

Delimited control and computational effects

PAUL DOWNEN and ZENA M. ARIOLA

University of Oregon

(*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 the various languages, we show how other computational effects can be encoded in terms of these control operators.

Keywords: control calculi, delimited control, multiple prompts, dynamic binding, reduction theory, continuation-passing style

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. *call/cc* allows the programmer to capture the surrounding context of an expression, creating a continuation that serves as a return point to “the rest of the program” from where *call/cc* was called. This style of control abstraction is called *abortive*, since invoking a continuation captured by *call/cc* aborts the computation currently in progress, and immediately returns to the context stored in the continuation. Even though *call/cc* is a very flexible control operator, it has limits. For example, *call/cc* alone is not enough to simulate mutable state in an otherwise state-free language.

Compared to abortive control, delimited control provides a more powerful abstraction. The difference of delimited control is that the continuation behaves like a normal function, so that multiple continuations may be composed together. In addition, the scope of the control operator can be managed by setting a *prompt*, limiting the context that can be captured. 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 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 as well as the scope of exception handlers leads to interference between the two.

Example 1

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

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

The expression `raise0` would be “caught” by the `reset`, rather than by the exception handler, giving the wrong result 0.

The solution used by Flatt *et al.* is to use multiple named prompts. When `reset` and `handle` both set differently named prompts, and `raise` refers to the prompt set by `handle`, then 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:

$$\begin{aligned} &(\text{reset}[(\text{abort}(\text{raise}0) * 5) \mathbf{handle} n \Rightarrow n + 1]) \\ &\mathbf{handle} n \Rightarrow n + 2 \end{aligned}$$

Does `abort` first evaluate `raise0` before jumping to the reset, thus raising the exception inside of the innermost handler and returning the result 1? Or instead does `abort` first jump to the reset and then evaluate `raise0`, 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 $\lambda\mu\hat{\tau}\hat{p}$. We use $\lambda\mu\hat{\tau}\hat{p}$ as a reference point since it has a well-understood reduction theory that directly expresses the operational semantics. By extending $\lambda\mu\hat{\tau}\hat{p}$ 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:

- 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 $\lambda\mu\hat{\tau}\hat{p}$, 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 language 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 $\lambda\mu\hat{\tau}\hat{p}$.

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 ($\lambda\mu\hat{\tau}\hat{p}$ in Section 4). Then, we branch out in two separate directions, extending $\lambda\mu\hat{\tau}\hat{p}$ with multiple static prompts ($\lambda\hat{\mu}$ in Section 6) and also a single dynamic prompt ($\lambda\mu\hat{\tau}\hat{p}_0$ in Section 7). Finally, we bring $\lambda\hat{\mu}$ and $\lambda\mu\hat{\tau}\hat{p}_0$ together, giving us a language of delimited control with multiple dynamic prompts ($\lambda\hat{\mu}_0$ 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, like 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 the semantics. The reduction rules are a set of local program transformations in the source language that correspond to reductions performed in the CPS transformed program. The operational semantics arise as both a restriction on the reduction rules and as the equivalent small-step evaluator for the CPS transformation. The abstract machine captures the same essential steps as the CPS transform, since it is derived by defunctionalizing the CPS transform (Reynolds, 1972;

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).

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

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

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

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

Danvy, 2004). We however present both, since some readers may find the abstract machine easier to follow.

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 $\lambda\mu\hat{\text{t}}\hat{\text{p}}$, 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 the 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 of 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

$$E \in \text{EvCtx} ::= \square \mid E \ t \mid V \ E$$

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

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

$$\begin{aligned} \mathcal{C}_\lambda[V] &= \lambda k.k \ \mathcal{C}_\lambda[V]^V & \mathcal{C}_\lambda[x]^V &= x \\ \mathcal{C}_\lambda[t_1 \ t_2] &= \lambda k.\mathcal{C}_\lambda[t_1] \ \lambda f.\mathcal{C}_\lambda[t_2] \ \lambda s.f \ s \ k & \mathcal{C}_\lambda[(\lambda x.t)]^V &= \lambda x.\mathcal{C}_\lambda[t] \end{aligned}$$

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

is the standard one is by giving an *evaluation context* which points out the location of the standard reduction. For a call-by-value λ -calculus, we have the set of evaluation contexts 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 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 which hard-wires the evaluation strategy into the term itself. The translation is called a *continuation passing style* (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_i is initialized as the identity function: $k_i = \lambda x.x$.

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

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

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

$$\begin{array}{ll}
\mathcal{C}_\lambda \llbracket [q]t \rrbracket = \mathcal{C}_\lambda \llbracket t \rrbracket \mathcal{C}_\lambda \llbracket q \rrbracket & \mathcal{C}_\lambda \llbracket [*] \rrbracket = \lambda x.x \\
\mathcal{C}_\lambda \llbracket V \rrbracket = \lambda k.k \mathcal{C}_\lambda \llbracket V \rrbracket^V & \mathcal{C}_\lambda \llbracket x \rrbracket^V = x \\
\mathcal{C}_\lambda \llbracket t_1 t_2 \rrbracket = \lambda k.\mathcal{C}_\lambda \llbracket t_1 \rrbracket \lambda f.\mathcal{C}_\lambda \llbracket t_2 \rrbracket \lambda s.f s k & \mathcal{C}_\lambda \llbracket \lambda x.t \rrbracket^V = \lambda x.\mathcal{C}_\lambda \llbracket t \rrbracket
\end{array}$$

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

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 in Figure 1 with two new syntactic categories, giving us the syntax shown in Figure 5. We also extend our CPS transform \mathcal{C}_λ 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 notice how this abstract machine can be viewed as a defunctionalized form of the \mathcal{C}_λ transform, using Danvy’s

$$\begin{array}{l}
S ::= \langle c \rangle_{\text{refocus}} \mid \langle t, F^* \rangle_{\text{refocus}} \mid \langle F^*, V \rangle_{\text{apply}} \mid \langle t, F^* \rangle_{\text{reduce}} \mid \langle V \rangle_{\text{done}} \\
F \in \mathit{Frame} ::= \square \mid t \mid V \mid \square \qquad F^* \in \mathit{Stack} ::= q \mid F^*[F]
\end{array}$$

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

$$\begin{array}{l}
\langle [*]t \rangle_{\text{refocus}} \rightsquigarrow \langle t, * \rangle_{\text{refocus}} \qquad \langle F^* [\square t], V \rangle_{\text{apply}} \rightsquigarrow \langle t, F^* [V \square] \rangle_{\text{refocus}} \\
\langle t t', F^* \rangle_{\text{refocus}} \rightsquigarrow \langle t, F^* [\square t'] \rangle_{\text{refocus}} \qquad \langle F^* [V' \square], V \rangle_{\text{apply}} \rightsquigarrow \langle V' V, F^* \rangle_{\text{reduce}} \\
\langle V, F^* \rangle_{\text{refocus}} \rightsquigarrow \langle F^*, V \rangle_{\text{apply}} \qquad \langle *, V \rangle_{\text{apply}} \rightsquigarrow \langle V \rangle_{\text{done}} \\
\langle (\lambda x.t) V, F^* \rangle_{\text{reduce}} \rightsquigarrow \langle t \{V/x\}, F^* \rangle_{\text{refocus}}
\end{array}$$

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

technique of deriving abstract machines from CPS transforms (Reynolds, 1972; Danvy, 2004). The refocus steps implement the definition of $\mathcal{C}_\lambda \llbracket c \rrbracket$ and β -reductions of the form $\mathcal{C}_\lambda \llbracket t \rrbracket k$, 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 $(\lambda x. \mathcal{C}_\lambda \llbracket t \rrbracket) \mathcal{C}_\lambda \llbracket V \rrbracket^V k$ that results from the translation of $\mathcal{C}_\lambda \llbracket (\lambda x.t) V \rrbracket k$.

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 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 (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 \mathcal{C}_λ 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 $\mathcal{C}_\lambda \llbracket M \rrbracket =_\beta \mathcal{C}_\lambda \llbracket M' \rrbracket$.

We would also like some guarantee that the opposite property holds, 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 $\mathcal{C}_\lambda \llbracket c \rrbracket =_\beta V$ then there is a final answer c' such that $c \mapsto c'$ and $\mathcal{C}_\lambda \llbracket c' \rrbracket =_\beta V$.

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 $c \mapsto c'$ then $c \twoheadrightarrow c'$.

Therefore, the reduction theory is also strong enough to find the same final answer as the operational semantics and 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.

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

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

$$\begin{array}{l}
(\lambda x.t) V \rightarrow t\{V/x\} \\
F[\mu\alpha.c] \rightarrow \mu\alpha.c\{[\alpha](F[t])/[\alpha]t\} \\
[q]\mu\alpha.c \rightarrow c\{q/\alpha\}
\end{array}$$

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

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 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

$$\begin{aligned}
E &::= \square \mid E \ t \mid V \ E & D &::= [q]E \\
D[(\lambda x.t) \ V] &\mapsto D[t\{V/x\}] \\
D[\mu \alpha.c] &\mapsto c\{D[t]/[\alpha]t\}
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{C}_{\lambda\mu}[[[q]t]] &= \mathcal{C}_{\lambda\mu}[[t]] \ \mathcal{C}_{\lambda\mu}[[q]] & \mathcal{C}_{\lambda\mu}[[*]] &= \lambda x.x \\
\mathcal{C}_{\lambda\mu}[[V]] &= \lambda k.k \ \mathcal{C}_{\lambda\mu}[[V]]^V & \mathcal{C}_{\lambda\mu}[[\alpha]] &= \alpha \\
\mathcal{C}_{\lambda\mu}[[t_1 \ t_2]] &= \lambda k.\mathcal{C}_{\lambda\mu}[[t_1]] \ \lambda f.\mathcal{C}_{\lambda\mu}[[t_2]] \ \lambda s.f \ s \ k & \mathcal{C}_{\lambda\mu}[[x]]^V &= x \\
\mathcal{C}_{\lambda\mu}[[\mu \alpha.c]] &= \lambda k.(\lambda \alpha.\mathcal{C}_{\lambda\mu}[[c]]) \ k & \mathcal{C}_{\lambda\mu}[[\lambda x.t]]^V &= \lambda x.\mathcal{C}_{\lambda\mu}[[t]]
\end{aligned}$$

