mathsf{tp}}_0$-calculus.

$$\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![\mu_0 \widehat{\mathsf{tp}}.c]\!] = \lambda k.\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![c]\!]\ k \qquad \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![[\widehat{\mathsf{tp}}]_0 t]\!] = \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![t]\!]$$

Fig. 43. Call-by-value CPS transform of the dynamic $\widehat{\mathsf{tp}}$ in the $\lambda\mu\widehat{\mathsf{tp}}_0$-calculus.

looking up the value of $\widehat{\mathsf{tp}}$ and binding it to a variable:

$$\widehat{\mathsf{tp}}\ t = (\lambda x.(\lambda f.f\ x)\ \widehat{\mathsf{tp}})\ t$$

We can extend the environment-passing style transform $\mathscr{D}_{\lambda\widehat{\mathsf{tp}}}$ from Figure 31 for the generalized dynamic operations. If we represent the dynamic environment concretely using pairs, we get the transform given in Figure 47. Composing the $\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}$ transform from Figure 45 with the $\mathscr{D}^{\times}_{\lambda\widehat{\mathsf{tp}}_0}$ environment-passing style transform is the same as the $\mathscr{C}^{\times}_{\lambda\mu\widehat{\mathsf{tp}}_0}$ transform from Figure 44, up to $\beta\eta$-equivalence in the λ-calculus, using a concrete stack for the meta-continuation. Alternatively, we can refunctionalize the concrete pairs in the above transform, giving us an environment-passing style transform into the pure λ-calculus shown in Figure 48, where the rest of the transform is the same as in Figure 31. Binding a value to $\widehat{\mathsf{tp}}$ in a dynamic environment corresponds to constructing a function that passes the new value and the old environment to a continuation. In order to access the most recent dynamic binding, the environment is applied to a continuation that binds the value to a variable and evaluates a term in the previous environment.

The functional representation of dynamic environments gives us an alternate transform of $\lambda\mu\widehat{\mathsf{tp}}_0$, taken as the composition of the $\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}$ and $\mathscr{D}_{\lambda\widehat{\mathsf{tp}}_0}$ transforms. This transform makes it explicit that unevaluated terms are passed to the meta-continuation. Also, note that the $\mathscr{D}_{\lambda\widehat{\mathsf{tp}}_0}\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}$ transform is the same as the double CPS transform $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}$, defined as the composition of $\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}_0}$ and $\mathscr{C}_{\lambda}$ up to $\beta\eta$-equivalence in the λ-calculus. The composed transform is shown in Figure 49, where the rest of the transform is the same as $\mathscr{C}_{\lambda\mu}$ from Figure 12.

Theorem 7.1
$\mathscr{D}^{\times}_{\lambda\widehat{\mathsf{tp}}_0}\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![M]\!] =_{\beta\eta} \mathscr{C}^{\times}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![M]\!]$ and $\mathscr{D}_{\lambda\widehat{\mathsf{tp}}_0}\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![M]\!] =_{\beta\eta} \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![M]\!]$

Abstract machine. The $\lambda\mu\widehat{\mathsf{tp}}_0$ abstract machine generalizes the reduce state of the $\lambda\mu\widehat{\mathsf{tp}}$ machine from Figure 21, as shown in Figure 50. Extending the $\lambda\mu\widehat{\mathsf{tp}}$ abstract machine for $\lambda\mu\widehat{\mathsf{tp}}_0$ only requires one additional clause to interpret the new form of commands and the

$$\begin{aligned}
\mathscr{C}^{\times}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![\mu_0\widehat{\mathsf{tp}}.c]\!] &= \lambda k.\lambda\gamma.\mathscr{C}^{\times}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![c]\!]\ \langle k,\gamma\rangle \\
\mathscr{C}^{\times}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![[\widehat{\mathsf{tp}}]_0 t]\!] &= \lambda\langle k,\gamma\rangle.\mathscr{C}^{\times}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![t]\!]k\ \gamma \\
\mathscr{C}^{\times}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![*]\!] &= \lambda x.\lambda\gamma.x
\end{aligned}$$

Fig. 44. Stack-based call-by-value CPS transform of the $\lambda\mu\widehat{\mathsf{tp}}_0$-calculus.

$$\begin{aligned}
\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![\mu_0\widehat{\mathsf{tp}}.c]\!] &= \lambda k.(\lambda\widehat{\mathsf{tp}}.\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![c]\!])\ k \\
\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![[\widehat{\mathsf{tp}}]_0 t]\!] &= \widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![t]\!]\ \widehat{\mathsf{tp}} \\
\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![*]\!] &= \lambda x.x
\end{aligned}$$

Fig. 45. Call-by-value CPS transform of $\widehat{\mathsf{tp}}$ in the $\lambda\mu\widehat{\mathsf{tp}}_0$-calculus using one dynamic variable.

generalization of one reduce step. The complete set of steps for the $\lambda\mu\widehat{\mathsf{tp}}_0$ machine are given in Figure 51.

Correctness. Again, we have the same notion of correctness of the reduction theory with respect to the CPS transform for $\lambda\mu\widehat{\mathsf{tp}}_0$ as we had with $\lambda\mu$, $\lambda\mu\widehat{\mathsf{tp}}$, and $\lambda\widehat{\mu}$.

Theorem 7.2 (*Soundness*)
If $M \twoheadrightarrow M'$ then $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![M]\!] =_{\beta\eta} \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![M']\!]$.

Theorem 7.3 (*Evaluation*)
If $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![c^2]\!] =_{\beta} V$ then there is a final answer c'^2 such that $c^2 \mapsto\!\!\!\!\twoheadrightarrow c'^2$ and $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![c'^2]\!] =_{\beta} V$.

A final answer of the $\lambda\mu\widehat{\mathsf{tp}}_0$-calculus is a meta-command of the form $D^2[[*]V]$. For further discussion and proofs of these theorems, see Section A.5 in the appendix.

Expressiveness. In order to give an encoding of reset_0, we need to recover the command $[\widehat{\mathsf{tp}}]t$ which evaluates its term first before accessing $\widehat{\mathsf{tp}}$. This command can be encoded in $\lambda\mu\widehat{\mathsf{tp}}_0$ as:

$$[\widehat{\mathsf{tp}}]t = [*]\mathbf{let}\, x = t\, \mathbf{in}\, \mu\_.[\widehat{\mathsf{tp}}]_0 x$$

We can derive the following reductions in $\lambda\mu\widehat{\mathsf{tp}}_0$, including the usual renaming rule for μ:

$$\begin{aligned}
[\widehat{\mathsf{tp}}]V &\twoheadrightarrow [\widehat{\mathsf{tp}}]_0 V \\
[\widehat{\mathsf{tp}}]\mu\alpha.c &\twoheadrightarrow c\{[\widehat{\mathsf{tp}}]t/[\alpha]t\}
\end{aligned}$$

With this command, the encoding of shift_0 ($\mathscr{S}_0$) and reset_0 ($\#_0$) in $\lambda\mu\widehat{\mathsf{tp}}_0$ shown in Figure 52 mirrors the encoding from Figure 23 of shift and reset in $\lambda\mu\widehat{\mathsf{tp}}$.

We can derive the operational rules for the two control operators from the operational semantics of $\lambda\mu\widehat{\mathsf{tp}}_0$ using the encodings in Figure 52. The two-part definition of evaluation contexts mirrors Materzok & Biernacki's (2011) presentation of $\mathscr{S}_0$ using contexts and trails. The derived operational semantics for shift_0 and reset_0 are shown in Figure 53.

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

Fig. 46. The syntax of the $\lambda\widehat{\mathsf{tp}}_0$-calculus of dynamic binding.

$$\mathscr{D}^{\times}_{\lambda\widehat{\mathsf{tp}}_0}[\![\lambda\widehat{\mathsf{tp}}.t]\!]^V = \lambda v.\lambda\gamma.\mathscr{D}^{\times}_{\lambda\widehat{\mathsf{tp}}_0}[\![t]\!]\ (\gamma[\widehat{\mathsf{tp}} \mapsto v])$$
$$\mathscr{D}^{\times}_{\lambda\widehat{\mathsf{tp}}_0}[\![t\ \widehat{\mathsf{tp}}]\!] = \lambda\gamma.\mathbf{let}\ \langle x, \gamma'\rangle = \gamma(\widehat{\mathsf{tp}})\ \mathbf{in}\ (\mathscr{D}^{\times}_{\lambda\widehat{\mathsf{tp}}_0}[\![t]\!]\gamma)\ x\ \gamma'$$

$$\gamma[\widehat{\mathsf{tp}} \mapsto v] = \langle v, \gamma\rangle \qquad \gamma(\widehat{\mathsf{tp}}) = \gamma$$

Fig. 47. Environment-passing style transform of the $\lambda\widehat{\mathsf{tp}}_0$-calculus using an environment.

These rules demonstrate that the fundamental difference between shift and shift_0 is the presence or absence of the nearest delimiter, reset or reset_0, respectively, after capturing their context.

Example 8
Consider the following folklore encoding of shift and reset in terms of shift_0 and reset_0, which simply replaces the surrounding delimiter removed by shift_0:

$$\begin{aligned}
\#'t &= \#_0 t \\
\mathscr{S}' &= \lambda h.\mathscr{S}_0\ \lambda k.\#_0(h\ k) \\
&\twoheadrightarrow \lambda h.\mu\alpha.[\widehat{\mathsf{tp}}]_0\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}](h\ (\lambda x.\mu_0\widehat{\mathsf{tp}}.[\alpha]x))
\end{aligned}$$

As pointed out by Materzok & Biernacki (2011), this encoding of shift and reset allows for the usual, coarse-grained reductions:

$$\#'V \twoheadrightarrow V$$
$$\#'E[\mathscr{S}'\ V] \twoheadrightarrow \#'(V\ (\lambda x.\#'E[x]))$$

However, shift' and reset' admit a different equational theory than the original encoding of shift and reset. For example, we do not have that reset' is idempotent:

$$\#'\#'t \neq \#'t$$

As another example, we can test the following axiom from Kameyama & Hasegawa (2003) involving shift:

$$\mathscr{S}\lambda k.k\ V = V$$

In contrast with shift, in an empty meta-context, shift' gets stuck, and does not produce a final answer:

$$\begin{aligned}
[\bullet][*]\mathscr{S}'\lambda k.k\ V &\twoheadrightarrow [\bullet][*]\mu\alpha.[\widehat{\mathsf{tp}}]_0\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mu_0\widehat{\mathsf{tp}}.[\alpha]V \\
&\to [\bullet][\widehat{\mathsf{tp}}]_0\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mu_0\widehat{\mathsf{tp}}.[*]V \\
&\to [\bullet][\widehat{\mathsf{tp}}]_0\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mu_{_}.[*]V \\
&\to [\bullet][\widehat{\mathsf{tp}}]_0\mu\widehat{\mathsf{tp}}.[*]V \\
&\to [\bullet][\widehat{\mathsf{tp}}]_0\mu_{_}.[*]V
\end{aligned}$$

$$\mathscr{D}_{\lambda\widehat{\mathsf{tp}}_0}[\![\lambda\widehat{\mathsf{tp}}.t]\!]^V = \lambda v.\lambda\gamma.\mathscr{D}_{\lambda\widehat{\mathsf{tp}}_0}[\![t]\!]\ (\gamma[\widehat{\mathsf{tp}} \mapsto v])$$
$$\mathscr{D}_{\lambda\widehat{\mathsf{tp}}_0}[\![t\ \widehat{\mathsf{tp}}]\!]\gamma = \gamma(\widehat{\mathsf{tp}})\ (\mathscr{D}_{\lambda\widehat{\mathsf{tp}}_0}[\![t]\!]\gamma)$$

$$\gamma[\widehat{\mathsf{tp}} \mapsto v] = \lambda g.g\ v\ \gamma \qquad \gamma(\widehat{\mathsf{tp}}) = \gamma$$

Fig. 48. Refunctionalized environment-passing style transform of the $\lambda\widehat{\mathsf{tp}}_0$-calculus.

$$\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![\mu_0\widehat{\mathsf{tp}}.c]\!] = \lambda k.\lambda\gamma.\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![c]\!]\lambda u.u\ k\ \gamma$$
$$\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![[\widehat{\mathsf{tp}}]_0 t]\!] = \lambda\gamma.\gamma\,\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![t]\!]$$
$$\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![*]\!] = \lambda x.\lambda\gamma.x$$

Fig. 49. Call-by-value double CPS transform of the $\lambda\mu\widehat{\mathsf{tp}}_0$-calculus.