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

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 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 \mathcal{C}_λ for definitions of the new syntax, giving us the $\mathcal{C}_{\lambda\mu}$ transform in Figure 12. The only change from the \mathcal{C}_λ 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 an additional refocus step and reduce step that implements 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 $*$. Notice 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,

$$\begin{aligned}
\langle [q]t \rangle_{\text{refocus}} &\rightsquigarrow \langle t, q \rangle_{\text{refocus}} & \langle F^*[\square t], V \rangle_{\text{apply}} &\rightsquigarrow \langle t, F^*[V \ \square] \rangle_{\text{refocus}} \\
\langle t \ t', F^* \rangle_{\text{refocus}} &\rightsquigarrow \langle t, F^*[\square t'] \rangle_{\text{refocus}} & \langle F^*[V' \ \square], V \rangle_{\text{apply}} &\rightsquigarrow \langle V' \ V, F^* \rangle_{\text{reduce}} \\
\langle \mu \alpha.c, F^* \rangle_{\text{refocus}} &\rightsquigarrow \langle \mu \alpha.c, F^* \rangle_{\text{reduce}} & \langle *, V \rangle_{\text{apply}} &\rightsquigarrow \langle V \rangle_{\text{done}} \\
\langle V, F^* \rangle_{\text{refocus}} &\rightsquigarrow \langle F^*, V \rangle_{\text{apply}} \\
\langle (\lambda x.t) \ V, F^* \rangle_{\text{reduce}} &\rightsquigarrow \langle t\{V/x\}, F^* \rangle_{\text{refocus}} \\
\langle \mu \alpha.c, F^* \rangle_{\text{reduce}} &\rightsquigarrow \langle c\{F^*/\alpha\} \rangle_{\text{refocus}}
\end{aligned}$$

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

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

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

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 $\mathcal{C}_{\lambda\mu}$ CPS transform.

Theorem 3.1 (Soundness)

If $M \twoheadrightarrow M'$ then $\mathcal{C}_{\lambda\mu} \llbracket M \rrbracket =_{\beta} \mathcal{C}_{\lambda\mu} \llbracket M' \rrbracket$.

Theorem 3.2 (Evaluation)

If $\mathcal{C}_{\lambda\mu} \llbracket c \rrbracket =_{\beta} V$ then there is a final answer c' such that $c \mapsto c'$ and $\mathcal{C}_{\lambda\mu} \llbracket c' \rrbracket =_{\beta} V$.

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 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 (\mathcal{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 (\mathcal{A}) operator and Felleisen’s \mathcal{C} operator, which is definable in terms of call/cc and abort.

4 Delimited control: $\lambda\mu\hat{\tau}$

Delimiting control means temporarily re-defining the top-level in a program, limiting the extent to which the evaluation context may be captured. Examples of delimited control operators are Felleisen’s \mathcal{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 rebindable dynamic continuation variable $\hat{\tau}$.

$$\begin{array}{ll}
t \in \text{Term} ::= V \mid t_1 t_2 \mid \mu q.c & c \in \text{Command} ::= [q]t \\
V \in \text{Value} ::= x \mid \lambda x.t & q \in \text{CoTerm} ::= \alpha \mid \widehat{\text{tp}}
\end{array}$$

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

$$\begin{array}{l}
(\lambda x.t) V \rightarrow t\{V/x\} \\
F[\mu\alpha.c] \rightarrow \mu\alpha.c\{[\alpha](F[t])/[\alpha]t\} \\
[q]\mu\alpha.c \rightarrow c\{q/\alpha\} \\
\mu\widehat{\text{tp}}.\widehat{\text{tp}}V \rightarrow V
\end{array}$$

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

The syntax of $\lambda\mu\widehat{\text{tp}}$ 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 $\widehat{\text{tp}}$, and the new μ -abstraction $\mu\widehat{\text{tp}}.c$ is added which binds $\widehat{\text{tp}}$ in c .

Reduction The dynamic nature of $\widehat{\text{tp}}$ is due to the fact that in a function like $\lambda x.\mu_.[\widehat{\text{tp}}]x$, the binding of $\widehat{\text{tp}}$ is taken from the environment active at the call site and not in the environment active when the function is defined. This dynamic nature is captured by adding the following reduction rule to the reduction theory of $\lambda\mu$ given in Figure 10:

$$\mu\widehat{\text{tp}}.\widehat{\text{tp}}V \rightarrow V$$

Notice that the renaming rule:

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

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

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

CPS transform We extend the $\mathcal{C}_{\lambda\mu}$ transform from Figure 12 to give $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}$, the CPS transform for $\lambda\mu\widehat{\text{tp}}$ shown in Figure 17. Here, $\widehat{\text{tp}}$ takes the place of the old constant $*$. However, now we also have a binding form for $\widehat{\text{tp}}$. When $\widehat{\text{tp}}$ is bound over a command, the current continuation is set aside and that command is run to completion. Then, when

$$\begin{array}{ll}
\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[[q]t] = \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[t] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[q] & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[\alpha] = \alpha \\
\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[V] = \lambda k.k \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[V]^V & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[\widehat{\text{tp}}] = \lambda x.x \\
\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[t_1 t_2] = \lambda k.\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[t_1] \lambda f.\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[t_2] \lambda s.f s k & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[x]^V = x \\
\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[\mu\alpha.c] = \lambda k.(\lambda\alpha.\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[c]) k & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[\lambda x.t]^V = \lambda x.\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[t] \\
\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[\mu\widehat{\text{tp}}.c] = \lambda k.k \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}[c] &
\end{array}$$

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

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

Fig. 18. Double CPS transform of $\hat{\text{tp}}$.

$$\begin{aligned} c^2 \in \text{Command}^2 &::= [q^2]c & q^2 \in \text{CoTerm}^2 &::= \otimes \\ \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2[[[q^2]c]] &= \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2[[c]]\ \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2[[q^2]] & \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2[[\otimes]] &= \lambda x.x \end{aligned}$$

Fig. 19. The $\lambda\mu\hat{\text{tp}}$ -calculus and $\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2$ extended with meta-commands.

the command has produced an answer value, that value is fed to the original continuation and that context is restored. Unfortunately, the above translation of $\mu\hat{\text{tp}}.c$ is not in CPS, since the term $\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2[[c]]$ is an application instead of a value. One can remedy the situation by taking the output from $\mathcal{C}_{\lambda\mu\hat{\text{tp}}}$ and running it through the CPS transform \mathcal{C}_λ (Danvy & Filinski, 1989). The composition of the two CPS transforms gives us $\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2$, a double CPS transform. There is no change to the clauses inherited from $\mathcal{C}_{\lambda\mu}$ since they were already in full CPS form. The only difference is in the translation of $\hat{\text{tp}}$ 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 a value and a meta-continuation as parameters. Here, the initial values for the continuation, k_i , and meta-continuation, γ_i , are:

$$k_i = \lambda x.\lambda\gamma.\gamma\ x \qquad \gamma_i = \lambda x.x$$

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_i and γ_i .

Explicit initial conditions Notice that we are now in the same situation as we were with the pure λ -calculus. The CPS translation of both terms and commands take an extra argument, but this fact is not reflected in the syntax of $\lambda\mu\hat{\text{tp}}$. To reconcile the difference between the CPS transform and the source language, we extend the syntax of $\lambda\mu\hat{\text{tp}}$ in the same way we extended the pure λ -calculus. We add a second-order command, or meta-command, which explicitly names the meta-continuation of the underlying first-order command. Since we can only mark the initial meta-continuation of a command, we add the constant \otimes , which is the meta-top-level of the program. Thus, we extend the syntax of $\lambda\mu\hat{\text{tp}}$ given in Figure 15 with meta-commands, as shown in Figure 19. This gives us two special continuation labels in our syntax: $\hat{\text{tp}}$ represents k_i in the CPS transform, whereas \otimes represents γ_i . The double CPS translation of meta-commands and the meta-top-level \otimes 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 $[\otimes][\hat{\text{tp}}]t$. If the meta-command is reduced to $[\otimes][\hat{\text{tp}}]V$, then the value V is the final answer.

$$[\otimes][\hat{\text{tp}}]t \text{ initial program} \qquad [\otimes][\hat{\text{tp}}]V \text{ final answer}$$

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

$$\begin{aligned}
E &::= \square \mid E \ t \mid V \ E & D &::= [q]E \\
E^2 &::= \square \mid D[\mu\widehat{\text{tp}}.E^2] & D^2 &::= [q^2]E^2 \\
D^2[D[(\lambda x.t) \ V]] &\mapsto D^2[D[t\{V/x\}]] \\
D^2[D[\mu\alpha.c]] &\mapsto D^2[c\{D[t]/[\alpha]t\}] \\
D^2[D[\mu\widehat{\text{tp}}.\widehat{\text{tp}}V]] &\mapsto D^2[D[V]]
\end{aligned}$$

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

$$\begin{aligned}
S &::= \langle c^2 \rangle_{\text{refocus}} \mid \langle c, F^{2*} \rangle_{\text{refocus}} \mid \langle t, F^*, F^{2*} \rangle_{\text{refocus}} \mid \langle F^*, V, F^{2*} \rangle_{\text{apply}} \\
&\mid \langle t, F^*, F^{2*} \rangle_{\text{reduce}} \mid \langle F^{2*}, \widehat{\text{tp}}, V \rangle_{\text{reduce}} \mid \langle V \rangle_{\text{done}} \\
F^{2*} \in \text{DynEnv} &::= q^2 \mid F^{2*}[\widehat{\text{tp}} \mapsto F^*]
\end{aligned}$$

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

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 $\lambda\mu\widehat{\text{tp}}$. This can be derived from defunctionalization (Reynolds, 1972; Danvy, 2004) of the continuation and meta-continuation used in the $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2$ transform. We now have two levels of evaluation contexts: ordinary evaluation contexts from the λ -calculus, and contexts dynamically bound to $\widehat{\text{tp}}$, 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. 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 $\lambda\mu\widehat{\text{tp}}$. Notice that by including the dynamic $\widehat{\text{tp}}$, our machine also gains an additional meta-context, in order to match the CPS transform and operational semantics. The states of the $\lambda\mu\widehat{\text{tp}}$ 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 $\widehat{\text{tp}}$, ending in a meta-co-term q^2 . The notation $F^{2*}[\widehat{\text{tp}} \mapsto F^*]$ is chosen to re-enforce the intuition that $\widehat{\text{tp}}$ is bound to some context, and corresponds with a top-down meta-context $D^2[D[\mu\widehat{\text{tp}}.\square]]$. 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 $\widehat{\text{tp}}$ 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 $\lambda\mu\widehat{\text{tp}}$ -calculus as we had with the $\lambda\mu$ -calculus.

Theorem 4.1 (Soundness)

If $M \twoheadrightarrow M'$ then $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[M] =_{\beta\eta} \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[M']$.

$$\begin{aligned}
\langle [q^2]c \rangle_{\text{refocus}} &\rightsquigarrow \langle c, q^2 \rangle_{\text{refocus}} \\
\langle [q]t, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle t, q, F^{2*} \rangle_{\text{refocus}} \\
\langle t t', F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle t, F^* [\square t'], F^{2*} \rangle_{\text{refocus}} \\
\langle \mu \alpha. c, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle \mu \alpha. c, F^*, F^{2*} \rangle_{\text{reduce}} \\
\langle \mu \widehat{\text{tp}}. c, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle c, F^{2*} [\widehat{\text{tp}} \mapsto F^*] \rangle_{\text{refocus}} \\
\langle V, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle F^*, V, F^{2*} \rangle_{\text{apply}} \\
\langle F^* [\square t], V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle t, F^* [V \square], F^{2*} \rangle_{\text{refocus}} \\
\langle F^* [V' \square], V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle V' V, F^*, F^{2*} \rangle_{\text{reduce}} \\
\langle \widehat{\text{tp}}, V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle F^{2*}, \widehat{\text{tp}}, V \rangle_{\text{reduce}} \\
\langle (\lambda x. t) V, F^*, F^{2*} \rangle_{\text{reduce}} &\rightsquigarrow \langle t \{V/x\}, F^*, F^{2*} \rangle_{\text{refocus}} \\
\langle \mu \alpha. c, F^*, F^{2*} \rangle_{\text{reduce}} &\rightsquigarrow \langle c \{F^*/\alpha\}, F^{2*} \rangle_{\text{refocus}} \\
\langle F^{2*} [\widehat{\text{tp}} \mapsto F^*], \widehat{\text{tp}}, V \rangle_{\text{reduce}} &\rightsquigarrow \langle F^*, V, F^{2*} \rangle_{\text{apply}} \\
\langle \otimes, \widehat{\text{tp}}, V \rangle_{\text{reduce}} &\rightsquigarrow \langle V \rangle_{\text{done}}
\end{aligned}$$

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

$$\begin{aligned}
\#t &= \mu \widehat{\text{tp}}. [\widehat{\text{tp}}]t \\
\mathcal{S} &= \lambda h. \mu \alpha. [\widehat{\text{tp}}]h (\lambda x. \mu \widehat{\text{tp}}. [\alpha]x) \\
\mathcal{A}^{\widehat{\text{tp}}} t &= \mathcal{S} \lambda _ . t = \mu _ . [\widehat{\text{tp}}]t
\end{aligned}$$

Fig. 23. Encodings of the \mathcal{S} and $\#$ control operators in $\lambda\mu\widehat{\text{tp}}$.**Theorem 4.2 (Evaluation)**

If $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[c^2]] =_{\beta} V$ then there is a final answer c'^2 such that $c^2 \mapsto c'^2$ and $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[c'^2]] =_{\beta} V$.

A final answer of the $\lambda\mu\widehat{\text{tp}}$ -calculus is a meta-command of the form $[\otimes][\widehat{\text{tp}}]V$ (or $D^2[[*]V]$ using the alternate initial conditions described in Section 4.1). Notice that soundness now requires the use of η equivalence, which is needed to equate the transformations of programs that use $\widehat{\text{tp}}$ 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.

Expressiveness The rebindable top-level is the additional power that allows us to encode the shift (\mathcal{S}) and reset ($\#$) control operators in $\lambda\mu\widehat{\text{tp}}$, using the encodings in Figure 23. These encodings resemble Filinski's (1994) encoding of \mathcal{S} and $\#$ in terms of Felleisen's \mathcal{C} and $\#$ operators. We can also encode a different abort operator, $\mathcal{A}^{\widehat{\text{tp}}}$, which aborts up to the nearest binding of $\widehat{\text{tp}}$. 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 $\widehat{\text{tp}}$.

$$\begin{aligned}
E &::= \square \mid E t \mid V E & D &::= \square \mid E[\#D] \\
D[E[(\lambda x.t) V]] &\mapsto D[E[t\{V/x\}]] \\
D[E[\mathcal{S} V]] &\mapsto D[V (\lambda x.\#(E[x]))] \\
D[E[\#V]] &\mapsto D[E[V]]
\end{aligned}$$

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

We can derive an operational semantics for shift and reset by their encoding into $\lambda\mu\hat{\text{tp}}$, shown in Figure 24. Since the operational semantics of $\lambda\mu\hat{\text{tp}}$ has two levels of evaluation contexts, so too does the semantics for shift and reset. The reset operator hides a binding of the dynamic $\hat{\text{tp}}$. Therefore, the evaluation meta-contexts D , which are a chain of ordinary evaluation contexts E separated by resets, correspond to the $\lambda\mu\hat{\text{tp}}$ 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 with the semantics given by Biernacka *et al.* (2005) (therein referred to as the reduction semantics). The definition of two-tiered evaluation contexts are 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 reset² validates the axiom $\mathcal{S}(\lambda k.k t) = t$ in every (meta-)context. Therefore, according to the CPS transform, naked shifts of the form $\mathcal{S}(\lambda k.k t)$ 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 $\lambda\mu\hat{\text{tp}}$ calculus.

Example 3

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).

Consider the evaluation of the $\lambda\mu\hat{\tau}\hat{\rho}$ representation of $\mathcal{S}(\lambda_9)$, in the default initial configuration.

$$[\otimes][\hat{\tau}\hat{\rho}]\mu\alpha.[\hat{\tau}\hat{\rho}]((\lambda_9) (\lambda x.\mu\hat{\tau}\hat{\rho}.[\alpha]x)) \rightarrow [\otimes][\hat{\tau}\hat{\rho}]9$$

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

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

$$\mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2[[\otimes][\hat{\tau}\hat{\rho}]9] = \mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2[[\hat{\tau}\hat{\rho}]9] \mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2[[\otimes]] = \mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2[[\otimes]]9 = 9$$

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 $\lambda\mu\hat{\tau}\hat{\rho}$, the $\hat{\tau}\hat{\rho}$ 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 $\hat{\tau}\hat{\rho}$, already comes with $\hat{\tau}\hat{\rho}$ bound to the *true* top-level of the program.

The choice to have the initial program state begin with $\hat{\tau}\hat{\rho}$ 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 $\hat{\tau}\hat{\rho}$ is unbound.

Alternative initial conditions What does it mean for $\hat{\tau}\hat{\rho}$ to be unbound? Operationally, when $\hat{\tau}\hat{\rho}$ is not bound to a context, attempting to lookup $\hat{\tau}\hat{\rho}$ in the command $[\hat{\tau}\hat{\rho}]V$ 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 $\hat{\tau}\hat{\rho}$ (by applying γ_0 to a value), the CPS transformed program gets stuck.

Now we run into a different problem. If we provide k_i and γ_0 as the initial continuation and meta-continuation to our CPS transformed program, then there is no way for evaluation to terminate normally with an answer instead of getting stuck. Since k_i and every other closed continuation eventually finishes 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:

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

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 that we now have both notions of abort as defined in Figures 14 and 23. $\mathcal{S}^{\hat{\tau}\hat{\rho}}$ removes

$$\begin{aligned}
 c^2 \in \text{Command}^2 &::= [q^2]c \\
 c \in \text{Command} &::= [q]t \\
 t \in \text{Term} &::= V \mid t_1 t_2 \mid \mu\alpha.c \mid \mu\hat{\text{tp}}.c & q^2 \in \text{CoTerm}^2 &::= \bullet \\
 V \in \text{Value} &::= x \mid \lambda x.t & q \in \text{CoTerm} &::= \alpha \mid \hat{\text{tp}} \mid *
 \end{aligned}$$

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

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

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

the context up to the nearest binding of $\hat{\text{tp}}$, whereas \mathcal{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 $\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2$ 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 $\hat{\text{tp}}$ is not needed. If the $\hat{\text{tp}}$ continuation is given a value without being bound, then the program gets stuck; since $\hat{\text{tp}}$ was not defined there is not enough information to continue.

The reduction semantics of $\lambda\mu\hat{\text{tp}}$ from Figure 16 is extended with one more rule to reduce an invocation of $*$ under a binding for $\hat{\text{tp}}$.

$$\mu\hat{\text{tp}}.[*]V \rightarrow \mu_.[*]V$$

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

Example 4

Let's revisit the previous example using $*$ and \bullet to initialize execution instead of $\hat{\text{tp}}$ and \otimes .

$$\begin{aligned}
 [\bullet][*]\mu\alpha.[\hat{\text{tp}}]((\lambda_.9) (\lambda x.\mu\hat{\text{tp}}.[\alpha]x)) &\rightarrow [\bullet][\hat{\text{tp}}]9 \\
 \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2[[\bullet][\hat{\text{tp}}]9] &= \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2[[\hat{\text{tp}}]9] \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2[[\bullet]] = \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2[[\bullet]] 9 = \gamma_0 9
 \end{aligned}$$

Since $\hat{\text{tp}}$ 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 $E[\mathcal{S} V]$, we get:

$$\begin{aligned}
 [\bullet][*]E[\mu\alpha.[\hat{\text{tp}}]V (\lambda x.\mu\hat{\text{tp}}.[\alpha]x)] &\rightarrow [\bullet][\hat{\text{tp}}]V (\lambda x.\mu\hat{\text{tp}}.[*](E[x])) \\
 &\rightarrow [\bullet][\hat{\text{tp}}]V (\lambda x.\mu_.[*](E[x]))
 \end{aligned}$$

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 $\mathcal{S}(\lambda k.k\ 9)$ under the two initial conditions. First, when we evaluate $[\otimes][\widehat{\text{tp}}]\mathcal{S}(\lambda k.k\ 9)$, we get:

$$\begin{aligned} [\otimes][\widehat{\text{tp}}]\mu\alpha.\widehat{\text{tp}}\mu\widehat{\text{tp}}.[\alpha]9 &\rightarrow [\otimes][\widehat{\text{tp}}]\mu\widehat{\text{tp}}.[\widehat{\text{tp}}]9 \\ &\rightarrow [\otimes][\widehat{\text{tp}}]9 \end{aligned}$$