Likewise, the CPS transform of this final program state is also stuck,

$$\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![[\bullet][\widehat{\mathsf{tp}}]_0\mu\_.[*]V]\!] \twoheadrightarrow \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![\bullet]\!]\ \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![\mu\_.[*]V]\!] \twoheadrightarrow \gamma_0\ (\lambda\_.\lambda\_.V)$$

8 Delimited control with multiple dynamic prompts: $\lambda\widehat{\mu}_0$

With just the simple addition of multiple static prompts, we still don't have enough expressive power in $\lambda\widehat{\mu}$ to encode operators with dynamic prompts like shift_0 and reset_0, or similar previous control operators given by Gunter *et al.* (1995), Dybvig *et al.* (2007), and Flatt *et al.* (2007). The dilemma is that in the presence of multiple prompts, a shift_0 up to a reset_0 for $\widehat{\alpha}$ not only captures its immediate context up to the nearest reset_0 but also captures all the contexts found behind non-matching reset_0s. Then the continuation that shift_0 captures will restore the captured context as well as seamlessly inserting a partial meta-context in place. For example, we want to be able to express the following reduction:

$$\#^{\widehat{\alpha}}_0(E[\#^{\widehat{\beta}}_0(E'[\mathscr{S}^{\widehat{\alpha}}_0\ V])]) \twoheadrightarrow V\ (\lambda x.\#^{\widehat{\alpha}}_0(E[\#^{\widehat{\beta}}_0(E'[x])]))$$

However, the term $\mu\alpha.c$ only captures its immediate evaluation context up to its surrounding command, stopping at the nearest binding of any dynamic co-variable. In order to express shift_0 with multiple prompts, we will need some way of directly manipulating the meta-context. This is reminiscent of the way single-prompt shift_0 removes the most recent binding of $\widehat{\mathsf{tp}}$ and exposes that context to an underlying term. Therefore, we need to incorporate both multiple prompt binding from Section 6 and dynamic prompts from Section 7.

Syntax. $\lambda\widehat{\mu}_0$ 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, as given in Figure 54. In the language of delimited control $\lambda\widehat{\mu}_0$, the new class of variables, Δ, stands in for a segment of a meta-context E^2, as in Figure 34, which represents zero or more evaluation contexts, E, bound to dynamic co-variables. The command $[\widehat{\alpha}]_0\Delta.t$ captures the segment of its meta-context as Δ, up to the nearest binding of $\widehat{\alpha}$. Then that segment of the meta-

$$S ::= \langle c^2 \rangle_{\mathsf{refocus}} \mid \langle c, F^{2*} \rangle_{\mathsf{refocus}} \mid \langle t, F^*, F^{2*} \rangle_{\mathsf{refocus}} \mid \langle F^*, V, F^{2*} \rangle_{\mathsf{apply}}$$
$$\mid \langle t, F^*, F^{2*} \rangle_{\mathsf{reduce}} \mid \langle F^{2*}, \widehat{\mathsf{tp}}, t \rangle_{\mathsf{reduce}} \mid \langle V \rangle_{\mathsf{done}}$$

Fig. 50. States of the call-by-value $\lambda\mu\widehat{\mathsf{tp}}_0$ machine.

$$\langle [q^2]c \rangle_{\mathsf{refocus}} \rightsquigarrow \langle c, q^2 \rangle_{\mathsf{refocus}}$$
$$\langle [q]t, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle t, q, F^{2*} \rangle_{\mathsf{refocus}}$$
$$\langle [\widehat{\mathsf{tp}}]_0 t, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle F^{2*}, \widehat{\mathsf{tp}}, t \rangle_{\mathsf{reduce}}$$
$$\langle t\ t', F^*, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle t, F^*[\Box\ t'], F^{2*} \rangle_{\mathsf{refocus}}$$
$$\langle \mu\alpha.c, F^*, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle \mu\alpha.c, F^*, F^{2*} \rangle_{\mathsf{reduce}}$$
$$\langle \mu_0\widehat{\mathsf{tp}}.c, F^*, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle c, F^{2*}[\widehat{\mathsf{tp}} \mapsto F^*] \rangle_{\mathsf{refocus}}$$
$$\langle V, F^*, F^{2*} \rangle_{\mathsf{refocus}} \rightsquigarrow \langle F^*, V, F^{2*} \rangle_{\mathsf{apply}}$$

$$\langle F^*[\Box\ t], V, F^{2*} \rangle_{\mathsf{apply}} \rightsquigarrow \langle t, F^*[V\ \Box], F^{2*} \rangle_{\mathsf{refocus}}$$
$$\langle F^*[V'\ \Box], V, F^{2*} \rangle_{\mathsf{apply}} \rightsquigarrow \langle V'\ V, F^*, F^{2*} \rangle_{\mathsf{reduce}}$$
$$\langle *, V, F^{2*} \rangle_{\mathsf{apply}} \rightsquigarrow \langle V \rangle_{\mathsf{done}}$$

$$\langle (\lambda x.t)\ V, F^*, F^{2*} \rangle_{\mathsf{reduce}} \rightsquigarrow \langle t\{V/x\}, F^*, F^{2*} \rangle_{\mathsf{refocus}}$$
$$\langle \mu\alpha.c, F^*, F^{2*} \rangle_{\mathsf{reduce}} \rightsquigarrow \langle c\{F^*/\alpha\}, F^{2*} \rangle_{\mathsf{refocus}}$$
$$\langle F^{2*}[\widehat{\mathsf{tp}} \mapsto F^*], \widehat{\mathsf{tp}}, t \rangle_{\mathsf{reduce}} \rightsquigarrow \langle t, F^*, F^{2*} \rangle_{\mathsf{refocus}}$$

Fig. 51. Abstract machine for call-by-value $\lambda\mu\widehat{\mathsf{tp}}_0$.

context is removed and the most recent binding of $\widehat{\alpha}$ becomes unbound. The term t is then evaluated in the context formerly bound to $\widehat{\alpha}$ and the remaining meta-context.

Reduction. The reduction rules for commands $[\widehat{\alpha}]_0\Delta.t$ 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, keeping the reduction rules fine-grained, as shown in Figure 55. When under a non-matching prompt $\widehat{\beta}$, the command $[\widehat{\alpha}]_0\Delta.t$ 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 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 the command $[\widehat{\alpha}]_0\Delta.t$ 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 rest of the reduction theory of the $\lambda\widehat{\mu}_0$-calculus is taken from the $\lambda\mu$ theory in Figure 10.

$$\#_0 t = \mu_0 \widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]t$$
$$\mathscr{S}_0 = \lambda h.\mu\alpha.[\widehat{\mathsf{tp}}]_0 h\ (\lambda x.\mu_0 \widehat{\mathsf{tp}}.[\alpha]x)$$

Fig. 52. Encodings of the $\mathscr{S}_0$ and $\#_0$ control operators in the $\lambda\mu\widehat{\mathsf{tp}}_0$-calculus.

$$E ::= \Box \mid E\ t \mid V\ E \qquad\qquad D ::= \Box \mid D[E[\#_0 \Box]]$$

$$D[E[(\lambda x.t)\ V]] \mapsto D[E[t\{V/x\}]]$$
$$D[E[\#_0 V]] \mapsto D[E[V]]$$
$$D[E'[\#_0 E[\mathscr{S}_0\ V]]] \mapsto D[E'[V\ (\lambda x.\#_0 E[x])]]$$

Fig. 53. Call-by-value evaluation contexts and operational semantics for the $\mathscr{S}_0$ and $\#_0$ operators.

Similar to $\lambda\mu\widehat{\mathsf{tp}}_0$, we can encode the $\lambda\widehat{\mu}$ command $[\widehat{\alpha}]t$ in $\lambda\widehat{\mu}_0$:

$$[\widehat{\alpha}]t = [*]\mathbf{let}\, x = t\, \mathbf{in}\, \mu\_.[\widehat{\alpha}]_0\_.x$$

With this encoding for $[\widehat{\alpha}]t$, we also have the following derived reductions:

$$[\widehat{\alpha}]V \twoheadrightarrow [\widehat{\alpha}]_0\_.V$$
$$[\widehat{\alpha}]\mu\beta.c \twoheadrightarrow c\{[\widehat{\alpha}]t/[\beta]t\}$$

This allows us to derive the usual reduction rules for dynamic co-variables in the $\lambda\widehat{\mu}$-calculus shown in Figure 33.

Example 9

The command $[\widehat{\beta}]_0\Delta.t$ removes dynamically bound contexts from its meta-context term, and can escape dynamic co-variables in t,

$$\begin{aligned}
&\mu_0\widehat{\alpha}.[\beta]\mu_0\widehat{\beta}.[\alpha]\mu_0\widehat{\alpha}.[\widehat{\beta}]_0\Delta.\mu_.[\widehat{\alpha}]x \\
&\to \mu_0\widehat{\alpha}.[\beta]\mu_0\widehat{\beta}.[\alpha]\mu\alpha'.[\widehat{\beta}]_0\Delta.\mu_.[\widehat{\alpha}]x \qquad\qquad \textbf{where}\ \widehat{\alpha} \neq \widehat{\beta} \\
&\to \mu_0\widehat{\alpha}.[\beta]\mu_0\widehat{\beta}.[\widehat{\beta}]_0\Delta.\mu_.[\widehat{\alpha}]x \\
&\to \mu_0\widehat{\alpha}.[\beta]\mu_.[\widehat{\alpha}]x \\
&\to \mu_0\widehat{\alpha}.[\widehat{\alpha}]x \\
&\twoheadrightarrow x
\end{aligned}$$

Example 10

Capturing and reinstalling a prefix of the evaluation meta-context, as performed by the command $c_0 = [\widehat{\alpha}_1]_0\Delta.\mu_0\widehat{\alpha}_1.[\Delta]c$, preserves its order so that $\mu_0\widehat{\alpha}_1.E^2[c_0] \twoheadrightarrow \mu_0\widehat{\alpha}_1.E^2[c]$

$$
\begin{array}{rl}
c^2 \in \mathit{Command}^2 ::= & [q^2]c \\
c \in \mathit{Command} ::= & [q]t \mid [\widehat{\alpha}]_0\Delta.t \mid [\Delta]c \\
t \in \mathit{Term} ::= V \mid t_1\ t_2 \mid \mu\alpha.c \mid \mu_0\widehat{\alpha}.c & \qquad q^2 \in \mathit{CoTerm}^2 ::= \bullet \\
V \in \mathit{Value} ::= x \mid \lambda x.t & \qquad q \in \mathit{CoTerm} ::= \alpha \mid *
\end{array}
$$

Fig. 54. The syntax of the $\lambda\widehat{\mu}_0$-calculus.

$$
\begin{array}{ll}
\mu_0\widehat{\alpha}.[\widehat{\alpha}]_0\Delta.t \to t\{c/[\Delta]c\} & \\
\mu_0\widehat{\alpha}.[\widehat{\beta}]_0\Delta.t \to \mu\alpha.[\widehat{\beta}]_0\Delta.t\{[\Delta][\alpha](\mu_0\widehat{\alpha}.c)/[\Delta]c\} & \textbf{where } \widehat{\alpha} \neq \widehat{\beta} \\
\mu_0\widehat{\alpha}.[*]V \to \mu\_.[*]V &
\end{array}
$$

Fig. 55. Call-by-value reduction theory of dynamic co-variables in the $\lambda\widehat{\mu}_0$-calculus.

when E^2 does not bind $\widehat{\alpha}_1$,

$$
\begin{array}{l}
[\beta_1]\mu_0\widehat{\alpha}_1.[\beta_2]\mu_0\widehat{\alpha}_2.[\beta_3]\mu_0\widehat{\alpha}_3.[\widehat{\alpha}_1]_0\Delta.\mu_0\widehat{\alpha}_1.[\Delta]c \\
\to [\beta_1]\mu_0\widehat{\alpha}_1.[\beta_2]\mu_0\widehat{\alpha}_2.[\beta_3]\mu\gamma.[\widehat{\alpha}_1]_0\Delta.\mu_0\widehat{\alpha}_1.[\Delta][\gamma]\mu_0\widehat{\alpha}_3.c \\
\to [\beta_1]\mu_0\widehat{\alpha}_1.[\beta_2]\mu_0\widehat{\alpha}_2.[\widehat{\alpha}_1]_0\Delta.\mu_0\widehat{\alpha}_1.[\Delta][\beta_3]\mu_0\widehat{\alpha}_3.c \\
\to [\beta_1]\mu_0\widehat{\alpha}_1.[\beta_2]\mu\gamma.[\widehat{\alpha}_1]_0\Delta.\mu_0\widehat{\alpha}_1.[\Delta][\gamma]\mu_0\widehat{\alpha}_2.[\beta_3]\mu_0\widehat{\alpha}_3.c \\
\to [\beta_1]\mu_0\widehat{\alpha}_1.[\widehat{\alpha}_1]_0\Delta.\mu\widehat{\alpha}_1.[\Delta][\beta_2]\mu_0\widehat{\alpha}_2.[\beta_3]\mu_0\widehat{\alpha}_3.c \\
\to [\beta_1]\mu_0\widehat{\alpha}_1.[\beta_2]\mu_0\widehat{\alpha}_2.[\beta_3]\mu_0\widehat{\alpha}_3.c
\end{array}
$$

Operational semantics. The operational semantics for $\lambda\widehat{\mu}_0$ is a variation of the semantics for $\lambda\widehat{\mu}$ from Figure 34, similar to the relationship between the operational semantics of $\lambda\mu\widehat{\mathsf{tp}}_0$ and $\lambda\mu\widehat{\mathsf{tp}}$. The modified (meta-)evaluation contexts and operational semantics are given in Figure 56. Similar to the evaluation meta-contexts of $\lambda\widehat{\mu}$ from Figure 34, contexts $E^2_{\widehat{\alpha}}$, i.e. prefixes of the evaluation meta-context up to the binding of $\widehat{\alpha}$, are defined as:

$$
E^2_{\widehat{\alpha}} ::= \Box \mid D[\mu_0\widehat{\beta}.E^2_{\widehat{\alpha}}] \qquad \textbf{where } \widehat{\beta} \not\equiv \widehat{\alpha}
$$

CPS transform. The shift_0 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 $V\ \widehat{x}$ to give us both the value that was stored in $\widehat{x}$ and 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 $\lambda\widehat{\mathsf{tp}}_0$ by combining both multiple dynamic variables and reversal of dynamic binding, giving us $\widehat{\lambda}_0$ in Figure 57. In the language of dynamic binding $\widehat{\lambda}_0$, the new class of variables, Δ, ranges over dynamic environment prefixes. Intuitively, the term

$$E^2 ::= \Box \mid D[\mu_0 \widehat{\alpha}.E^2] \mid [\Delta]E^2 \qquad D^2 ::= [q^2]E^2$$
$$E ::= \Box \mid E\ t \mid V\ E \qquad D ::= [q]E$$

$$D^2[D[(\lambda x.t)\ V]] \mapsto D^2[D[t\{V/x\}]]$$
$$D^2[D[\mu\alpha.c]] \mapsto D^2[c\{D[t]/[\alpha]t\}]$$
$$D^2[D[\mu_0\widehat{\alpha}.E^2_{\widehat{\alpha}}[[\widehat{\alpha}]_0\Delta.t]]] \mapsto D^2[D[t\{E^2_{\widehat{\alpha}}[c]/[\Delta]c\}]]$$

Fig. 56. Call-by-value evaluation contexts and operational semantics for the $\lambda\widehat{\mu}_0$-calculus.