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

$$\begin{aligned} [\bullet][*]\mu\alpha.\widehat{\text{tp}}\mu\widehat{\text{tp}}.[\alpha]9 &\rightarrow [\bullet][\widehat{\text{tp}}]\mu\widehat{\text{tp}}.[*]9 \\ &\rightarrow [\bullet][\widehat{\text{tp}}]\mu\text{--}.[*]9 \\ &\rightarrow [\bullet][*]9 \end{aligned}$$

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

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

Theorem 4.4

$$\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\bullet][*]\mu\widehat{\text{tp}}.c] \rightsquigarrow \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\otimes]c]$$

Proof

$$\begin{aligned} \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\bullet][*]\mu\widehat{\text{tp}}.c] &= \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\mu\widehat{\text{tp}}.c] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[*] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[\bullet]] \\ &\rightarrow \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[c]\lambda x.\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[*] x \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[\bullet]] \\ &\rightarrow \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[c]\lambda x.x \\ &= \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[c] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[\otimes]] \\ &= \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\otimes]c] \end{aligned}$$

□

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 $*$:

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

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

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

Ariola *et al.* (2009) showed how the CPS of $\lambda\mu\widehat{\text{tp}}$ can be factored into a state-passing transformation to $\lambda\mu$ extended with subtraction combined with a translation to λ -calculus with pairs. In order to better understand the dynamic nature of the prompt binding, we

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

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

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 $\lambda\mu\widehat{\text{tp}}$ ($\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}$), leaving behind the dynamic binding of $\widehat{\text{tp}}$. 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 $\widehat{\text{tp}}$. We thus propose another way of translating the dynamic binding that models the behavior of the prompt ($\mathcal{D}_{\lambda\widehat{\text{tp}}}$).

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 delimited control can. 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 $\lambda\mu\widehat{\text{tp}}$ to an intermediate language with one dynamic variable $\widehat{\text{tp}}$, 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 $\lambda\mu\widehat{\text{tp}}$ -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 $\widehat{\text{tp}}$, 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, $\lambda\widehat{\text{tp}}.t$, much in the same way that static let bindings are defined in terms of static functions:

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

The $\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}$ transform shown in Figure 28 defines the call-by-value application and the context capturing behavior of $\mu\tilde{\alpha}.c$ while using the dynamic variable in $\lambda\widehat{\text{tp}}$ to manage the binding of $\widehat{\text{tp}}$. 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 $\mu\widehat{\text{tp}}.c$ introduce a static and dynamic variable in the CPS program, respectively.

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

$$\begin{array}{ll}
\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[[q^2]c] = [\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[[q^2]]]\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[c] & \widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[\alpha] = \alpha \\
\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[[q]t] = \widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[t] \widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[q] & \widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[\widehat{\text{tp}}] = \lambda x. \widehat{\text{tp}} x \\
\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[V] = \lambda k.k \widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[V]^V & \widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[*] = \lambda x.x \\
\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[t_1 t_2] = \lambda k.\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[t_1] \lambda f.\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[t_2] \lambda s.f s k & \widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[\bullet] = \bullet \\
\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[\mu\tilde{\alpha}.c] = \lambda k.(\lambda\tilde{\alpha}.\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[c]) k & \widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[x]^V = x \\
& \widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[\lambda x.t]^V = \lambda x.\widehat{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}}[t]
\end{array}$$

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

$$\begin{array}{ll}
\mathcal{D}[[e]t] = \mathcal{D}[t] \mathcal{D}[e] & \mathcal{D}[\bullet] = \gamma_0 \\
\mathcal{D}[V] = \lambda \gamma.\mathcal{D}[V]^V & \mathcal{D}[x]^V = x \\
\mathcal{D}[\widehat{\text{tp}}] = \lambda \gamma.\gamma & \mathcal{D}[\lambda x.t]^V = \lambda x.\mathcal{D}[t] \\
\mathcal{D}[t_1 t_2] = \lambda \gamma.(\mathcal{D}[t_1]\gamma) (\mathcal{D}[t_2]\gamma) \gamma & \mathcal{D}[\lambda \widehat{\text{tp}}.t]^V = \lambda v.\lambda \gamma.\mathcal{D}[t]v
\end{array}$$

Fig. 29. Simple environment-passing style transform of the λ -calculus with one dynamic variable.*Example 6*

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

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

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

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

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

5.2 Simple dynamic binding

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

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

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

$$\begin{aligned}
\mathcal{D}_{\lambda\widehat{\text{tp}}}[\lambda\widehat{\text{tp}}.t]^V &= \lambda v.\lambda\gamma.\mathcal{D}_{\lambda\widehat{\text{tp}}}[t] (\gamma[\widehat{\text{tp}} \mapsto v]) & \gamma[\widehat{\text{tp}} \mapsto v] &= \lambda x.v x \gamma \\
\mathcal{D}_{\lambda\widehat{\text{tp}}}[\widehat{\text{tp}} t] &= \lambda\gamma.\gamma(\widehat{\text{tp}}) (\mathcal{D}_{\lambda\widehat{\text{tp}}}[t]\gamma) & \gamma(\widehat{\text{tp}}) &= \gamma
\end{aligned}$$

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

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

$$(\lambda v.\lambda\gamma'.v x v) (\lambda y.\lambda\gamma'.\gamma' y \gamma') \gamma \rightarrow (\lambda y.\lambda\gamma'.\gamma' y \gamma') x (\lambda y.\lambda\gamma'.\gamma' y \gamma') \rightarrow \dots$$

This does not match the reductions of $\lambda\mu\widehat{\text{tp}}$, since one has: $[\widehat{\text{tp}}]\mu\widehat{\text{tp}}.\widehat{\text{tp}}x \rightarrow [\widehat{\text{tp}}]x$. In Moreau's (1998) framework, this corresponds to the reduction:

$$\mathbf{dlet}\widehat{\text{tp}} = (\lambda y.\widehat{\text{tp}} y) \mathbf{in}\widehat{\text{tp}} x \rightarrow \mathbf{dlet}\widehat{\text{tp}} = (\lambda y.\widehat{\text{tp}} y) \mathbf{in} (\lambda y.\widehat{\text{tp}} y) x \rightarrow \dots$$

Remark 1

Additionally, we can translate the dynamic let binding as:

$$\mathcal{D}[\mathbf{dlet}\widehat{\text{tp}} = t \mathbf{in} u] = \lambda\gamma.\mathcal{D}[u] (\mathcal{D}[t]\gamma)$$

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

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

5.3 Backtracking the environment

We see vicious cycles arise because dynamic binding allows for self-reference. In order to evaluate the application $\widehat{\text{tp}} V$, we (1) lookup the value f most recently bound to $\widehat{\text{tp}}$, and (2) evaluate $f V$ in the current environment where f is still bound. The root of our problem is in step (2). Instead, we want to evaluate $f V$ in a different environment where that same f isn't bound. In particular, we want to backtrack to the environment that was active just before f was bound to $\widehat{\text{tp}}$. To do this, we restrict the grammar of the dynamic language, as shown in Figure 30, so that $\widehat{\text{tp}}$ can only be used if it is immediately being applied to something, giving us $\lambda\widehat{\text{tp}}$. 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 $\widehat{\text{tp}}$ to backtrack to a previous environment, as shown in Figure 31. Here, the notation $\gamma[\widehat{\text{tp}} \mapsto v]$ means that γ , as representation of the dynamic environment, is

extended so that $\hat{\text{tp}}$ is bound to v , and $\gamma(\hat{\text{tp}})$ signifies looking up the most recent binding of $\hat{\text{tp}}$ in γ . In the environment-passing style transform, the environment is represented abstractly as a function that, when invoked, will access the most recent binding of $\hat{\text{tp}}$. Therefore, looking up the binding of $\hat{\text{tp}}$ 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 $\hat{\text{tp}}$, we create a function that, when applied, will call the function v to the argument in the previous environment. Since values bound to $\hat{\text{tp}}$ are equipped with their own environment, we do not need to pass the current dynamic environment to the application $\hat{\text{tp}} V$. Compare the new translation of $\hat{\text{tp}} x$ in Figure 31, $\mathcal{D}_{\lambda\hat{\text{tp}}}[\hat{\text{tp}} x]\gamma = \gamma x$, with the original translation from Figure 29, $\mathcal{D}[\hat{\text{tp}} x]\gamma = \gamma x \gamma$. Notice that the restriction on how $\hat{\text{tp}}$ is used allows us to eliminate the self-application of the environment γ . With the new backtracking definition of dynamic binding, the composed transform $\mathcal{D}_{\lambda\hat{\text{tp}}}\hat{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}$ no longer create the same cycle as before in the reduction of $\mathcal{D}_{\lambda\hat{\text{tp}}}\hat{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}[[\hat{\text{tp}}]\mu\hat{\text{tp}}.[\hat{\text{tp}}]x]\gamma$:

$$(\lambda v.\lambda\gamma'.v x \gamma') (\lambda y.\lambda\gamma'.\gamma' y) x \gamma \rightarrow (\lambda y.\lambda\gamma'.\gamma' y) x \gamma \rightarrow \gamma x = \mathcal{D}_{\lambda\hat{\text{tp}}}[\hat{\text{tp}} x]\gamma$$

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

Theorem 5.1

$$\mathcal{D}_{\lambda\hat{\text{tp}}}\hat{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}[M] =_{\beta\eta} \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2[M]$$

Remark 2

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

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

We want to extend $\lambda\mu\hat{\text{tp}}$ with multiple dynamic co-variables, written by convention with a hat like \hat{a} , so that binding \hat{a} does not interfere with \hat{b} 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 $\lambda\mu\hat{\text{tp}}$ 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, $\lambda\hat{\mu}$ shown in Figure 32, is a simple extension of $\lambda\mu\hat{\text{tp}}$ with multiple dynamic top-level binders.

Reduction The reduction rules for multiple prompts are a generalization of the reduction rules for single prompt $\hat{\text{tp}}$. The reduction theory for the $\lambda\hat{\mu}$ -calculus is based on the $\lambda\mu$ -calculus theory given in Figure 10 and extended with the additional rules for dynamic

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

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

$$\begin{array}{l}
\mu \hat{\alpha}.[\hat{\alpha}]V \rightarrow V \\
\mu \hat{\alpha}.[\hat{\beta}]V \rightarrow \mu \dots.[\hat{\beta}]V \quad \mathbf{where} \hat{\alpha} \neq \hat{\beta} \\
\mu \hat{\alpha}.[*]V \rightarrow \mu \dots.[*]V
\end{array}$$

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

co-variables given in Figure 33. Just like how invocation of $*$ throws away the dynamic environment, invocation of $\hat{\beta}$ will throw away portions of its dynamic environment until the correct binding is found. Then the usual reduction of $\mu \hat{\alpha}.[\hat{\alpha}]V$ is available to plug V into the correct context.