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

Fig. 57. The syntax of the $\widehat{\lambda}_0$-calculus of dynamic binding.

$(\lambda\langle\Delta,x\rangle.t)\ \widehat{x}$ looks up the most recent binding of $\widehat{x}$, substituting the value bound to $\widehat{x}$ for x while also capturing the prefix of the environment more recent than $\widehat{x}$ and substituting it for 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. Therefore, in a program such as:

$$[\bullet]\mathbf{dlet}\,\widehat{x} = V_1\,\mathbf{in}\,\mathbf{dlet}\,\widehat{y} = V_2\,\mathbf{in}\,\mathbf{dlet}\,\widehat{z} = V_3\,\mathbf{in}\,(\lambda\langle\Delta,x\rangle.t)\ \widehat{x}$$

the static variable x will be instantiated with V_1 and Δ will be instantiated with the dynamic bindings of V_2 to $\widehat{y}$ and V_3 to $\widehat{z}$. This means that every free occurrence of the closure $[\Delta]u$ in the underlying term t will be substituted with: $\mathbf{dlet}\,\widehat{y} = V_2\,\mathbf{in}\,\mathbf{dlet}\,\widehat{z} = V_3\,\mathbf{in}\,u$.

The semantics of $\widehat{\lambda}_0$, such as $\lambda\widehat{\mathsf{tp}}_0$, 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 7, we first define the new environment concretely, using lists to implement environments and prefixes and tuples to return multiple values as shown in Figure 58. 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 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 occurred originally.

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. The environment-passing style transform $\mathscr{D}_{\widehat{\lambda}_0}$ is given in Figure 59.

The CPS translation from $\lambda\widehat{\mu}_0$ to $\widehat{\lambda}_0$ is a merging of $\widehat{\mathscr{C}}_{\lambda\widehat{\mu}}$ and $\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}$. The new syntactic forms in $\lambda\widehat{\mu}_0$ can be defined in terms of the intermediate language $\widehat{\lambda}_0$. Capturing a portion of the meta-context up to $\widehat{\alpha}$ translates to capturing a prefix of the dynamic environment

$$\mathscr{D}^{\times}_{\widehat{\lambda}_0}[\![\lambda\langle\Delta,x\rangle.t]\!]^V = \lambda\Delta.\lambda x.\mathscr{D}^{\times}_{\widehat{\lambda}_0}[\![t]\!]$$
$$\mathscr{D}^{\times}_{\widehat{\lambda}_0}[\![t\ \widehat{x}]\!] = \lambda\gamma.\,\mathbf{let}\,\langle\Delta,v,\gamma'\rangle = \gamma(\widehat{x})\,\mathbf{in}\,(\mathscr{D}^{\times}_{\widehat{\lambda}_0}[\![t]\!]\gamma)\ \Delta\ v\ \gamma'$$
$$\mathscr{D}^{\times}_{\widehat{\lambda}_0}[\![[\Delta]t]\!] = \lambda\gamma.\mathscr{D}^{\times}_{\widehat{\lambda}_0}[\![t]\!]\ (\gamma@\Delta)$$

$$\gamma[\widehat{x}\mapsto v](\widehat{x}) = \langle[],v,\gamma\rangle$$
$$\gamma[\widehat{y}\mapsto v](\widehat{x}) = \mathbf{let}\,\langle\Delta,v',\gamma'\rangle = \gamma(\widehat{x})\ \mathbf{in}\,\langle\Delta[\widehat{y}\mapsto v],v',\gamma'\rangle$$

$$\gamma@[] = \gamma$$
$$\gamma@(\Delta[\widehat{x}\mapsto v]) = (\gamma@\Delta)[\widehat{x}\mapsto v]$$

Fig. 58. Environment-passing style transform of the $\widehat{\lambda}_0$-calculus using an environment.

$$\mathscr{D}_{\widehat{\lambda}_0}[\![t\ \widehat{x}]\!] = \lambda\gamma.\gamma(\widehat{x})\ (\mathscr{D}_{\widehat{\lambda}_0}[\![t]\!]\gamma)$$
$$\mathscr{D}_{\widehat{\lambda}_0}[\![[\Delta]t]\!] = \lambda\gamma.\mathscr{D}_{\widehat{\lambda}_0}[\![t]\!]\ (\Delta\ \gamma)$$

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

Fig. 59. Refunctionalized environment-passing style transform of the $\widehat{\lambda}_0$-calculus.

while unbinding $\widehat{\alpha}$, and extending the meta-context becomes extending the dynamic environment. As in $\widehat{\mathscr{C}}_{\lambda\mu\widehat{\mathsf{tp}}_0}$, the invocation of a prompt is changed due to the change in the way dynamic variable lookup is performed. The CPS transform for $\lambda\widehat{\mu}_0$ is an extension of the basic $\mathscr{C}_{\lambda\mu}$ transform from Figure 12, and is given in Figure 60.

The final derived transform shares a resemblance with the one given by Dybvig *et al.* (2007), which lies in-between the meta-continuation approach of Danvy & Filinski (1989) and the abstract continuation approach of Felleisen *et al.* (1988). However, there is a subtle difference in the stack-like structure of the meta-continuation. Since the meta-continuation in the composed $\mathscr{D}_{\widehat{\lambda}_0}\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}$ transform is derived from an environment of bindings, its shape is always a list of co-variable, continuation pairs. In other words, the only way to push a continuation into the meta-continuation is to label it with some dynamic co-variable as is syntactically required in $E[\mu\widehat{\alpha}.c]$. Conversely, when a dynamic co-variable is removed from the meta-continuation through the lookup process, $[\widehat{\alpha}]_0\Delta.t$, the continuation bound to it is also extracted from the meta-continuation and is used as the immediate continuation for t.

These restrictions on the evaluation contexts and meta-contexts allow us to fully interpret the meaning of the meta-context statically at transformation time, rather than describe it as a data structure that must be interpreted dynamically during evaluation of the CPS program.

$$\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![x]\!]^V = x \qquad \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![\alpha]\!] = \alpha \qquad \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![\bullet]\!] = \bullet$$
$$\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![\lambda x.t]\!]^V = \lambda x.\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![t]\!] \qquad \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![*]\!] = \lambda x.x$$

$$\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![V]\!] = \lambda k.k\ \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![V]\!]^V \qquad \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![[q^2]c]\!] = [\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![q^2]\!]]\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![c]\!]$$
$$\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![t_1\ t_2]\!] = \lambda k.\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![t_1]\!]\lambda f.\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![t_2]\!]\lambda s.f\ s\ k \qquad \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![[q]t]\!] = \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![t]\!]\ \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![q]\!]$$
$$\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![\mu\alpha.c]\!] = \lambda k.(\lambda\alpha.\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![c]\!])\ k \qquad \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![[\widehat{\alpha}]_0\Delta.t]\!] = (\lambda\langle\Delta,k\rangle.\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![t]\!]k)\ \widehat{\alpha}$$
$$\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![\mu_0\widehat{\alpha}.c]\!] = \lambda k.(\lambda\widehat{\alpha}.\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![c]\!])\ k \qquad \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![[\Delta]c]\!] = [\Delta]\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![c]\!]$$

Fig. 60. Call-by-value CPS transform of $\widehat{\lambda}_0$ using dynamic binding.

$$\begin{aligned}
S ::= {} & \langle c^2\rangle_{\mathsf{refocus}} \mid \langle c, F^{2*}\rangle_{\mathsf{refocus}} \mid \langle t, F^*, F^{2*}\rangle_{\mathsf{refocus}} \\
& \mid \langle F^*, V, F^{2*}\rangle_{\mathsf{apply}} \mid \langle t, F^*, F^{2*}\rangle_{\mathsf{reduce}} \mid \langle \Delta.t, E^2, F^*, F^{2*}\rangle_{\mathsf{reduce}} \\
& \mid \langle F^{2*}, \widehat{\alpha}, L\rangle_{\mathsf{lookup}} \mid \langle L, E^2, F^*, F^{2*}\rangle_{\mathsf{collect}} \mid \langle V\rangle_{\mathsf{done}} \\
L ::= {} & \Delta.t \mid L[\widehat{\alpha} \mapsto F^*]
\end{aligned}$$

Fig. 61. States of the call-by-value $\lambda\widehat{\mu}_0$ machine.

In the $\mathscr{D}_{\widehat{\lambda}_0}\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}$ transform, there are no 'partial' continuations; every continuation ends by determining what to do next. This is reflected in the syntax of the language, where an evaluation context is made complete by terminating it in a command. Every complete evaluation context finishes by explicitly returning a value to another context ($[\alpha]E$), invoking a dynamically bound context ($[\widehat{\alpha}]E$), or exiting the program completely ($[*]E$). Additionally, both context and meta-context have exactly one operation: plug in a value and continue evaluation, or look up a dynamically bound context, respectively. Therefore, the continuation and meta-continuation may be transformed into functions that implement their respective single operation. Contrarily, the continuations from Felleisen *et al.* (1988) and the meta-continuations from Dybvig *et al.* (2007) effectively support several operations, i.e. plug in a value *and* search for a prompt, which prevents the transformation from concrete data structures to functions.

Abstract machine. Our final abstract machine for $\lambda\widehat{\mu}_0$ has the states shown in Figure 61, where L is a context that is formed during dynamic lookup, storing all the bindings that were skipped while searching for a specific dynamic co-variable. The steps of this machine are given in Figure 62.

Correctness. Finally, we have the same notion of correctness of the reduction theory with respect to the CPS transform for $\lambda\widehat{\mu}_0$ as we had with $\lambda\mu$, $\lambda\mu\widehat{\mathsf{tp}}$, $\lambda\widehat{\mu}$, and $\lambda\mu\widehat{\mathsf{tp}}_0$.

Theorem 8.1 (*Soundness*)
If $M \twoheadrightarrow M'$ then $\mathscr{D}_{\widehat{\lambda}_0}\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![M]\!] =_{\beta\eta} \mathscr{D}_{\widehat{\lambda}_0}\widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![M']\!]$.

$$\langle [q^2]c\rangle_{\mathsf{refocus}} \rightsquigarrow \langle c, q^2\rangle_{\mathsf{refocus}}$$
$$\langle [q]t, F^{2*}\rangle_{\mathsf{refocus}} \rightsquigarrow \langle t, q, F^{2*}\rangle_{\mathsf{refocus}}$$
$$\langle [\Delta]c, F^{2*}\rangle_{\mathsf{refocus}} \rightsquigarrow \langle c, F^{2*}@\Delta\rangle_{\mathsf{refocus}}$$
$$\langle [\widehat{\alpha}]_0\Delta.t, F^{2*}\rangle_{\mathsf{refocus}} \rightsquigarrow \langle F^{2*}, \widehat{\alpha}, \Delta.t\rangle_{\mathsf{lookup}}$$
$$\langle t\ t', F^*, F^{2*}\rangle_{\mathsf{refocus}} \rightsquigarrow \langle t, F^*[\Box\ t'], F^{2*}\rangle_{\mathsf{refocus}}$$
$$\langle \mu\alpha.c, F^*, F^{2*}\rangle_{\mathsf{refocus}} \rightsquigarrow \langle \mu\alpha.c, F^*, F^{2*}\rangle_{\mathsf{reduce}}$$
$$\langle \mu_0\widehat{\alpha}.c, F^*, F^{2*}\rangle_{\mathsf{refocus}} \rightsquigarrow \langle c, F^{2*}[\widehat{\alpha} \mapsto F^*]\rangle_{\mathsf{refocus}}$$
$$\langle V, F^*, F^{2*}\rangle_{\mathsf{refocus}} \rightsquigarrow \langle F^*, V, F^{2*}\rangle_{\mathsf{apply}}$$

$$\langle F^*[\Box\ t], V, F^{2*}\rangle_{\mathsf{apply}} \rightsquigarrow \langle t, F^*[V\ \Box], F^{2*}\rangle_{\mathsf{refocus}}$$
$$\langle F^*[V'\ \Box], V, F^{2*}\rangle_{\mathsf{apply}} \rightsquigarrow \langle V'\ V, F^*, F^{2*}\rangle_{\mathsf{reduce}}$$
$$\langle *, V, F^{2*}\rangle_{\mathsf{apply}} \rightsquigarrow \langle V\rangle_{\mathsf{done}}$$

$$\langle (\lambda x.t)\ V, F^*, F^{2*}\rangle_{\mathsf{reduce}} \rightsquigarrow \langle t\{V/x\}, F^*, F^{2*}\rangle_{\mathsf{refocus}}$$
$$\langle \mu\alpha.c, F^*, F^{2*}\rangle_{\mathsf{reduce}} \rightsquigarrow \langle c\{F^*/\alpha\}, F^{2*}\rangle_{\mathsf{refocus}}$$
$$\langle \Delta.t, E^2, F^*, F^{2*}\rangle_{\mathsf{reduce}} \rightsquigarrow \langle t\{E^2/\Delta\}, F^*, F^{2*}\rangle_{\mathsf{refocus}}$$

$$\langle F^{2*}[\widehat{\alpha} \mapsto F^*], \widehat{\alpha}, L\rangle_{\mathsf{lookup}} \rightsquigarrow \langle L, [], F^*, F^{2*}\rangle_{\mathsf{collect}}$$
$$\langle F^{2*}[\widehat{\beta} \mapsto F^*], \widehat{\alpha}, L\rangle_{\mathsf{lookup}} \rightsquigarrow \langle F^{2*}, \widehat{\alpha}, L[\widehat{\beta} \mapsto F^*]\rangle_{\mathsf{lookup}}$$
$$\langle F^{2*}@E'^2, \widehat{\alpha}, L\rangle_{\mathsf{lookup}} \rightsquigarrow \langle F^{2*}E'^2, \widehat{\alpha}, L\rangle_{\mathsf{lookup}}$$

$$\langle \Delta.t, E^2, F^*, F^{2*}\rangle_{\mathsf{collect}} \rightsquigarrow \langle \Delta.t, E^2, F^*, F^{2*}\rangle_{\mathsf{reduce}}$$
$$\langle L[\widehat{\alpha} \mapsto F'^*], E^2, F^*, F^{2*}\rangle_{\mathsf{collect}} \rightsquigarrow \langle L, E^2[\widehat{\alpha} \mapsto F'^*], F^*, F^{2*}\rangle_{\mathsf{collect}}$$

Fig. 62. Abstract machine for the call-by-value $\lambda\widehat{\mu}_0$ calculus.

Theorem 8.2 (*Evaluation*)
If $\mathscr{D}_{\widehat{\lambda}_0}\ \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![c^2]\!] =_\beta V$ then there is a final answer c'^2 such that $c^2 \mapsto\!\!\!\!\rightarrow c'^2$ and $\mathscr{D}_{\widehat{\lambda}_0}\ \widehat{\mathscr{C}}_{\lambda\widehat{\mu}_0}[\![c'^2]\!] =_\beta V$.

A final answer of the $\lambda\widehat{\mu}_0$-calculus is a meta-command of the form $D^2[[*]V]$, the same as for $\lambda\widehat{\mu}$ and $\lambda\mu\widehat{\mathsf{tp}}_0$. For further discussion and proofs of these theorems, see Section A.6 in the appendix.

Expressiveness. With the ability to capture the dynamic environment up to a given prompt, we can encode the behavior of shift_0 and reset_0 with multiple prompts as given in Figure 63. The term $\#_0^{\widehat{\alpha}}t$ binds its current context to the dynamic co-variable α and then evaluates t under that binding. If t evaluates to a value V, then V is returned as the result of the expression. Otherwise, if the term $\mathscr{S}_0^{\widehat{\alpha}}\ V'$ is encountered while evaluating t, then the $\mathscr{S}_0^{\widehat{\alpha}}$ captures the current context as well as the dynamic prefix up to the most recent binding

$$\#_0^{\widehat{\alpha}} t = \mu_0 \widehat{\alpha}.[\widehat{\alpha}]t$$
$$\mathscr{S}_0^{\widehat{\alpha}} = \lambda h.\mu\beta.[\widehat{\alpha}]_0\Delta.(h\ (\lambda x.\mu_0\widehat{\alpha}.[\Delta][\beta]x))$$

Fig. 63. Encodings of the $\mathscr{S}_0^{\widehat{\alpha}}$ and $\#_0^{\widehat{\alpha}}$ control operators in the $\lambda\widehat{\mu}_0$-calculus.

$$E ::= \Box \mid E\ t \mid V\ E$$
$$D ::= \Box \mid E[\#_0^{\widehat{\alpha}} D]$$

$$D[E[(\lambda x.t)\ V]] \mapsto D[E[t\{V/x\}]]$$
$$D[E[\#_0^{\widehat{\alpha}} V]] \mapsto D[E[V]]$$
$$D[E[\#_0^{\widehat{\alpha}} D'_{\widehat{\alpha}}[E'[\mathscr{S}_0^{\widehat{\alpha}}\ V]]]] \mapsto D[E[V\ (\lambda x.\#_0^{\widehat{\alpha}} D'_{\widehat{\alpha}}[E'[x]])]]$$

Fig. 64. Call-by-value evaluation contexts and operational semantics of the $\mathscr{S}_0^{\widehat{\alpha}}$ and $\#_0^{\widehat{\alpha}}$ control operators.

of $\widehat{\alpha}$, removing the prompt $\#_0^{\widehat{\alpha}}$ binding in the process. Then V' is given a function which, when applied, will evaluate its argument in the captured context and dynamic prefix under a new binding of $\widehat{\alpha}$.

Using the operational semantics for $\lambda\widehat{\mu}_0$, we derive the operational semantics for our encoding of the $\#_0^{\widehat{\alpha}}$ and $\mathscr{S}_0^{\widehat{\alpha}}$ control operators as shown in Figure 64. Note that in the third rule, the context $D'_{\widehat{\alpha}}$ does not contain a visible reset of $\widehat{\alpha}$.

Example 11

To illustrate how the reduction theory of $\lambda\widehat{\mu}_0$ can simulate the $\mathscr{S}_0^{\widehat{\alpha}}$ and $\#_0^{\widehat{\alpha}}$ control operators, consider the following step according to the operational semantics from Figure 64:

$$\#_0^{\widehat{\alpha}} E[\#_0^{\widehat{\beta}} E'[\mathscr{S}_0^{\widehat{\alpha}}\ V]] \mapsto V\ (\lambda x.\#_0^{\widehat{\alpha}} E[\#_0^{\widehat{\beta}} E'[x]])$$

Using the encoding of these control operators from Figure 63, we have the following reduction sequence:

$$\begin{aligned}
&\#_0^{\widehat{\alpha}} E[\#_0^{\widehat{\beta}} E'[\mathscr{S}_0^{\widehat{\alpha}}\ V]] \\
&\to \mu_0\widehat{\alpha}.[\widehat{\alpha}]E[\mu_0\widehat{\beta}.[\widehat{\beta}]E'[\mu\gamma.[\widehat{\alpha}]_0\Delta.V\ (\lambda x.\mu_0\widehat{\alpha}.[\Delta][\gamma]x)] \\
&\twoheadrightarrow \mu_0\widehat{\alpha}.[\widehat{\alpha}]E[\mu_0\widehat{\beta}.[\widehat{\beta}]\mu\gamma.[\widehat{\alpha}]_0\Delta.V\ (\lambda x.\mu_0\widehat{\alpha}.[\Delta][\gamma]E'[x])] \\
&\to \mu_0\widehat{\alpha}.[\widehat{\alpha}]E[\mu_0\widehat{\beta}.[\widehat{\alpha}]_0\Delta.V\ (\lambda x.\mu_0\widehat{\alpha}.[\Delta][\widehat{\beta}]E'[x])] \\
&\to \mu_0\widehat{\alpha}.[\widehat{\alpha}]E[\mu\gamma.[\widehat{\alpha}]_0\Delta.V\ (\lambda x.\mu_0\widehat{\alpha}.[\Delta][\gamma]\mu_0\widehat{\beta}.[\widehat{\beta}]E'[x])] \\
&\twoheadrightarrow \mu_0\widehat{\alpha}.[\widehat{\alpha}]\mu\gamma.[\widehat{\alpha}]_0\Delta.V\ (\lambda x.\mu_0\widehat{\alpha}.[\Delta][\gamma]E[\mu_0\widehat{\beta}.[\widehat{\beta}]E'[x]]) \\
&\twoheadrightarrow \mu_0\widehat{\alpha}.[\widehat{\alpha}]_0\Delta.V\ (\lambda x.\mu_0\widehat{\alpha}.[\Delta][\widehat{\alpha}]E[\mu_0\widehat{\beta}.[\widehat{\beta}]E'[x]]) \\
&\to V\ (\lambda x.\mu_0\widehat{\alpha}.[\widehat{\alpha}]E[\mu_0\widehat{\beta}.[\widehat{\beta}]E'[x]]) \\
&= V\ (\lambda x.\#_0^{\widehat{\alpha}} E[\#_0^{\widehat{\beta}} E'[x]])
\end{aligned}$$

The first step is a β_v-reduction passing in the value V to the encoding of $\mathscr{S}_0^{\widehat{\alpha}}$. The next two reductions capture the evaluation context immediately surrounding the call site of shift_0,

represented as an ordinary μ-abstraction of the $\lambda\mu$-calculus. The remaining reductions perform the search through the dynamic meta-context, capturing the prefix until the reset_0 of $\widehat{\alpha}$ is found.

9 Computational effects

Part of the interest in delimited control is its ability to simulate many other computational effects in direct style. This means that definitions of effectful operations can be defined locally using delimited control, e.g. as functions, without the need of transforming the entire program as in a CPS transform. We first show this technique of defining computational effects in the $\lambda\mu\widehat{\mathsf{tp}}$-calculus by monadic encodings in the style of Filinski (1994, 1999). We use exceptions and mutable state as toy examples of the technique. We then show how the extended languages of delimited control, $\lambda\widehat{\mu}$, $\lambda\mu\widehat{\mathsf{tp}}_0$, and $\lambda\widehat{\mu}_0$, allow us to model richer, more full-fledged versions of these effects through the use of multiple delimiters and more control over an expression's dynamic context.

9.1 Delimited control as a universal effect

Pure functional programming languages, such as Haskell in particular, can simulate effectful programming through the use of monads. The general idea is to encode the effect as some type constructor T so that values of type $T\ a$ represent computations that return results of type a while also using effects allowed in T. The operations common to all monads form the glue blocks that allow programmers to assemble larger computations out of the primitives. In Haskell, these monadic operations are return_T and $>\!\!>\!\!=_T$[3], with the types:

$$\mathsf{return}_T : a \to T\ a \qquad\qquad >\!\!>\!\!=_T : T\ a \to (a \to T\ b) \to T\ b$$

The return_T function turns a value into a trivial T-computation that returns the value without performing any effects. The $>\!\!>\!\!=_T$ operator (often pronounced 'bind') chains two T-computations together, running the first computation to extract its return value and passing that value along to the given function to produce the next computation to be executed.

As noted by Moggi (1989, 1991), the connection between computations and values of the monad can be officially supported in a language through monadic reflection. In other words, a programmer can *reflect* a value of $T\ a$, promoting it to an actual effectful computation of type a. Conversely, a computation producing a result of type a can be *reified* into a value of type $T\ a$, freezing any effects that would occur during the execution of the computation. Although not ordinary functions, these special operations for a particular monad T have the types:

$$\mathbf{reify}_T : a \to T\ a \qquad\qquad \mathbf{reflect}_T : T\ a \to a$$

In order to give the correct result, **reflect** acts as an ordinary call-by-value function and evaluates its argument first, whereas $\mathbf{reify}\,t$ is a special form that first creates a barrier

[3] Here the monadic operations are annotated with the type of the specific monad to which they belong.

around t to isolate its effects before evaluating it. Furthermore, **reflect** and **reify** represent inverse operations from one another, and so they ought to obey the following laws for any monad:

$$\mathbf{reflect}_T(\mathbf{reify}_T\, t) = t \qquad\qquad \mathbf{reify}_T(\mathbf{reflect}_T\, V) = V$$

In his seminal work, Filinski (1994, 1999) observed that a call-by-value functional language with delimited control has all the tools necessary to support monadic reflection. In particular, the **reflect** and **reify** operations can be encoded in terms of the shift and reset operators,

$$\mathbf{reify}_T\, t = \#(\mathsf{return}_T\, t)$$
$$\mathbf{reflect}_T\, t = \mathscr{S}\lambda k.(t >\!\!>\!\!=_T k)$$

This can be equivalently encoded in $\lambda\mu\widehat{\mathsf{tp}}$, using our previous definitions of shift and reset,

$$\mathbf{reify}_T\, t = \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}](\mathsf{return}_T\, t)$$
$$\mathbf{reflect}_T\, t = \mu\alpha.[\widehat{\mathsf{tp}}](t >\!\!>\!\!=_T \lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x)$$

Although we are working in an untyped setting, in order to understand how **reflect** and **reify** interact with one another, it may help to first understand the types involved on an informal level.[4] First, suppose that $\mathbf{reflect}_T$ is used on some value V in a context where it has a surrounding reset.

$$\ldots\#E[\mathbf{reflect}_T\, V]\ldots$$

Expanding the definition in terms of $\mathscr{S}$, we have:

$$\ldots\#E[\mathscr{S}\lambda k.(V >\!\!>\!\!=_T k)]\ldots$$

Now according to the definition of the $>\!\!>\!\!=_T$ operator, V must have type $T\ a$ and the function k must have the type $a \to T\ b$. Since k is a functional representation of the evaluation context E, this means that E is expecting a value of type a, and so the entire expression $\mathbf{reflect}_T\, V$ must have type a. Furthermore, E must produce an output of type $T\ b$, and so the nearest reset must be expecting a value of type $T\ b$. Finally, the $\mathscr{S}$ delivers the result of $V >\!\!>\!\!=_T k$ to the nearest reset, which also has the type $T\ b$, as is expected. On the other hand, for a term t of type a, $\mathbf{reify}_T\, t$ surrounds the term with a reset expecting a value of type $T\ a$. If t produces a value without using any control effects, then that value is injected into the type $T\ a$ with a return_T. Otherwise any control effect in t, like $\mathbf{reflect}_T$, is expected to give a monadic value.