Operational semantics To define the operational semantics for $\lambda\hat{\mu}$, 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 the $\lambda\hat{\mu}$ is given in Figure 34. Note that in the third rule, the context $E_{\hat{\alpha}}^2$ does not bind the dynamic co-variable $\hat{\alpha}$, and is defined as:

$$E_{\hat{\alpha}}^2 ::= \square \mid D[\mu \hat{\beta}.E_{\hat{\alpha}}^2] \quad \mathbf{where} \hat{\beta} \neq \hat{\alpha}$$

CPS transform The intermediate language of dynamic binding, $\hat{\lambda}$, is extended with the syntax shown in Figure 35. The definition of $\hat{\lambda}$ uses the same backtracking environment-passing style translation as $\hat{\lambda}\hat{\tau}\hat{\rho}$ from Figure 31. The only thing that needs to change from $\hat{\lambda}\hat{\tau}\hat{\rho}$ to $\hat{\lambda}$ 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, $\gamma(\hat{x})$ and $\gamma[\hat{x} \mapsto v]$ are notation for looking up bindings and

$$\begin{array}{l}
E^2 ::= \square \mid D[\mu \hat{\alpha}.E^2] \\
E ::= \square \mid E t \mid V E \\
D^2 ::= [q^2]E^2 \\
D ::= [q]E \\
D^2[D[(\lambda x.t) V]] \mapsto D^2[D\{t\{V/x\}\}] \\
D^2[D[\mu \alpha.c]] \mapsto D^2[c\{D[t]/[\alpha]t\}] \\
D^2[D[\mu \hat{\alpha}.E_{\hat{\alpha}}^2[[\hat{\alpha}]V]]] \mapsto D^2[D[V]]
\end{array}$$

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

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

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

$$\gamma(\widehat{x}) = \gamma \lceil \widehat{x} \rceil \quad \gamma[\widehat{x} \mapsto v] = \lambda p. \mathbf{if} p \equiv \lceil \widehat{x} \rceil \mathbf{then} (\lambda x.v x \gamma) \mathbf{else} \gamma p$$

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

extending the environment γ , which is represented in a functionalized form. Additionally the quotation brackets, $\lceil \widehat{x} \rceil$, 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 $\mathcal{D}_{\widehat{\lambda}} \widehat{\mathcal{C}}_{\lambda \widehat{\mu}}$, where $\widehat{\mathcal{C}}_{\lambda \widehat{\mu}}$ is the same as $\widehat{\mathcal{C}}_{\lambda \mu \widehat{\text{tp}}}$ from Figure 28 except that multiple dynamic variables are used by $\widehat{\mathcal{C}}_{\lambda \widehat{\mu}}$, with one unique dynamic variable for each different dynamic co-variable in the source program.

Abstract machine We extend the abstract machine for $\lambda \mu \widehat{\text{tp}}$ 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 $\lambda \mu \widehat{\text{tp}}$.

Theorem 6.1 (Soundness)

$$\text{If } M \rightarrow M' \text{ then } \mathcal{D}_{\widehat{\lambda}} \widehat{\mathcal{C}}_{\lambda \widehat{\mu}} \llbracket M \rrbracket =_{\beta \eta} \mathcal{D}_{\widehat{\lambda}} \widehat{\mathcal{C}}_{\lambda \widehat{\mu}} \llbracket M' \rrbracket.$$

Theorem 6.2 (Evaluation)

$$\text{If } \mathcal{D}_{\widehat{\lambda}} \widehat{\mathcal{C}}_{\lambda \widehat{\mu}} \llbracket c^2 \rrbracket =_{\beta} V \text{ then there is a final answer } c'^2 \text{ such that } c^2 \mapsto c'^2 \text{ and } \mathcal{D}_{\widehat{\lambda}} \widehat{\mathcal{C}}_{\lambda \widehat{\mu}} \llbracket c'^2 \rrbracket =_{\beta} V.$$

A final answer of the $\lambda \widehat{\mu}$ -calculus is a meta-command of the form $D^2 \llbracket [*] V \rrbracket$. 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 ($\mathcal{A} \widehat{\alpha}$) operators in Figure 39.

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

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

$$\begin{aligned}
\langle [q^2]c \rangle_{\text{refocus}} &\rightsquigarrow \langle c, q^2 \rangle_{\text{refocus}} \\
\langle [q]t, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle t, q, F^{2*} \rangle_{\text{refocus}} \\
\langle t t', F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle t, F^* [\square t'], F^{2*} \rangle_{\text{refocus}} \\
\langle \mu \alpha.c, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle \mu \alpha.c, F^*, F^{2*} \rangle_{\text{reduce}} \\
\langle \mu \hat{\alpha}.c, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle c, F^{2*} [\hat{\alpha} \mapsto F^*] \rangle_{\text{refocus}} \\
\langle V, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle F^*, V, F^{2*} \rangle_{\text{apply}} \\
\\
\langle F^* [\square t], V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle t, F^* [V \square], F^{2*} \rangle_{\text{refocus}} \\
\langle F^* [V' \square], V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle V' V, F^*, F^{2*} \rangle_{\text{reduce}} \\
\langle \hat{\alpha}, V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle F^{2*}, \hat{\alpha}, V \rangle_{\text{lookup}} \\
\langle *, V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle V \rangle_{\text{done}} \\
\\
\langle (\lambda x.t) V, F^*, F^{2*} \rangle_{\text{reduce}} &\rightsquigarrow \langle t \{V/x\}, F^*, F^{2*} \rangle_{\text{refocus}} \\
\langle \mu \alpha.c, F^*, F^{2*} \rangle_{\text{reduce}} &\rightsquigarrow \langle c \{F^*/\alpha\}, F^{2*} \rangle_{\text{refocus}} \\
\langle F^{2*} [\hat{\alpha} \mapsto F^*], \hat{\alpha}, V \rangle_{\text{reduce}} &\rightsquigarrow \langle V, F^*, F^{2*} \rangle_{\text{apply}} \\
\\
\langle F^{2*} [\hat{\alpha} \mapsto F^*], \hat{\alpha}, V \rangle_{\text{lookup}} &\rightsquigarrow \langle F^{2*} [\hat{\alpha} \mapsto F^*], \hat{\alpha}, V \rangle_{\text{reduce}} \\
\langle F^{2*} [\hat{\beta} \mapsto F^*], \hat{\alpha}, V \rangle_{\text{lookup}} &\rightsquigarrow \langle F^{2*}, \hat{\alpha}, V \rangle_{\text{lookup}} \quad \text{where } \hat{\alpha} \neq \hat{\beta}
\end{aligned}$$

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

$$\begin{aligned}
\#^{\hat{\alpha}} t &= \mu \hat{\alpha}. [\hat{\alpha}] t \\
\mathcal{S}^{\hat{\alpha}} t &= \mu _ . [\hat{\alpha}] t
\end{aligned}$$

Fig. 39. The multi-prompt $\#^{\hat{\alpha}}$ and $\mathcal{S}^{\hat{\alpha}}$ control operators.

7 Delimited control with a dynamic prompt: $\lambda\mu\hat{\text{tr}}_0$

We now take a break from $\lambda\hat{\mu}$ and multiple prompts, and return to $\lambda\mu\hat{\text{tr}}$ in order to examine an alternate extension. Another important delimited control operator to consider is shift_0 (\mathcal{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.

Example 7

Consider the behavior of the following program which uses the shift_0 operator twice in a row:

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

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

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

$$\mu_0 \widehat{\text{tp}}. [\widehat{\text{tp}}]_0 t \rightarrow t \qquad \mu_0 \widehat{\text{tp}}. [*]V \rightarrow \mu_0. [*]V$$

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

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 .

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

Notice how after the first shift_0 is reduced, its body is evaluated outside of 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 $\lambda\mu\widehat{\text{tp}}$. However, to capture the additional behavior of shift_0 we need to extend $\lambda\mu\widehat{\text{tp}}$ with the ability to render the binding of a prompt transparent, making it immediately disappear and letting underlying terms see through to their surrounding context. To that end, we introduce a new form of command, $[\widehat{\text{tp}}]_0 t$, in which the dynamic co-term $\widehat{\text{tp}}$ has priority over the term t , rather than the other way around. In a sense, this is a “non-strict” variant of the command $[\widehat{\text{tp}}]t$, where invoking the dynamic top-level happens without first evaluating the term t , as is the case in the expression $\mathcal{S}_0 \lambda k. t$. 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 $\lambda\mu\widehat{\text{tp}}_0$ is given in Figure 40. The new command $[\widehat{\text{tp}}]_0 t$ represents lifting the unevaluated term t through the most recent binding of $\widehat{\text{tp}}$ and embedding the term in that context. Correspondingly, the binding of $\widehat{\text{tp}}$, $\mu_0 \widehat{\text{tp}}.c$, 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 $[\widehat{\text{tp}}]t$ from $\lambda\mu\widehat{\text{tp}}$, but with different priorities between the continuation and the term. In the $\lambda\mu\widehat{\text{tp}}$ -calculus term $\mu\widehat{\text{tp}}. [\widehat{\text{tp}}]t$, $\widehat{\text{tp}}$ is reduced only when t is a value. The opposite occurs with $\mu_0 \widehat{\text{tp}}. [\widehat{\text{tp}}]_0 t$, where $\widehat{\text{tp}}$ is reduced immediately without considering t , as shown in Figure 41. Note that due to the reversed priorities in the command $[\widehat{\text{tp}}]_0 t$, the usual renaming rule for μ does not apply to this use of $\widehat{\text{tp}}$:

$$[\widehat{\text{tp}}]_0 \mu\alpha.c \not\rightarrow c\{\widehat{\text{tp}}/\alpha\}$$

$$\begin{aligned}
E^2 &::= \square \mid D[\mu_0 \widehat{\text{tp}}.E^2] & D^2 &::= [q^2]E^2 \\
E &::= \square \mid E \ t \mid V \ E & D &::= [q]E
\end{aligned}$$