The general strategy to defining an effect through monadic reflection is as follows:

1. Define the type[5] T and the corresponding monadic operations for the particular effect.
2. Simulate the primitive operations of the effect as pure functional programs in the monadic type.

[4] For a more formal discussion on type and effect systems for delimited control, see Danvy & Filinski (1989); Filinski (1999); and Ariola *et al.* (2009) .

[5] Because we are in the untyped setting, the monadic type is for illustrative purposes, and we do not statically check for type-safety of the following programs.

3. Promote the pure primitives to their effectful counterparts using **reflect** and **reify**, as appropriate.

We will follow this exercise for simplistic versions of two basic effects in programming languages: exceptions and mutable state.

9.1.1 Exceptions

Simple exceptions can be represented monadically as a sum type, tagging the result as either a successful return value, or some exception of type e. Whenever an exception is encountered, the rest of the current computation is abandoned and the exception is raised,

$$\begin{aligned} \mathsf{Error}_e\, a &= \mathsf{Exn}\, e \mid \mathsf{OK}\, a \\ \mathsf{return}_{\mathsf{Error}_e}\, x &= \mathsf{OK}\, x \\ m \gg\!=_{\mathsf{Error}_e} f &= \mathbf{case}\, m\, \mathbf{of}\, \mathsf{OK}\, x \Rightarrow f\, x \mid \mathsf{Exn}\, z \Rightarrow \mathsf{Exn}\, z \end{aligned}$$

Raising an exception just requires marking an appropriate value with the Exn tag,

$$\mathsf{raise}' = \lambda z.\, \mathsf{Exn}\, z$$

This pure version of exception raising is promoted to the effectful version of the operation by using **reflect** for the Error_e monad.

$$\begin{aligned} \mathsf{raise} &= \lambda z.\, \mathbf{reflect}_{\mathsf{Error}_e}(\mathsf{raise}'\, z) \\ \mathsf{raise} &= \lambda z.\mu\_.[\widehat{\mathsf{tp}}]\mathsf{Exn}\, z \end{aligned}$$

However, in a practical setting, we are also interested in handling exceptions. This corresponds exactly with using **reify** to freeze the computation as a value of the sum type $\mathsf{Error}_e\, a$ and checking for whether or not an exception was raised. An SML-like exception handling mechanism can be defined as:

$$\begin{aligned} t\, \mathbf{handle}\, z \Rightarrow u &= \mathbf{case}\ (\mathbf{reify}_{\mathsf{Error}_e}\, t)\, \mathbf{of} \\ &\qquad \mid \mathsf{OK}\, x \Rightarrow x \\ &\qquad \mid \mathsf{Exn}\, z \Rightarrow u \\ t\, \mathbf{handle}\, z \Rightarrow u &= \mathbf{case}(\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{OK}\, t)\, \mathbf{of} \\ &\qquad \mid \mathsf{OK}\, x \Rightarrow x \\ &\qquad \mid \mathsf{Exn}\, z \Rightarrow u \end{aligned}$$

Using the reduction semantics of $\lambda\mu\widehat{\mathsf{tp}}$, we can show that our encoding of exceptions behaves as expected. When the body reduces to a value without raising an exception, then that value is returned as the result. However, if an exception is raised during execution, then the rest of the computation is aborted, and that value is provided to the exception handler.

$$\begin{aligned} V\, \mathbf{handle}\, z \Rightarrow t &\longrightarrow\!\!\!\!\rightarrow V \\ E[\mathsf{raise}\, V]\, \mathbf{handle}\, z \Rightarrow t &\longrightarrow\!\!\!\!\rightarrow t\{V/z\} \end{aligned}$$

Example 12
Stepping through the derivation of the above reductions gives an idea of how this encoding captures the behavior of exceptions. In general, there are only two possibilities: execution of the body successfully produces a value, or execution of the body is interrupted by some raised exception. These two cases are handled as follows:

$$\begin{aligned} V\,\mathbf{handle}\,z \Rightarrow t &= \mathbf{case}\,(\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{OK}\,V)\,\mathbf{of}\,\mathsf{OK}\,x \Rightarrow x \mid \mathsf{Exn}\,z \Rightarrow t \\ &\to \mathbf{case}\,\mathsf{OK}\,V\,\mathbf{of}\,\mathsf{OK}\,x \Rightarrow x \mid \mathsf{Exn}\,z \Rightarrow t \\ &\to V \end{aligned}$$

$$\begin{aligned} E[\mathsf{raise}\,V]\,\mathbf{handle}\,z \Rightarrow t &= \mathbf{case}\,(\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{OK}\,E[\mu\_.[\widehat{\mathsf{tp}}]\mathsf{Exn}\,V])\,\mathbf{of}\,\mathsf{OK}\,x \Rightarrow x \mid \mathsf{Exn}\,z \Rightarrow t \\ &\twoheadrightarrow \mathbf{case}\,(\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{Exn}\,V)\,\mathbf{of}\,\mathsf{OK}\,x \Rightarrow x \mid \mathsf{Exn}\,z \Rightarrow t \\ &\to \mathbf{case}\,\mathsf{Exn}\,V\,\mathbf{of}\,\mathsf{OK}\,x \Rightarrow x \mid \mathsf{Exn}\,z \Rightarrow t \\ &\to t\{V/z\} \end{aligned}$$

We also have an operational semantics for exceptions that is derived from our encoding,

$$\begin{aligned} E &::= \square \mid E\,t \mid V\,E \\ D &::= \square \mid E[D\,\mathbf{handle}\,z \Rightarrow t] \end{aligned}$$

$$\begin{aligned} D[E[(\lambda x.t)\,V]] &\mapsto D[E[t\{V/x\}]] \\ D[E[V\,\mathbf{handle}\,z \Rightarrow t]] &\mapsto D[E[V]] \\ D[E[(E'[\mathsf{raise}\,V])\,\mathbf{handle}\,z \Rightarrow t]] &\mapsto D[E[t\{V/z\}]] \end{aligned}$$

In the last rule, E' does not contain a handler, by definition.

9.1.2 State

Mutable state can be represented monadically as a function transforming an initial state into a return value and an updated state. This models the simplified case when there is exactly one mutable cell,

$$\begin{aligned} \mathsf{State}_s\,a &= s \to (a,s) \\ \mathsf{return}_{\mathsf{State}_s}\,x &= \lambda z.(x,z) \\ m \gg\!=_{\mathsf{State}_s} f &= \lambda z.\,\mathbf{let}(x,z') = m\,z\,\mathbf{in}\,f\,x\,z' \end{aligned}$$

The primitive operations for getting the current value of the state and putting a new value for the state are defined as:

$$\mathsf{get}' = \lambda z.(z,z) \qquad\qquad \mathsf{put}' = \lambda z'.\lambda\_.((),z')$$

The pure functional primitives are promoted to their effectful counterparts using **reflect** for the State_s monad,

$$\begin{aligned} \mathsf{get} &= \mathbf{reflect}_{\mathsf{State}_s}\,\mathsf{get}' & \mathsf{put} &= \lambda z'.\,\mathbf{reflect}_{\mathsf{State}_s}(\mathsf{put}'\,z') \\ \mathsf{get} &= \mu\alpha.[\widehat{\mathsf{tp}}](\lambda z.(\mu\widehat{\mathsf{tp}}.[\alpha]z)\,z) & \mathsf{put} &= \lambda z'.\mu\alpha.[\widehat{\mathsf{tp}}](\lambda\_.(\mu\widehat{\mathsf{tp}}.[\alpha]())\,z') \end{aligned}$$

However, in order for these operations to work, they must be used in a context in which there is a delimiter, i.e. **reify**, that is prepared to handle stateful operations. This is similar to the requirement that the mutable cell must first be allocated before it is used. The allocation delimiter can be defined as:

$$\begin{aligned}\mathbf{alloc}\, V\, \mathbf{in}\, t &= (\mathbf{reify}_{\mathsf{State}_s}\, t)\ V \\ \mathbf{alloc}\, V\, \mathbf{in}\, t &= (\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{return}_{\mathsf{State}_s}\, t)\ V\end{aligned}$$

The expression $\mathbf{alloc}\, V\, \mathbf{in}\, t$ allocates the value V as the value of the single mutable cell. When t is interrupted by a get or put operation, the current execution is paused, and V is provided as the value of the state. Using the reduction semantics of $\lambda\mu\widehat{\mathsf{tp}}$, we get the following rules for mutable state:

$$\begin{aligned}\mathbf{alloc}\, V\, \mathbf{in}\, V' &\twoheadrightarrow (V', V) \\ \mathbf{alloc}\, V\, \mathbf{in}\, E[\mathsf{get}] &\twoheadrightarrow \mathbf{alloc}\, V\, \mathbf{in}\, E[V] \\ \mathbf{alloc}\, V\, \mathbf{in}\, E[\mathsf{put}\, V'] &\twoheadrightarrow \mathbf{alloc}\, V'\, \mathbf{in}\, E[()]\end{aligned}$$

Example 13

As before, we can look to the derivation of the above reductions to understand how stateful computation takes place. The first case to consider is how values are returned from an allocation expression,

$$\begin{aligned}\mathbf{alloc}\, V\, \mathbf{in}\, V' &= (\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{return}_{\mathsf{State}_s}\, V')\ V \\ &\to (\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}](\lambda z.(V', z)))\ V \\ &\to (\lambda z.(V', z))\ V \\ &\to (V', V)\end{aligned}$$

Next, we can see how the two basic stateful primitives, get and put, reach out to the surrounding allocation delimiter in order to check the current value of the state,

$$\begin{aligned}\mathbf{alloc}\, V\, \mathbf{in}\, E[\mathsf{get}] &= (\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{return}_{\mathsf{State}_s}\, E[\mu\alpha.[\widehat{\mathsf{tp}}]\lambda z.(\mu\widehat{\mathsf{tp}}.[\alpha]z)\ z])\ V \\ &\twoheadrightarrow (\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}](\lambda z.(\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{return}_{\mathsf{State}_s}\, E[z])\ z))\ V \\ &\to (\lambda z.(\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{return}_{\mathsf{State}_s}\, E[z])\ z)\ V \\ &\to (\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{return}_{\mathsf{State}_s}\, E[V])\ V \\ &= \mathbf{alloc}\, V\, \mathbf{in}\, E[V]\end{aligned}$$

$$\begin{aligned}\mathbf{alloc}\, V\, \mathbf{in}\, E[\mathsf{put}\, V'] &\to (\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{return}_{\mathsf{State}_s}\, E[\mu\alpha.[\widehat{\mathsf{tp}}]\lambda z.(\mu\widehat{\mathsf{tp}}.[\alpha]())\ V'])\ V \\ &\twoheadrightarrow (\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\lambda z.(\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{return}_{\mathsf{State}_s}\, E[()])\ V')\ V \\ &\to (\lambda z.(\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{return}_{\mathsf{State}_s}\, E[()])\ V')\ V \\ &\to (\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathsf{return}_{\mathsf{State}_s}\, E[()])\ V' \\ &= \mathbf{alloc}\, V'\, \mathbf{in}\, E[()]\end{aligned}$$

We can further derive an operational semantics for this style of delimited mutable state from our encoding into $\lambda\mu\widehat{\mathsf{tp}}$

$$E ::= \Box \mid E\ t \mid V\ E$$
$$D ::= \Box \mid E[\mathbf{alloc}\, V\, \mathbf{in}\, D]$$

$$D[E[(\lambda x.t)\ V]] \mapsto D[E[t\{V/x\}]]$$
$$D[E[\mathbf{alloc}\, V\, \mathbf{in}\, V']] \mapsto D[E[(V, V')]]$$
$$D[E[\mathbf{alloc}\, V\, \mathbf{in}\, E'[\mathsf{get}]]] \mapsto D[E[\mathbf{alloc}\, V\, \mathbf{in}\, E'[V]]]$$
$$D[E[\mathbf{alloc}\, V\, \mathbf{in}\, E'[\mathsf{put}\, V']]] \mapsto D[E[\mathbf{alloc}\, V'\, \mathbf{in}\, E'[()]]]$$

9.2 Delimited control as exceptions

In our previous encoding of exceptions, a handler caught every exception that was raised in its body. However, a more useful model of exceptions allows for only certain exceptions to be caught by a handler. The previous encoding can be extended with this extra feature by using the many different dynamic co-variables available in $\lambda\widehat{\mu}$. The key difference is to extend the raise and **handle** primitives from using only $\widehat{\mathsf{tp}}$ to instead allow any dynamic co-variable,

$$\mathsf{raise}\ \widehat{\alpha} = \lambda z.\mu\_.[\widehat{\alpha}]\mathsf{Exn}\, z$$
$$t\ \mathbf{handle}\ \widehat{\alpha}\ z \Rightarrow u = \mathbf{case}\, \mu\widehat{\alpha}.[\widehat{\alpha}]\mathsf{OK}\, t\ \mathbf{of}\, \mathsf{OK}\, x \Rightarrow x \mid \mathsf{Exn}\, z \Rightarrow u$$

The expression $\mathsf{raise}\,\widehat{\alpha}\ t$ evaluates t and then aborts to the dynamically nearest handler for $\widehat{\alpha}$ with an exceptional value. The handling expression $t\ \mathbf{handle}\ \widehat{\alpha}\ z \Rightarrow u$ 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 $\widehat{\alpha}$ is raised (with the exceptional value represented as $\mathsf{Exn}\, V$), then u is evaluated with V bound to z.

Using the reduction semantics of $\lambda\widehat{\mu}$, we find the behavior for this extended encoding of named exception handling

$$V\ \mathbf{handle}\ \widehat{\alpha}\ z \Rightarrow t \twoheadrightarrow V$$
$$E[\mathsf{raise}\,\widehat{\alpha}\ V]\ \mathbf{handle}\ \widehat{\alpha}\ z \Rightarrow t \twoheadrightarrow t\{V/z\}$$
$$E[\mathsf{raise}\,\widehat{\alpha}\ V]\ \mathbf{handle}\ \widehat{\beta}\ z \Rightarrow t \twoheadrightarrow \mathsf{raise}\,\widehat{\alpha}\ V \qquad \mathbf{where}\ \widehat{\alpha} \neq \widehat{\beta}$$

Example 14

The derivation of the first reduction, when the body of a handler successfully produces a value, is essentially the same as the previous encoding in $\lambda\mu\widehat{\mathsf{tp}}$. When an $\widehat{\alpha}$ exception is raised inside a handler for $\widehat{\alpha}$, the handler stops the exception and takes its recovery branch,

$$\begin{aligned}
&E[\mathsf{raise}\,\widehat{\alpha}\ V]\ \mathbf{handle}\ \widehat{\alpha}\ z \Rightarrow t \\
&\to \mathbf{case}(\mu\widehat{\alpha}.[\widehat{\alpha}]\mathsf{OK}\, E[\mu_.[\widehat{\alpha}]\mathsf{Exn}\, V])\ \mathbf{of}\, \mathsf{OK}\, x \Rightarrow x \mid \mathsf{Exn}\, z \Rightarrow t \\
&\twoheadrightarrow \mathbf{case}(\mu\widehat{\alpha}.[\widehat{\alpha}]\mathsf{Exn}\, V)\ \mathbf{of}\, \mathsf{OK}\, x \Rightarrow x \mid \mathsf{Exn}\, z \Rightarrow t \\
&\to \mathbf{case}\, \mathsf{Exn}\, V\ \mathbf{of}\, \mathsf{OK}\, x \Rightarrow x \mid \mathsf{Exn}\, z \Rightarrow t \\
&\to t\{V/z\}
\end{aligned}$$

However, if an $\widehat{\alpha}$ exception is raised inside a handler for some other kind of exception, the handler is discarded and the exception continues to propagate,

$$\begin{aligned}
&E[\mathsf{raise}\,\widehat{\alpha}\ V]\,\mathbf{handle}\,\widehat{\beta}\ z \Rightarrow t \\
&\to \mathbf{case}(\mu\widehat{\beta}.[\widehat{\beta}]\mathsf{OK}\,E[\mu_.[\widehat{\alpha}]\mathsf{Exn}\,V])\,\mathbf{of}\,\mathsf{OK}\,x \Rightarrow x \mid \mathsf{Exn}\,z \Rightarrow t \\
&\twoheadrightarrow \mathbf{case}(\mu\widehat{\beta}.[\widehat{\alpha}]\mathsf{Exn}\,V)\,\mathbf{of}\,\mathsf{OK}\,x \Rightarrow x \mid \mathsf{Exn}\,z \Rightarrow t \\
&\to \mathbf{case}(\mu_.[\widehat{\alpha}]\mathsf{Exn}\,V)\,\mathbf{of}\,\mathsf{OK}\,x \Rightarrow x \mid \mathsf{Exn}\,z \Rightarrow t \\
&\to \mu_.[\widehat{\alpha}]\mathsf{Exn}\,V \\
&= \mathsf{raise}\,\widehat{\alpha}\ V
\end{aligned}$$

As before, we can derive an operational semantics of named exceptions from this encoding into $\lambda\widehat{\mu}$, where a meta-context $D_{\widehat{\alpha}}$ does not contain a handler for $\widehat{\alpha}$

$$E ::= \square \mid E\ t \mid V\ E \mid \mathsf{raise}\ \widehat{\alpha}\ E \qquad\qquad D ::= \square \mid E[D\,\mathbf{handle}\,\widehat{\alpha}\ x \Rightarrow u]$$

$$\begin{aligned}
D[E[(\lambda x.t)\ V]] &\mapsto D[E[t\{V/x\}]] \\
D[E[V\,\mathbf{handle}\,\widehat{\alpha}\ z \Rightarrow u]] &\mapsto D[E[V]] \\
D[E[(D'_{\widehat{\alpha}}[E'[\mathsf{raise}\,\widehat{\alpha}\ V]])\,\mathbf{handle}\,\widehat{\alpha}\ z \Rightarrow u]] &\mapsto D[E[u\{V/z\}]]
\end{aligned}$$

9.3 Delimited control and the CPS hierarchy

As we saw in Section 9.1, delimited control, and specifically shift and reset, can be used to encode any monadic effect in direct style. However, having only one delimiter makes it awkward to layer multiple independent effects on top of one another in this fashion. For instance, when implementing an efficient, direct-style backtracking search with delimited control, it is useful to distinguish the following two kinds of delimiters:

1. A delimiter that marks a decision point so that when searching the current branch fails, the program backtracks to the most recent decision and tries an alternate choice.
2. A delimiter that marks the beginning of the entire search process so that when a valid solution is found, the program jumps all the way back to the start (skipping all the intermediate decision delimiters) and returns the answer, effectively cutting off all other potential branches that have not yet been considered.

Extending our basic language of delimited control by nesting delimiters in this way gives us a hierarchy of control operators. In essence, each shift and reset is given an index indicating its place in the hierarchy so that calling shift_i may capture any reset less than i and is delimited by the nearest reset of i or greater. For instance, we have the following operational step:

$$\#_3E''[\#_2E'[\#_1E[\mathscr{S}_2V]]] \mapsto \#_3E''[\#_2V\ (\lambda x.\#_2E'[\#_1E[x]])]$$

This concept of a hierarchy of nested delimiters and control operators was first introduced by Danvy & Filinski (1990) as the CPS hierarchy as well as by Sitaram & Felleisen (1990a). The essential insight is that instead of limiting the CPS transform of delimited

control to only one meta-continuation, we can iterate the CPS transform further to allow for two, three, or more nested meta-continuations. This way a program that uses at most shift_n and reset_n will be transformed to a CPS program with n meta-continuations. Then in the resulting CPS program, shift_i captures the first i (meta-)continuations, while reset_i pushes the first i (meta-)continuations into the next one.

Approaching the semantics of hierarchical delimited control as an iterated CPS transform requires us to know how many meta-continuations are necessary to determine a bound on the iteration. Or, in other words, we must know *a priori* the highest index for shift_i and reset_i that a program uses in order to translate it. Materzok & Biernacki (2012) showed that there is an alternate approach to translating hierarchical delimited control which does not need such a bound. Instead, the family of indexed control operators is all compiled down to shift_0 and reset_0, which can access arbitrarily deep into the stack of meta-continuations.

In their encoding, Materzok & Biernacki (2012) use the standard shift_0 operator, but introduce a revised version of the delimiter as the \$ operator. This revision is equally as expressive as shift_0 and reset_0, but allows for more readable terms. This operator can be encoded using shift_0 and reset_0, which we can then encode into $\lambda\mu\widehat{\mathsf{tp}}_0$ from Section 7:

$$\begin{aligned} t\,\$\,u &= \mathbf{let}\,k = t\,\mathbf{in}\,\#_0(\mathbf{let}\,x = u\,\mathbf{in}\,\mathscr{S}_0(\lambda\_.k\;x)) \\ &= \mathbf{let}\,k = t\,\mathbf{in}\,(\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}](\mathbf{let}\,x = u\,\mathbf{in}\,\mu\_.[\widehat{\mathsf{tp}}]_0 k\;x)) \end{aligned}$$

The expression $t\,\$\,u$ first evaluates t and binds the resulting value to k. Then u is evaluated in a new context and the resulting value is bound to x. Finally the delimiter is removed and the function k is applied to the result x. We can recover the operator by using the identity function, which is the functional representation of an empty context,

$$\#_0 t = (\lambda x.x)\,\$\,t$$

This alternate encoding of reset_0 reduces to the same CPS transform as the encoding from Figure 23, using either the simple single-pass CPS transform $\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}_0}$ (Figure 43) or the double CPS transforms using environments or functions, $\mathscr{C}^{\times}_{\lambda\mu\widehat{\mathsf{tp}}_0}$ (Figure 44) or $\mathscr{C}^{2}_{\lambda\mu\widehat{\mathsf{tp}}_0}$ (Figure 49), respectively,

$$\begin{aligned} (\lambda x.x)\,\$\,t &\twoheadrightarrow \mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathbf{let}\,x = t\,\mathbf{in}\,\mu\_.[\widehat{\mathsf{tp}}]_0 x \\ \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathbf{let}\,x = t\,\mathbf{in}\,\mu\_.[\widehat{\mathsf{tp}}]_0 x]\!] &\twoheadrightarrow \mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]t]\!] \\ &= \lambda k.\mathscr{C}_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![t]\!]\;(\lambda x.\lambda k'.k'\;x)\;k \end{aligned}$$

When evaluating a term of the form $V\,\$\,t$, one of the two things may happen. If t reduces to a value, then we just plug this value into the continuation V through function application

$$\begin{aligned} V\,\$\,V' &= \mathbf{let}\,k = V\,\mathbf{in}\,\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathbf{let}\,x = V'\,\mathbf{in}\,\mu\_.[\widehat{\mathsf{tp}}]_0 k\;x \\ &\twoheadrightarrow \mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mu\_.[\widehat{\mathsf{tp}}]_0 V\;V' \\ &\to \mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]_0 V\;V' \\ &\to V\;V' \end{aligned}$$

Otherwise, if evaluating t leads to a shift_0, then the context up to and including V is captured,

$$\begin{aligned} V\,\$\,E[\mathscr{S}_0 V'] &\twoheadrightarrow \mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathbf{let}\,x = E[\mu\alpha.[\widehat{\mathsf{tp}}]_0 V'\ (\lambda y.\mu_0\widehat{\mathsf{tp}}.[\alpha]y)]\,\mathbf{in}\,\mu\_.[\widehat{\mathsf{tp}}]_0 V\ x \\ &\twoheadrightarrow \mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]_0 V'\ (\lambda y.\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathbf{let}\,x = E[y]\,\mathbf{in}\,\mu\_.[\widehat{\mathsf{tp}}]_0 V\ x) \\ &\to V'\ (\lambda y.\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathbf{let}\,x = E[y]\,\mathbf{in}\,\mu\_.[\widehat{\mathsf{tp}}]_0 V\ x) \\ &= V'\ (\lambda y.V\,\$\,E[y]) \end{aligned}$$

With the above encodings of $\mathscr{S}_0$ and \$, we can give an encoding of the hierarchical family of delimited control operators $\mathscr{S}_i$ and $\#_i$ in $\lambda\mu\widehat{\mathsf{tp}}_0$.

$$\begin{aligned} \mathscr{S}_i &= \lambda h.\mathscr{S}_0\lambda k_1.\ldots.\mathscr{S}_0\lambda k_i.\#_0.\overset{i}{\ldots}.\#_0(h\ k') \\ &\quad \mathbf{where}\,k' = \lambda x.\mathscr{S}_0\lambda q_1.\ldots.\mathscr{S}_0\lambda q_{i+1}.(\lambda w.q_{i+1}\,\$\ldots\$\,q_1\,\$\,w)\,\$\,k_i\,\$\ldots\$\,k_1\,\$\,x \\ \#_i t &= \mathscr{S}_0\lambda q_1.\mathscr{S}_0\lambda q_{i+1}.(\lambda w.q_{i+1}\,\$\ldots\$\,q_1\,\$\,w)\,\$\,\#_0.\overset{i}{\ldots}.\#_0 t \end{aligned}$$

In the expression $\mathscr{S}_i V$, the $\mathscr{S}_i$ operator captures i of its surrounding contexts and replaces them with i empty contexts. The captured contexts are all bundled together in a single function that, when applied to a value, pushes the current i contexts into the next $i+1$ meta-context and then replaces them with the captured contexts. On the other hand, $\#_i t$ pushes i of its surrounding contexts into the $i+1$ meta-context and replaces them with empty contexts.