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

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

$$\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0} \llbracket \mu_0 \widehat{\text{tp}}.c \rrbracket = \lambda k. \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0} \llbracket c \rrbracket k \qquad \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0} \llbracket [\widehat{\text{tp}}]_0 t \rrbracket = \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0} \llbracket t \rrbracket$$

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

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

CPS transform We define a CPS for $\lambda\mu\widehat{\text{tp}}_0$ in the style of Materzok & Biernacki’s (2011) definition of shift_0 , shown in Figure 43. This is an extension of the basic $\mathcal{C}_{\lambda\mu}$ transform from Figure 12. Note how the translation of $\mu_0 \widehat{\text{tp}}.c$ in $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}$ is different from the translation of $\mu \widehat{\text{tp}}.c$ in $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}$. 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 $[\widehat{\text{tp}}]_0 t$, 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 $\mu_0 \widehat{\text{tp}}.c$ 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 $\mathcal{C}_{\lambda\mu}$ 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 $\mu_0 \widehat{\text{tp}}.c$ corresponds to pushing a continuation onto the meta-continuation stack, whereas accessing the most

$$\begin{aligned}
\mathcal{C}_{\lambda\mu\hat{\tau}\rho_0}^\times \llbracket \mu_0\hat{\tau}\rho.c \rrbracket &= \lambda k. \lambda \gamma. \mathcal{C}_{\lambda\mu\hat{\tau}\rho_0}^\times \llbracket c \rrbracket \langle k, \gamma \rangle \\
\mathcal{C}_{\lambda\mu\hat{\tau}\rho_0}^\times \llbracket [\hat{\tau}\rho]_0 t \rrbracket &= \lambda \langle k, \gamma \rangle. \mathcal{C}_{\lambda\mu\hat{\tau}\rho_0}^\times \llbracket t \rrbracket k \gamma \\
\mathcal{C}_{\lambda\mu\hat{\tau}\rho_0}^\times \llbracket * \rrbracket &= \lambda x. \lambda \gamma. x
\end{aligned}$$

Fig. 44. Stack-based call-by-value CPS transform of the $\lambda\mu\hat{\tau}\rho_0$ -calculus.

$$\begin{aligned}
\hat{\mathcal{C}}_{\lambda\mu\hat{\tau}\rho_0} \llbracket \mu_0\hat{\tau}\rho.c \rrbracket &= \lambda k. (\lambda \hat{\tau}\rho. \hat{\mathcal{C}}_{\lambda\mu\hat{\tau}\rho_0} \llbracket c \rrbracket) k \\
\hat{\mathcal{C}}_{\lambda\mu\hat{\tau}\rho_0} \llbracket [\hat{\tau}\rho]_0 t \rrbracket &= \hat{\mathcal{C}}_{\lambda\mu\hat{\tau}\rho_0} \llbracket t \rrbracket \hat{\tau}\rho \\
\hat{\mathcal{C}}_{\lambda\mu\hat{\tau}\rho_0} \llbracket * \rrbracket &= \lambda x. x
\end{aligned}$$

Fig. 45. Call-by-value CPS transform of $\hat{\tau}\rho$ in the $\lambda\mu\hat{\tau}\rho_0$ -calculus using one dynamic variable.

recent binding of $\hat{\tau}\rho$ with $[\hat{\tau}\rho]_0 t$ corresponds to popping the top continuation off the stack and evaluating the term t in that continuation.

As with the transform for $\lambda\mu\hat{\tau}\rho$ from Figure 28, we can express the CPS transform in terms of a language of dynamic binding as shown in Figure 45. Notice that in order to accommodate the new command $[\hat{\tau}\rho]_0 t$, we need to relax the restrictions on when we can lookup the dynamic $\hat{\tau}\rho$. In $\lambda\mu\hat{\tau}\rho$, we only ever need to lookup $\hat{\tau}\rho$ when we are applying it to a value. However, in $\lambda\mu\hat{\tau}\rho_0$, we need to lookup the continuation bound to $\hat{\tau}\rho$ before we are ready to apply the continuation to a value. As before with $\lambda\hat{\tau}\rho$, looking up the dynamic $\hat{\tau}\rho$ backtracks the dynamic environment to its previous state before $\hat{\tau}\rho$ was bound. The syntax of this dynamic language, $\lambda\hat{\tau}\rho_0$, is given in Figure 46. This is an extension of the previous dynamic language $\lambda\hat{\tau}\rho$ since we can still apply $\hat{\tau}\rho$ to a term by first explicitly looking up the value of $\hat{\tau}\rho$ and binding it to a variable:

$$\hat{\tau}\rho t = (\lambda x. (\lambda f. f x) \hat{\tau}\rho) t$$

We can extend the environment-passing style transform $\mathcal{D}_{\lambda\hat{\tau}\rho}$ 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 $\hat{\mathcal{C}}_{\lambda\mu\hat{\tau}\rho_0}$ transform from Figure 45 with the $\mathcal{D}_{\lambda\hat{\tau}\rho_0}^\times$ environment-passing style transform is the same as the $\mathcal{C}_{\lambda\mu\hat{\tau}\rho_0}^\times$ 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 Figure 31. Binding a value to $\hat{\tau}\rho$ in a dynamic environment corresponds to constructing a function that passes

$$\begin{aligned}
c \in \text{Closure} &::= [e]t \\
t \in \text{Term} &::= V \mid t_1 t_2 \mid t \hat{\tau}\rho & e \in \text{Environment} &::= \bullet \\
V \in \text{Value} &::= x \mid \lambda \tilde{x}. t & \tilde{x} \in \text{Var} &::= x \mid \hat{\tau}\rho
\end{aligned}$$

Fig. 46. The syntax of the $\lambda\hat{\tau}\rho_0$ -calculus of dynamic binding.

$$\begin{aligned}
\mathcal{D}_{\lambda\hat{\text{tp}}_0}^\times \llbracket \lambda\hat{\text{tp}}.t \rrbracket^V &= \lambda v. \lambda \gamma. \mathcal{D}_{\lambda\hat{\text{tp}}_0}^\times \llbracket t \rrbracket (\gamma(\hat{\text{tp}} \mapsto v)) \\
\mathcal{D}_{\lambda\hat{\text{tp}}_0}^\times \llbracket t \hat{\text{tp}} \rrbracket &= \lambda \gamma. \mathbf{let} \langle x, \gamma' \rangle = \gamma(\hat{\text{tp}}) \mathbf{in} (\mathcal{D}_{\lambda\hat{\text{tp}}_0}^\times \llbracket t \rrbracket \gamma) x \gamma' \\
\gamma(\hat{\text{tp}} \mapsto v) &= \langle v, \gamma \rangle \quad \gamma(\hat{\text{tp}}) = \gamma
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket \lambda\hat{\text{tp}}.t \rrbracket^V &= \lambda v. \lambda \gamma. \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket t \rrbracket (\gamma(\hat{\text{tp}} \mapsto v)) \\
\mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket t \hat{\text{tp}} \rrbracket \gamma &= \gamma(\hat{\text{tp}}) (\mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket t \rrbracket \gamma) \\
\gamma(\hat{\text{tp}} \mapsto v) &= \lambda g. g v \gamma \quad \gamma(\hat{\text{tp}}) = \gamma
\end{aligned}$$

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

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 $\lambda\mu\hat{\text{tp}}_0$, taken as the composition of the $\mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}$ and $\mathcal{D}_{\lambda\hat{\text{tp}}_0}$ transforms. This transform makes it explicit that unevaluated terms are passed to the meta-continuation. Also, notice that the $\mathcal{D}_{\lambda\hat{\text{tp}}_0} \mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}$ transform is the same as the double CPS transform $\mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}^2$, defined as the composition of $\mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}$ and \mathcal{E}_λ , 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 $\mathcal{E}_{\lambda\mu}$ from Figure 12.

Theorem 7.1

$$\mathcal{D}_{\lambda\hat{\text{tp}}_0}^\times \widehat{\mathcal{E}}_{\lambda\mu\hat{\text{tp}}_0} \llbracket M \rrbracket = \beta\eta \mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}^\times \llbracket M \rrbracket \quad \text{and} \quad \mathcal{D}_{\lambda\hat{\text{tp}}_0} \widehat{\mathcal{E}}_{\lambda\mu\hat{\text{tp}}_0} \llbracket M \rrbracket = \beta\eta \mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket M \rrbracket$$

Abstract machine The $\lambda\mu\hat{\text{tp}}_0$ abstract machine generalizes the reduce state of the $\lambda\mu\hat{\text{tp}}$ machine from Figure 21, as shown in Figure 50. Extending the $\lambda\mu\hat{\text{tp}}$ abstract machine for $\lambda\mu\hat{\text{tp}}_0$ only requires one additional clause to interpret the new form of commands and the generalization of one reduce step. The complete set of steps for the $\lambda\mu\hat{\text{tp}}_0$ 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 $\lambda\mu\hat{\text{tp}}_0$ as we had with $\lambda\mu$, $\lambda\mu\hat{\text{tp}}$, and $\lambda\hat{\mu}$.

$$\begin{aligned}
\mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket \mu_0\hat{\text{tp}}.c \rrbracket &= \lambda k. \lambda \gamma. \mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket c \rrbracket \lambda u. u k \gamma \\
\mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket [\hat{\text{tp}}]_0 t \rrbracket &= \lambda \gamma. \gamma \mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket t \rrbracket \\
\mathcal{E}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket * \rrbracket &= \lambda x. \lambda \gamma. x
\end{aligned}$$

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

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

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

$$\begin{aligned} \langle [q^2]c \rangle_{\text{refocus}} &\rightsquigarrow \langle c, q^2 \rangle_{\text{refocus}} \\ \langle [q]t, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle t, q, F^{2*} \rangle_{\text{refocus}} \\ \langle [\widehat{\text{tp}}]_0 t, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle F^{2*}, \widehat{\text{tp}}, t \rangle_{\text{reduce}} \\ \langle t t', F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle t, F^*[\square t'], F^{2*} \rangle_{\text{refocus}} \\ \langle \mu\alpha.c, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle \mu\alpha.c, F^*, F^{2*} \rangle_{\text{reduce}} \\ \langle \mu_0 \widehat{\text{tp}}.c, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle c, F^{2*}[\widehat{\text{tp}} \mapsto F^*] \rangle_{\text{refocus}} \\ \langle V, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle F^*, V, F^{2*} \rangle_{\text{apply}} \\ \\ \langle F^*[\square t], V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle t, F^*[V \square], F^{2*} \rangle_{\text{refocus}} \\ \langle F^*[V' \square], V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle V' V, F^*, F^{2*} \rangle_{\text{reduce}} \\ \langle *, V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle V \rangle_{\text{done}} \\ \\ \langle (\lambda x.t) V, F^*, F^{2*} \rangle_{\text{reduce}} &\rightsquigarrow \langle t\{V/x\}, F^*, F^{2*} \rangle_{\text{refocus}} \\ \langle \mu\alpha.c, F^*, F^{2*} \rangle_{\text{reduce}} &\rightsquigarrow \langle c\{F^*/\alpha\}, F^{2*} \rangle_{\text{refocus}} \\ \langle F^{2*}[\widehat{\text{tp}} \mapsto F^*], \widehat{\text{tp}}, t \rangle_{\text{reduce}} &\rightsquigarrow \langle t, F^*, F^{2*} \rangle_{\text{refocus}} \end{aligned}$$

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

Theorem 7.2 (Soundness)

If $M \rightarrow M'$ then $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2 \llbracket M \rrbracket =_{\beta\eta} \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2 \llbracket M' \rrbracket$.