It is also informative to consider the simplest form of this encoding when i is 1. In this case, we get back encodings for the basic, non-hierarchical shift and reset operators.

$$\begin{aligned} \mathscr{S} &= \lambda h.\mathscr{S}_0\lambda k.\#_0(h\ k') \\ &\quad \mathbf{where}\,k' = \lambda x.\mathscr{S}_0\lambda q_1.\mathscr{S}_0\lambda q_2.(\lambda w.q_2\,\$\,q_1\,\$\,w)\,\$\,k\,\$\,x \\ \#t &= \mathscr{S}_0\lambda q_1.\mathscr{S}_0\lambda q_2.(\lambda w.q_2\,\$\,q_1\,\$\,w)\,\$\,\#_0 t \end{aligned}$$

While this encoding of shift and reset operators in terms of shift_0 appears more complicated than the folklore encoding, it does exhibit some desirable properties. For instance, using this encoding of reset, the idempotence of reset holds in the transform $\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}$:

$$\mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![\#\#t]\!] =_{\beta\eta} \mathscr{C}^2_{\lambda\mu\widehat{\mathsf{tp}}_0}[\![\#t]\!]$$

Whereas reset_0 is not idempotent, since repeated calls to shift_0 are able to observe the number of surrounding reset_0s. Additionally, this encoding of reset in $\lambda\mu\widehat{\mathsf{tp}}_0$ suggests a way to encode $\mu\widehat{\mathsf{tp}}.c$ from $\lambda\mu\widehat{\mathsf{tp}}$.

$$\mu\widehat{\mathsf{tp}}.c = \mu\alpha.[\widehat{\mathsf{tp}}]_0(\mathbf{let}\,x = \mu_0\widehat{\mathsf{tp}}.c\,\mathbf{in}\,\mu_0\widehat{\mathsf{tp}}.[\alpha]x)$$

Note that in order to evaluate a term t using this encoding, it is important to provide the appropriate number of bindings for $\widehat{\mathsf{tp}}$. That is, if a term t uses hierarchical delimited control operators up to shift_n and reset_n, then execution should be initialized with the following meta-command, with n repetitions of $\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\Box$:

$$[\circledast][\widehat{\mathsf{tp}}](\mu_0\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}].\overset{n}{\ldots}.(t))$$

This is necessary because even the simple term $\#_1 V$ uses shift_0, and thus will get stuck if there are too few bindings of $\widehat{\mathsf{tp}}$ in its context. Initializing the program this way corresponds to providing the correct number of (meta-)continuations to the n-fold CPS program.

9.4 Delimited control as state

With multiple dynamic co-variables, we can extend our encoding of state with multiple mutable cells. Generalizing the delimited **alloc** region to allocate a named mutable cell is achieved by binding the appropriate dynamic co-variable rather than $\widehat{\mathsf{tp}}$.

$$\mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,t = (\mu_0\widehat{\alpha}.[\widehat{\alpha}](\lambda x.\lambda s.(x,s))\ t)\ V$$

Extending get and put to handle multiple cells, however, is a more delicate operation. Not only does the cell with the matching name has to be found, we also need to be careful to not disrupt any other mutable cells that may have been allocated more recently. Therefore, in the encoding of $\mathsf{get}\,\widehat{\alpha}$ and $\mathsf{put}\,\widehat{\alpha}$, we have to remember any cells we skipped over when searching for the α cell, and then put them back in place after manipulating the α cell. This ability to manipulate bindings stored in the meta-context is provided by Dybvig *et al.*'s (2007) monadic framework for delimited control, and can be expressed in the $\lambda\widehat{\mu}_0$-calculus,

$$\begin{aligned}\mathsf{get}\,\widehat{\alpha} &= \mu\beta.[\widehat{\alpha}]_0\Delta.(\lambda s.(\mu_0\widehat{\alpha}.[\Delta][\beta]s)\ s)\\ \mathsf{put}\,\widehat{\alpha} &= \lambda s'.\mu\beta.[\widehat{\alpha}]_0\Delta.(\lambda\_.(\mu_0\widehat{\alpha}.[\Delta][\beta]())\ s')\end{aligned}$$

With a bit of calculation, we can show that **alloc**, get, and put still behave as expected in the one-cell case,

$$\begin{aligned}&\mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,E[\mathsf{get}\,\widehat{\alpha}]\\ &= (\mu_0\widehat{\alpha}.[\widehat{\alpha}](\lambda x.\lambda s.(x,s))\ E[(\mu\beta.[\widehat{\alpha}]_0\Delta.(\lambda s.(\mu_0\widehat{\alpha}.[\Delta][\beta]s)\ s))])\ V\\ &\twoheadrightarrow (\mu_0\widehat{\alpha}.[\widehat{\alpha}]_0\Delta.(\lambda s.(\mu_0\widehat{\alpha}.[\Delta][\widehat{\alpha}](\lambda x.\lambda s.(x,s))\ E[s])\ s))\ V\\ &\twoheadrightarrow (\lambda s.(\mu_0\widehat{\alpha}.[\Delta][\widehat{\alpha}](\lambda x.\lambda s.(x,s))\ E[s])\ s)\ V\\ &\twoheadrightarrow \mu_0\widehat{\alpha}.[\Delta][\widehat{\alpha}](\lambda x.\lambda s.(x,s))\ E[V])\ V\\ &= \mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,E[V]\end{aligned}$$

In this case, where the state cell we are looking for is the most recently allocated one, we don't need the full generality of $\lambda\widehat{\mu}_0$, the simpler $\lambda\widehat{\mu}$ would do. However, when we are looking for an older state cell that is not the most recently allocated one, the simple lookup of $\lambda\widehat{\mu}$ is insufficient to represent state, since it would forget all the intermediate allocations. While looking for a particular state cell, we are obligated to remember all other allocations, which requires us to re-bind the more recent allocations found during lookup. For instance,

we have the following reduction with multi-cell state:

$$\begin{aligned}
&\mathbf{alloc}\,\widehat{\beta} := V\,\mathbf{in}\,E[\mathsf{get}\,\widehat{\alpha}] \\
&= (\mu_0\widehat{\beta}.[\widehat{\beta}](\lambda x.\lambda s'.(x,s'))\ (E[\mu\alpha.[\widehat{\alpha}]_0\Delta.(\lambda s.(\mu_0\widehat{\alpha}.[\Delta][\alpha]s)\ s)]))\ V \\
&\twoheadrightarrow (\mu_0\widehat{\beta}.[\widehat{\alpha}]_0\Delta.(\lambda s.(\mu_0\widehat{\alpha}.[\Delta][\widehat{\beta}](\lambda x.\lambda s'.(x,s'))\ E[s])\ s))\ V \\
&\twoheadrightarrow \mu\beta.[\widehat{\alpha}]_0\Delta.(\lambda s.(\mu_0\widehat{\alpha}.[\Delta][\beta](\mu_0\widehat{\beta}.[\widehat{\beta}](\lambda x.\lambda s'.(x,s'))\ E[s])\ V)\ s) \\
&= \mu\beta.[\widehat{\alpha}]_0\Delta.(\lambda s.(\mu_0\widehat{\alpha}.[\Delta][\beta]\mathbf{alloc}\,\widehat{\beta} := V\,\mathbf{in}\,E[s])\ s))
\end{aligned}$$

which demonstrates how the intermediate allocation of the $\widehat{\beta}$ state cell is remembered during access to the $\widehat{\alpha}$ cell. Once the value V allocated in the cell $\widehat{\alpha}$ is found, the $\widehat{\beta}$ cell will be restored before continuing on with the evaluation of $E[V]$. Finally, we use a sequentialization operation, $t_1;t_2$, as a convenience for working with stateful operations. This operation is defined as syntactic sugar by using the sequentializing behavior of call-by-value function application:

$$t_1;t_2 = (\lambda\_.t2)\ t_1$$

In general, we can derive a reduction semantics for delimited, multi-cell state. The (meta-)evaluation contexts are defined as:

$$E ::= \square \mid E\ t \mid V\ E \mid \mathsf{put}\,\widehat{\alpha}\ E \mid E;t \qquad\qquad D ::= \square \mid E[\mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,D]$$

In terms of these (meta-)evaluation contexts, the coarse-grained reduction rules for state are:

$$\begin{aligned}
\mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,V' &\twoheadrightarrow (V',V) \\
\mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,D_{\widehat{\alpha}}[E[\mathsf{get}\,\widehat{\alpha}]] &\twoheadrightarrow \mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,D_{\widehat{\alpha}}[E[V]] \\
\mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,D_{\widehat{\alpha}}[E[\mathsf{put}\,\widehat{\alpha}\ V']] &\twoheadrightarrow \mathbf{alloc}\,\widehat{\alpha} := V'\,\mathbf{in}\,D_{\widehat{\alpha}}[E[()]] \\
V;t &\twoheadrightarrow t
\end{aligned}$$

where $D_{\widehat{\alpha}}$ does not contain an allocation of $\widehat{\alpha}$. Additionally, the complete operational semantics is given by only applying reduction rules in the evaluation contexts described by $D[E]$:

$$\begin{aligned}
D[E[(\lambda x.t)\ V]] &\mapsto D[E[t\{V/x\}]] \\
D[E[\mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,V']] &\mapsto D[E[(V',V)]] \\
D[E[\mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,D'_{\widehat{\alpha}}[E'[\mathsf{get}\,\widehat{\alpha}]]]] &\mapsto D[E[\mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,D'_{\widehat{\alpha}}[E'[V]]]] \\
D[E[\mathbf{alloc}\,\widehat{\alpha} := V\,\mathbf{in}\,D'_{\widehat{\alpha}}[E'[\mathsf{put}\,\widehat{\alpha}\ V']]]] &\mapsto D[E[\mathbf{alloc}\,\widehat{\alpha} := V'\,\mathbf{in}\,D'_{\widehat{\alpha}}[E'[()]]]] \\
D[E[V;t]] &\mapsto D[E[t]]
\end{aligned}$$

Example 15

To illustrate how these encoded effects interact with one another, we show the behavior of a program that uses both mutable state and exceptions, as defined in Sections 9.4 and 9.2, respectively. The reduction theory and operational semantics for the combination of these two effects are given as the simple union of independent semantics, which is validated by the encodings of state and exceptions into the $\lambda\widehat{\mu}_0$-calculus. First, we show the typical

$$
\begin{array}{l}
\mathbf{alloc}\,\widehat{\alpha} := 10\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in} \\
\Big(\Big(\mathsf{put}\,\widehat{\alpha}\;(\mathsf{get}\,\widehat{\alpha} * \mathsf{get}\,\widehat{\beta}); \mathsf{raise}\,\widehat{\varepsilon}\;(\mathsf{get}\,\widehat{\alpha})\Big)\;\mathbf{handle}\,\widehat{\varepsilon}\;x \Rightarrow x + \mathsf{get}\,\widehat{\alpha}\Big) \\
\longmapsto \begin{array}{l}\mathbf{alloc}\,\widehat{\alpha} := 10\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in} \\ ((\mathsf{put}\,\widehat{\alpha}\;(10*2); \mathsf{raise}\,\widehat{\varepsilon}\;(\mathsf{get}\,\widehat{\alpha}))\;\mathbf{handle}\,\widehat{\varepsilon}\;x \Rightarrow x + \mathsf{get}\,\widehat{\alpha})\end{array} \\
\longmapsto \begin{array}{l}\mathbf{alloc}\,\widehat{\alpha} := 20\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in} \\ ((\mathsf{raise}\,\widehat{\varepsilon}\;(\mathsf{get}\,\widehat{\alpha}))\;\mathbf{handle}\,\widehat{\varepsilon}\;x \Rightarrow x + \mathsf{get}\,\widehat{\alpha})\end{array} \\
\longmapsto \begin{array}{l}\mathbf{alloc}\,\widehat{\alpha} := 20\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in} \\ ((\mathsf{raise}\,\widehat{\varepsilon}\;20)\;\mathbf{handle}\,\widehat{\varepsilon}\;x \Rightarrow x + \mathsf{get}\,\widehat{\alpha})\end{array} \\
\longmapsto \mathbf{alloc}\,\widehat{\alpha} := 20\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in}\;(20 + \mathsf{get}\,\widehat{\alpha}) \\
\longmapsto \mathbf{alloc}\,\widehat{\alpha} := 20\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in}\;(20 + 20) \\
\longmapsto \mathbf{alloc}\,\widehat{\alpha} := 20\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in}\,40
\end{array}
$$

Fig. 65. Evaluation of a program using both mutable state and exceptions.

case, where the mutable cells are stored globally, 'outside' the program, and are unaffected by the flow of control. We model this behavior by listing all of the allocations of mutable cells in the outermost part of the program. The example given in Figure 65 shows that the values in the mutable cells are not affected by raising an exception. However, the notion of delimited mutable state given here is more flexible, and allows for the allocation of mutable cells to exist within the program itself. In this case, internal allocations are subject to alterations of control flow in the program. Example in Figure 66 shows the same program with an internal re-allocation of the $\widehat{\alpha}$ mutable cell. In this program the internal allocation of $\widehat{\alpha}$ overrides the external one until an exception is raised. At that point the internal allocation of $\widehat{\alpha}$ is discarded, and the external one is used. This back-tracking of mutable cells on an exception gives a result of 210 instead of 400. The difference between allocations that are external or internal to an exception is similar to the different layerings of the StateT and ErrorT monad transformers in Haskell. An allocated cell outside the bounds of an exception behaves like the usual notion of global state, and corresponds to the Haskell monad given by $\mathsf{ErrorT}\,e\;(\mathsf{StateT}\,s\;m)\;a$. On the other hand, an allocated cell inside the bounds of an exception is discarded, and corresponds to the monad given by $\mathsf{StateT}\,s\;(\mathsf{ErrorT}\,e\;m)\;a$.

10 Conclusions

By now we have explored a series of increasingly expressive languages, and demonstrated some computational effects expressible in each language. We started with the pure, call-by-value λ-calculus as our foundation and extended it to Parigot's $\lambda\mu$ calculus, giving us classical control and the call/cc control operator. We then added a re-bindable top-level to the $\lambda\mu$ calculus, giving us the $\lambda\mu\widehat{\mathsf{tp}}$ calculus of delimited control and allowing us to represent any monadic effect as a first-class computational effect (Filinski 1994) such as exceptions and mutable state. From there we extended the language with multiple, named, re-bindable top-levels as a simple extension, $\lambda\widehat{\mu}$, which allowed us to express exception handling with more than one named exception. We also considered a variant of

$$\begin{array}{l}
\mathbf{alloc}\,\widehat{\alpha} := 10\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in} \\
\left(\left(\begin{array}{l}\mathbf{alloc}\,\widehat{\alpha} := 100\,\mathbf{in} \\ \mathsf{put}\,\widehat{\alpha}\;(\mathsf{get}\,\widehat{\alpha} * \mathsf{get}\,\widehat{\beta}); \mathsf{raise}\,\widehat{\varepsilon}\;(\mathsf{get}\,\widehat{\alpha})\end{array}\right)\mathbf{handle}\,\widehat{\varepsilon}\;x \Rightarrow x + \mathsf{get}\,\widehat{\alpha}\right) \\
\\
\quad \mathbf{alloc}\,\widehat{\alpha} := 10\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in} \\
\mapsto\!\!\!\!\mapsto \left(\left(\begin{array}{l}\mathbf{alloc}\,\widehat{\alpha} := 100\,\mathbf{in} \\ \mathsf{put}\,\widehat{\alpha}\;(100 * 2); \mathsf{raise}\,\widehat{\varepsilon}\;(\mathsf{get}\,\widehat{\alpha})\end{array}\right)\mathbf{handle}\,\widehat{\varepsilon}\;x \Rightarrow x + \mathsf{get}\,\widehat{\alpha}\right) \\
\\
\quad \mathbf{alloc}\,\widehat{\alpha} := 10\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in} \\
\mapsto\!\!\!\!\mapsto \left(\left(\begin{array}{l}\mathbf{alloc}\,\widehat{\alpha} := 200\,\mathbf{in} \\ \mathsf{raise}\,\widehat{\varepsilon}\;(\mathsf{get}\,\widehat{\alpha})\end{array}\right)\mathbf{handle}\,\widehat{\varepsilon}\;x \Rightarrow x + \mathsf{get}\,\widehat{\alpha}\right) \\
\\
\quad \mathbf{alloc}\,\widehat{\alpha} := 10\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in} \\
\mapsto \left(\left(\begin{array}{l}\mathbf{alloc}\,\widehat{\alpha} := 200\,\mathbf{in} \\ \mathsf{raise}\,\widehat{\varepsilon}\;200\end{array}\right)\mathbf{handle}\,\widehat{\varepsilon}\;x \Rightarrow x + \mathsf{get}\,\widehat{\alpha}\right) \\
\\
\mapsto \mathbf{alloc}\,\widehat{\alpha} := 10\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in}\;(200 + \mathsf{get}\,\widehat{\alpha}) \\
\mapsto \mathbf{alloc}\,\widehat{\alpha} := 10\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in}\;(200 + 10) \\
\mapsto \mathbf{alloc}\,\widehat{\alpha} := 10\,\mathbf{in}\,\mathbf{alloc}\,\widehat{\beta} := 2\,\mathbf{in}\,210
\end{array}$$

Fig. 66. Interaction between delimiters for mutable state and exceptions.

delimited control with dynamic access to prompts, $\lambda\mu\widehat{\mathsf{tp}}_0$, which allowed us to express an unbounded hierarchy of nested control operators (Materzok & Biernacki 2012). Finally, we brought the two back together to achieve a more powerful language of delimited control with multiple prompts, $\lambda\widehat{\mu}_0$, where we can remember and restore the state of different prompts. This allowed us to implement mutable state with multiple stateful cells in a way that preserves the heap while accessing an allocated cell.

For each of the languages of control, we have described their semantics in a variety of ways, and each semantic artifact highlights a crucial aspect of the behavior of the languages. The reduction semantics can be applied anywhere in a program, such as underneath a λ-abstraction, and only analyzes a small portion of the program at a time. These reduction rules capture optimizations that a compiler may perform, since the rules are applied to small fragments of the source program and are valid outside the normal evaluation order. The operational semantics provides an intuitive description of standard evaluation in terms of the source language, and shows the complete behavior of each language construct in its evaluation context. At a more low-level, the CPS transform gives a compilation from the source language to some target language such as the pure λ-calculus or one of the languages of dynamic binding. With the compositional transforms presented here, this lets us use the target language as a tool for reasoning about the source language independent of

context. Alternatively, an abstract machine gives a tail-recursive interpreter for the source language and forms a bridge for more practical implementations of the language.

We have provided a calculus that 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.* (2006) 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 to 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.

The languages of control given here provide a framework for describing high-level control operators using a variety of semantic tools. The control operations are given as fine-grained primitives that can express many higher-level control operators. Consider again the questions raised in Example 2:

$$(\mathsf{reset}[(\mathsf{abort}(\mathsf{raise}\,0) * 5)\,\mathbf{handle}\, n \Rightarrow n+1])$$
$$\mathbf{handle}\, n \Rightarrow n+2$$

By giving a definition of the abort and raise operators in terms of $\lambda\widehat{\mu}_0$, we can specify the precise behavior that we intend, including finer details of the semantics such as when portions of the context are captured or cleared. Suppose that we take the definition of exception handling in Section 9.2, using a default name for the dynamic co-variable that is different from $\widehat{\mathsf{tp}}$. If we define abort as:

$$\mathsf{abort} = \lambda x.\mu\_.[\widehat{\mathsf{tp}}]x$$

then due to the call-by-value semantics of function application, first $\mathsf{raise}\,0$ is evaluated, clearing the context $(\mathsf{abort}\,\Box) * 5$ and invoking the innermost exception handler and giving the result 1. If we define abort as:

$$\mathsf{abort}\,t = \mu\_.[\widehat{\mathsf{tp}}]t$$

then first abort takes control and clears the context $\Box * 5$, but then evaluates $\mathsf{raise}\,0$ still inside the innermost handler and also gives the result 1. If we instead define abort as:

$$\mathsf{abort}\,t = \mu\_.[\widehat{\mathsf{tp}}]_0\_.t$$

then abort clears the context $\Box * 5$ and also removes the innermost handler while searching for reset. After abort finds and removes the reset, $\mathsf{raise}\,0$ is evaluated where the reset was, thereby invoking the outermost handler and giving the result 2. Each of these choices for abort has different semantics, and the difference in behavior is clearly expressed by different encodings in $\lambda\widehat{\mu}_0$.

We have seen how delimited control with multiple prompts can be used to represent a variety of computational effects, in the style of Filinski (1994, 1999), so that they can be combined without interference within a single program. This can be used to combine different effects such as exceptions and short-circuiting abort in Example 2, or to extend a single effect such as exception handling with multiple named exception in Section 9.2 or mutable state with multiple cells in Section 9.4. By giving the control operations as a collection of well-behaved, and fine-grained primitives, we can express a wide range of

different behaviors for our chosen high-level control operators, and use the same set of semantic tools for reasoning about each. The $\lambda\mu$, $\lambda\mu\widehat{\mathtt{tp}}$, and $\lambda\mu\widehat{\mathtt{tp}}_0$ calculi each contain a small number of control primitives with basic semantics. However, it is unclear if the $\lambda\widehat{\mu}_0$ calculus can be broken down into even simpler primitives in the style of $\lambda\mu\widehat{\mathtt{tp}}_0$ *et al.* Such a development could show how named prompts can be extended to a hierarchy, integrating both delimited control with multiple named prompts and the CPS hierarchy, which we leave as a future direction of our work.

Acknowledgments

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

Supplementary materials

For supplementary materials for this article, please visit dx.doi.org/10.1017/S0956796813000312.

References

Ariola, Z. M. & Herbelin, H. (2008) Control reduction theories: The benefit of structural substitution. *J. Funct. Program.* **18**(May), 373–419.

Ariola, Z. M., Herbelin, H., Herman, D. & Keith, D. (2011) A robust implementation of delimited control. In *First International Workshop on the Theory and Practice of Delimited Continuations*, Novi Sad, Serbia, p. 6.

Ariola, Z. M., Herbelin, H. & Sabry, A. (2009) A type-theoretic foundation of delimited continuations. *Higher Order Symb. Comput.* **22**(3), 233–273.

Biernacka, M., Biernacki, D. & Danvy, O. (2005) An operational foundation for delimited continuations in the CPS hierarchy. *Log. Methods Comput. Sci.* **1**(2), 1–39.

Danvy, O. (2004) On evaluation contexts, continuations, and the rest of the computation. In *ACM Sigplan Continuations Workshop*, pp. 13–23.

Danvy, O. & Filinski, A. (1989) *A Functional Abstraction of Typed Contexts*. Tech. rept. 89/12. DIKU, University of Copenhagen, Copenhagen, Denmark.

Danvy, O. & Filinski, A. (1990) Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. Pittsburgh, PA: ACM Press, pp. 151–160.

Downen, P. & Ariola, Z. M. (2012) A systematic approach to delimited control with multiple prompts. In *Procedings of the 21st European Symposium on Programming*. Berlin, Germany: Springer, pp. 234–253.

Dybvig, R. K., Jones, S. P. & Sabry, A. (2007) A monadic framework for delimited continuations. *J. Funct. Program.* **17**(6), 687–730.

Felleisen, M. (1988) The theory and practice of first-class prompts. In *Principles of Programming Languages '88*, pp. 180–190.

Felleisen, M. (1991) On the expressive power of programming languages. *Sci. Comput. Program.* **17**(1–3), 35–75.

Felleisen, M. & Friedman, D. P. (1987) A reduction semantics for imperative higher-order languages. In *Parallel Architectures and Languages Europe (PARLE)*, Lecture Notes in Computer Science, vol. 259. Berlin, Germany: Springer, pp. 206–223.

Felleisen, M., Friedman, D. P., Kohlbecker, E. E. & Duba, B. F. (1987) A syntactic theory of sequential control. *Theor. Comput. Sci.* **52**, 205–237.

Felleisen, M. & Hieb, R. (1992) The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.* **103**(2), 235–271.

Felleisen, M., Wand, M., Friedman, D. P. & Duba, B. F. (1988) Abstract continuations: A mathematical semantics for handling full jumps. In *LISP and Functional Programming*, Snowbird, UT, pp. 52–62.

Filinski, A. (1994) Representing monads. In *Principles of Programming Languages '94*. Pittsburgh, PA: ACM, pp. 446–457.

Filinski, A. (1999) Representing layered monads. In *Principles of Programming Languages '99*, pp. 175–188.

Flatt, M., Yu, G., Findler, R. B. & Felleisen, M. (2007) Adding delimited and composable control to a production programming environment. In *Proceedings of the 12th ACM Sigplan International Conference on Functional Programming*, vol. 1., pp. 165–176.

Gunter, C. A., Rémy, D. & Riecke, J. G. (1995) A generalization of exceptions and control in ML-like languages. In *Functional Programming Languages and Computer Architecture '95*. New York, NY: ACM, pp. 12–23.

Kameyama, Y. & Hasegawa, M. (2003) A sound and complete axiomatization of delimited continuations. In *Proceedings of the Eighth ACM Sigplan International Conference on Functional Programming (ICFP '03)*. New York, NY: ACM, pp. 177–188.

Kiselyov, O. (2010) Delimited control in OCaml, abstractly and concretely: System description. In *Functional and Logic. Programming*, Lecture Notes in Computer Science vol. 6009. New York, NY: Springer, 304–320.

Kiselyov, O., Shan, C.-C. & Sabry, A. (2006) Delimited dynamic binding. In *Proceedings of the Eleventh ACM Sigplan International Conference on Functional Programming (ICFP '06)*. New York, NY: ACM, pp. 26–37.

Materzok, M. & Biernacki, D. (2011) Subtyping delimited continuations. In *Proceeding of the 16th ACM Sigplan International Conference on Functional Programming (ICFP '11)*. New York, NY: ACM, pp. 81–93.

Materzok, M. & Biernacki, D. (2012) A dynamic interpretation of the CPS hierarchy. In *10th Asian Symposium on Programming Languages and Systems (APLAS 2012)*, pp. 296–311.

Moggi, E. (1989) Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science* (IEEE), pp. 14–23.

Moggi, E. (1991) Notions of computation and monads. *Inf. Comput.* **93**(1), 55–92.

Moreau, L. (1998) A syntactic theory of dynamic binding. *Higher Order Symb. Comput.* **11**(3), 233–279.

Parigot, M. (1992) Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning*. New York, NY: Springer, pp. 190–201.

Plotkin, G. D. (1975) Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.* **1**, 125–159.

Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference.* New York, NY: ACM, pp. 717–740.

Shan, C.-C. (2007) A static simulation of dynamic delimited control. *Higher Order Symb. Comput.* **20**(4), 371–401.

Sitaram, D. & Felleisen, M. (1990a) Control delimiters and their hierarchies. *LISP Symb. Comput.* **3**(1), 67–99.

Sitaram, D. & Felleisen, M. (1990b) Reasoning with continuations II: Full abstraction for models of control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. New York, NY: ACM, pp. 161–175.