Theorem 7.3 (Evaluation)

If $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2 \llbracket c^2 \rrbracket =_{\beta} V$ then there is a final answer c'^2 such that $c^2 \mapsto c'^2$ and $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2 \llbracket c'^2 \rrbracket =_{\beta} V$.

A final answer of the $\lambda\mu\widehat{\text{tp}}_0$ -calculus is a meta-command of the form $D^2 \llbracket [*]V \rrbracket$. 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 $[\widehat{\text{tp}}]t$ which evaluates its term first before accessing $\widehat{\text{tp}}$. This command can be encoded in $\lambda\mu\widehat{\text{tp}}_0$ as:

$$[\widehat{\text{tp}}]t = [*]\text{let } x = t \text{ in } \mu_{-}. [\widehat{\text{tp}}]_0 x$$

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

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

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

$$\begin{aligned}\#_0 t &= \mu_0 \widehat{\text{tp}}. [\widehat{\text{tp}}] t \\ \mathcal{S}_0 &= \lambda h. \mu \alpha. [\widehat{\text{tp}}]_0 h (\lambda x. \mu_0 \widehat{\text{tp}}. [\alpha] x)\end{aligned}$$

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

$$\begin{aligned}E &::= \square \mid E t \mid V E & D &::= \square \mid D[E \#_0 \square] \\ D[E[(\lambda x. t) V]] &\mapsto D[E\{t\{V/x\}}] \\ D[E[\#_0 V]] &\mapsto D[E[V]] \\ D[E'[\#_0 E[\mathcal{S}_0 V]]] &\mapsto D[E'[V (\lambda x. \#_0 E[x])]]\end{aligned}$$

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

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

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

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

$$\begin{aligned}\#' V &\rightarrow V \\ \#' E[\mathcal{S}' V] &\rightarrow \#' (V (\lambda x. \#' E[x]))\end{aligned}$$

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 :

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

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

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

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

$$\begin{aligned}
[\bullet][*]\mathcal{S}'\lambda k.k V &\rightarrow [\bullet][*]\mu \alpha.[\hat{\text{tp}}]_0 \mu_0 \hat{\text{tp}}.[\hat{\text{tp}}]_0 \mu_0 \hat{\text{tp}}.[\alpha]V \\
&\rightarrow [\bullet][\hat{\text{tp}}]_0 \mu_0 \hat{\text{tp}}.[\hat{\text{tp}}]_0 \mu_0 \hat{\text{tp}}.[*]V \\
&\rightarrow [\bullet][\hat{\text{tp}}]_0 \mu_0 \hat{\text{tp}}.[\hat{\text{tp}}]_0 \mu_0 \hat{\text{tp}}.[*]V \\
&\rightarrow [\bullet][\hat{\text{tp}}]_0 \mu_0 \hat{\text{tp}}.[*]V \\
&\rightarrow [\bullet][\hat{\text{tp}}]_0 \mu_0 \hat{\text{tp}}.[*]V
\end{aligned}$$

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

$$\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket [\bullet][\hat{\text{tp}}]_0 \mu_0 \hat{\text{tp}}.[*]V \rrbracket \rightarrow \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket [\bullet] \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket \mu_0 \hat{\text{tp}}.[*]V \rrbracket \rrbracket \rightarrow \gamma_0 (\lambda_0 \lambda_0 V)$$

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

With just the simple addition of multiple static prompts, we still don't have enough expressive power in $\lambda\hat{\mu}$ to encode operators with dynamic prompts like shift_0 and reset_0 , or similar previous control operators given by Gunter *et al.* (1995) and 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 $\hat{\alpha}$ not only captures its immediate context up to the nearest reset_0 , but also captures all the contexts found behind non-matching reset_0 s. 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:

$$\#_0^{\hat{\alpha}}(E[\#_0^{\hat{\beta}}(E'[\mathcal{S}_0^{\hat{\alpha}} V])]) \rightarrow V (\lambda x.\#_0^{\hat{\alpha}}(E[\#_0^{\hat{\beta}}(E'[x])]))$$

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 $\hat{\text{tp}}$ and exposes that context to an underlying term. Therefore, we need to incorporate both multiple prompt binding from Section 6 as well as dynamic prompts from Section 7.

Syntax $\lambda\hat{\mu}_0$ extends $\lambda\hat{\mu}$ 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 $\lambda\hat{\mu}_0$, 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 $[\hat{\alpha}]_0 \Delta.t$ captures the segment

$$\begin{aligned}
\mu_0 \widehat{\alpha}. [\widehat{\alpha}]_0 \Delta. t &\rightarrow t\{c/[\Delta]c\} \\
\mu_0 \widehat{\alpha}. [\widehat{\beta}]_0 \Delta. t &\rightarrow \mu \alpha. [\widehat{\beta}]_0 \Delta. t\{[\Delta][\alpha](\mu_0 \widehat{\alpha}. c)/[\Delta]c\} && \text{where } \widehat{\alpha} \neq \widehat{\beta} \\
\mu_0 \widehat{\alpha}. [*]V &\rightarrow \mu _ . [*]V
\end{aligned}$$

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

of its meta-context as Δ , up to the nearest binding of $\widehat{\alpha}$. Then, that segment of the meta-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 of Δ . The static μ -abstraction is then able to reduce further, incrementally absorbing its context and filling in the renewed bindings for $\widehat{\beta}$ inside Δ . If instead 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.

Similar to $\lambda\mu\widehat{\text{tp}}_0$, 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:

$$\begin{aligned}
[\widehat{\alpha}]V &\twoheadrightarrow [\widehat{\alpha}]_{0 _} V \\
[\widehat{\alpha}]\mu\beta.c &\twoheadrightarrow c\{[\widehat{\alpha}]t/[\beta]t\}
\end{aligned}$$

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 of dynamic co-variables in t .

$$\begin{aligned}
\mu_0 \widehat{\alpha}. [\beta]\mu_0 \widehat{\beta}. [\alpha]\mu_0 \widehat{\alpha}. [\widehat{\beta}]_0 \Delta. \mu _ . [\widehat{\alpha}]x \\
\rightarrow \mu_0 \widehat{\alpha}. [\beta]\mu_0 \widehat{\beta}. [\alpha]\mu \alpha'. [\widehat{\beta}]_0 \Delta. \mu _ . [\widehat{\alpha}]x && \text{where } \widehat{\alpha} \neq \widehat{\beta} \\
\rightarrow \mu_0 \widehat{\alpha}. [\beta]\mu_0 \widehat{\beta}. [\widehat{\beta}]_0 \Delta. \mu _ . [\widehat{\alpha}]x \\
\rightarrow \mu_0 \widehat{\alpha}. [\beta]\mu _ . [\widehat{\alpha}]x \\
\rightarrow \mu_0 \widehat{\alpha}. [\widehat{\alpha}]x \\
\twoheadrightarrow x
\end{aligned}$$

$$\begin{aligned}
E^2 &::= \square \mid D[\mu_0 \hat{\alpha}. E^2] \mid [\Delta] E^2 & D^2 &::= [q^2] E^2 \\
E &::= \square \mid E \ t \mid V \ E & D &::= [q] E \\
D^2[D[(\lambda x.t) \ V]] &\mapsto D^2[D[t\{V/x\}]] \\
D^2[D[\mu \alpha.c]] &\mapsto D^2[c\{D[t]/[\alpha]t\}] \\
D^2[D[\mu_0 \hat{\alpha}. E^2_\alpha[[\hat{\alpha}]_0 \Delta.t]]] &\mapsto D^2[D[t\{E^2_\alpha[c]/[\Delta]c\}]]
\end{aligned}$$

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

Example 10

Capturing and reinstalling a prefix of the evaluation meta-context, as performed by the command $c_0 = [\hat{\alpha}_1]_0 \Delta. \mu_0 \hat{\alpha}_1. [\Delta] c$, preserves its order, so that $\mu_0 \hat{\alpha}_1. E^2[c_0] \rightarrow \mu_0 \hat{\alpha}_1. E^2[c]$ when E^2 does not bind $\hat{\alpha}_1$.

$$\begin{aligned}
&[\beta_1] \mu_0 \hat{\alpha}_1. [\beta_2] \mu_0 \hat{\alpha}_2. [\beta_3] \mu_0 \hat{\alpha}_3. [\hat{\alpha}_1]_0 \Delta. \mu_0 \hat{\alpha}_1. [\Delta] c \\
&\rightarrow [\beta_1] \mu_0 \hat{\alpha}_1. [\beta_2] \mu_0 \hat{\alpha}_2. [\beta_3] \mu \gamma. [\hat{\alpha}_1]_0 \Delta. \mu_0 \hat{\alpha}_1. [\Delta] [\gamma] \mu_0 \hat{\alpha}_3. c \\
&\rightarrow [\beta_1] \mu_0 \hat{\alpha}_1. [\beta_2] \mu_0 \hat{\alpha}_2. [\hat{\alpha}_1]_0 \Delta. \mu_0 \hat{\alpha}_1. [\Delta] [\beta_3] \mu_0 \hat{\alpha}_3. c \\
&\rightarrow [\beta_1] \mu_0 \hat{\alpha}_1. [\beta_2] \mu \gamma. [\hat{\alpha}_1]_0 \Delta. \mu_0 \hat{\alpha}_1. [\Delta] [\gamma] \mu_0 \hat{\alpha}_2. [\beta_3] \mu_0 \hat{\alpha}_3. c \\
&\rightarrow [\beta_1] \mu_0 \hat{\alpha}_1. [\hat{\alpha}_1]_0 \Delta. \mu \hat{\alpha}_1. [\Delta] [\beta_2] \mu_0 \hat{\alpha}_2. [\beta_3] \mu_0 \hat{\alpha}_3. c \\
&\rightarrow [\beta_1] \mu_0 \hat{\alpha}_1. [\beta_2] \mu_0 \hat{\alpha}_2. [\beta_3] \mu_0 \hat{\alpha}_3. c
\end{aligned}$$

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

$$E^2_\alpha ::= \square \mid D[\mu_0 \hat{\beta}. E^2_\alpha] \quad \text{where } \hat{\beta} \neq \hat{\alpha}$$

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 $V \hat{x}$ to give us both the value that was stored in \hat{x} as well as a trace of all the changes to the environment after \hat{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 $\hat{\lambda}$ and $\lambda \hat{t}p_0$, by combining both multiple dynamic variables and reversal of dynamic binding, giving us $\hat{\lambda}_0$ in Figure 57. In the language of dynamic binding $\hat{\lambda}_0$, the new class of variables, Δ , ranges over dynamic environment prefixes. Intuitively, the term $(\lambda \langle \Delta, x \rangle. t) \hat{x}$ looks up the most recent binding of \hat{x} , substituting the value bound to \hat{x} for x while also capturing the prefix of the environment more recent than \hat{x} and substituting it for Δ . Then, the term t is evaluated in the dynamic environment that was in place immediately before \hat{x} was bound. Closure under the prefix, $[\Delta]t$, extends the surrounding environment

$$\begin{aligned}
c \in \text{Closure} &::= [e]t \\
t \in \text{Term} &::= V \mid t_1 t_2 \mid t \hat{x} \mid [\Delta]t & e \in \text{Environment} &::= \bullet \\
V \in \text{Value} &::= x \mid \lambda \tilde{x}.t \mid \lambda \langle \Delta, x \rangle .t & \tilde{x} \in \text{Var} &::= x \mid \hat{x}
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{D}_{\lambda_0}^\times [[\lambda \langle \Delta, x \rangle .t]]^V &= \lambda \Delta. \lambda x. \mathcal{D}_{\lambda_0}^\times [[t]] \\
\mathcal{D}_{\lambda_0}^\times [[t \hat{x}]] &= \lambda \gamma. \mathbf{let} \langle \Delta, v, \gamma' \rangle = \gamma(\hat{x}) \mathbf{in} (\mathcal{D}_{\lambda_0}^\times [[t]] \gamma) \Delta v \gamma' \\
\mathcal{D}_{\lambda_0}^\times [[[\Delta]t]] &= \lambda \gamma. \mathcal{D}_{\lambda_0}^\times [[t]] (\gamma @ \Delta) \\
\gamma[\hat{x} \mapsto v](\hat{x}) &= \langle [], v, \gamma \rangle \\
\gamma[\hat{y} \mapsto v](\hat{x}) &= \mathbf{let} \langle \Delta, v', \gamma' \rangle = \gamma(\hat{x}) \\
&\quad \mathbf{in} \langle \Delta[\hat{y} \mapsto v], v', \gamma' \rangle \\
\gamma @ [] &= \gamma \\
\gamma @ (\Delta[\hat{x} \mapsto v]) &= (\gamma @ \Delta)[\hat{x} \mapsto v]
\end{aligned}$$

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

with all the dynamic bindings stored in Δ . Therefore, in a program such as:

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

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

The semantics of $\widehat{\lambda}_0$, like $\lambda \widehat{\text{tp}}_0$, 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 a term's dynamic environment. Note that when a prefix extends a term, the bindings in that prefix are more recent than the surrounding dynamic environment and are bound in exactly the same order in which they originally occurred.

Taking the concrete implementation, we can derive the pure λ -calculus encoding by refunctionalizing the data structures. The environment prefix is now a function mapping terms to terms which implements the extension operation from before. Multiple return values are emulated by taking a continuation that accepts each of the three return values separately. The environment-passing style transform $\mathcal{D}_{\lambda_0}^\times$ is given in Figure 59.

The CPS translation from $\lambda \widehat{\mu}_0$ to $\widehat{\lambda}_0$ is a merging of $\widehat{\mathcal{E}}_{\lambda \widehat{\mu}}$ and $\widehat{\mathcal{E}}_{\lambda \widehat{\text{tp}}_0}$. The new syntactic forms in $\lambda \widehat{\mu}_0$ can be defined in terms of the intermediate language $\widehat{\lambda}_0$. Capturing a portion

$$\begin{aligned}
\mathcal{D}_{\hat{\lambda}_0} \llbracket t \hat{x} \rrbracket &= \lambda \gamma. \gamma(\hat{x}) (\mathcal{D}_{\hat{\lambda}_0} \llbracket t \rrbracket \gamma) \\
\mathcal{D}_{\hat{\lambda}_0} \llbracket [\Delta] t \rrbracket &= \lambda \gamma. \mathcal{D}_{\hat{\lambda}_0} \llbracket t \rrbracket (\Delta \gamma) \\
\gamma(\hat{x}) &= \gamma \ulcorner \hat{x} \urcorner \\
\gamma[\hat{x} \mapsto v] &= \lambda p. \mathbf{if} p \equiv \ulcorner \hat{x} \urcorner \\
&\quad \mathbf{then} (\lambda g. g (\lambda \gamma'. \gamma') v \gamma) \\
&\quad \mathbf{else} (\lambda g. \gamma p (\lambda \Delta. g (\lambda \gamma'. (\Delta \gamma') [\hat{x} \mapsto v])))
\end{aligned}$$

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

$$\begin{aligned}
\hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket x \rrbracket^V &= x & \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket \alpha \rrbracket &= \alpha & \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket \bullet \rrbracket &= \bullet \\
\hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket \lambda x. t \rrbracket^V &= \lambda x. \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t \rrbracket & \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket * \rrbracket &= \lambda x. x \\
\hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket V \rrbracket &= \lambda k. k \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket V \rrbracket^V & \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [q^2] c \rrbracket &= [\hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket q^2 \rrbracket] \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket c \rrbracket \\
\hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t_1 t_2 \rrbracket &= \lambda k. \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t_1 \rrbracket \lambda f. \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t_2 \rrbracket \lambda s. f s k & \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [q] t \rrbracket &= \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t \rrbracket \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket q \rrbracket \\
\hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket \mu \alpha. c \rrbracket &= \lambda k. (\lambda \alpha. \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket c \rrbracket) k & \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [\hat{\alpha}]_0 \Delta. t \rrbracket &= (\lambda \langle \Delta, k \rangle. \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t \rrbracket k) \hat{\alpha} \\
\hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket \mu_0 \hat{\alpha}. c \rrbracket &= \lambda k. (\lambda \hat{\alpha}. \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket c \rrbracket) k & \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [\Delta] c \rrbracket &= [\Delta] \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket c \rrbracket
\end{aligned}$$

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

of the meta-context up to $\hat{\alpha}$ translates to capturing a prefix of the dynamic environment while unbinding $\hat{\alpha}$, and extending the meta-context becomes extending the dynamic environment. Like in $\hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}\hat{\tau}_0}$, the invocation of a prompt is changed due to the change in the way dynamic variable lookup is performed. The CPS transform for $\hat{\lambda}\hat{\mu}_0$ is an extension of the basic $\hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}}$ 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 $\mathcal{D}_{\hat{\lambda}_0} \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0}$ 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 $E[\mu \hat{\alpha}. c]$. Conversely, when a dynamic co-variable is removed from the meta-continuation through the lookup process, $[\hat{\alpha}]_0 \Delta. t$, 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. In the $\mathcal{D}_{\hat{\lambda}_0} \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0}$ 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

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

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

evaluation context finishes by explicitly returning a value to another context ($[\alpha]E$), invoking a dynamically bound context ($[\hat{\alpha}]E$), or exiting the program completely ($[*]E$). Additionally, both the context and meta-context have exactly one operation: plug in a value and continue evaluation, or lookup 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 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 $\lambda\hat{\mu}_0$ 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 $\lambda\hat{\mu}_0$ as we had with $\lambda\mu$, $\lambda\mu\hat{\text{tr}}$, $\lambda\hat{\mu}$, and $\lambda\mu\hat{\text{tr}}_0$.

Theorem 8.1 (Soundness)

If $M \twoheadrightarrow M'$ then $\mathcal{D}_{\lambda_0} \hat{\mathcal{E}}_{\lambda\hat{\mu}_0} \llbracket M \rrbracket =_{\beta\eta} \mathcal{D}_{\lambda_0} \hat{\mathcal{E}}_{\lambda\hat{\mu}_0} \llbracket M' \rrbracket$.

Theorem 8.2 (Evaluation)

If $\mathcal{D}_{\lambda_0} \hat{\mathcal{E}}_{\lambda\hat{\mu}_0} \llbracket c^2 \rrbracket =_{\beta} V$ then there is a final answer c'^2 such that $c^2 \mapsto c'^2$ and $\mathcal{D}_{\lambda_0} \hat{\mathcal{E}}_{\lambda\hat{\mu}_0} \llbracket c'^2 \rrbracket =_{\beta} V$.

A final answer of the $\lambda\hat{\mu}_0$ -calculus is a meta-command of the form $D^2[[*]V]$, the same as for $\lambda\hat{\mu}$ and $\lambda\mu\hat{\text{tr}}_0$. 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 $\#_0^{\hat{\alpha}} t$ 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 $\mathcal{S}_0^{\hat{\alpha}} V'$ is encountered while evaluating t , then the $\mathcal{S}_0^{\hat{\alpha}}$ captures the current context as well as the dynamic prefix up to the most recent binding of $\hat{\alpha}$, removing the prompt $\#_0^{\hat{\alpha}}$ 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 $\hat{\alpha}$.

$$\begin{aligned}
\langle [q^2]c \rangle_{\text{refocus}} &\rightsquigarrow \langle c, q^2 \rangle_{\text{refocus}} \\
\langle [q]t, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle t, q, F^{2*} \rangle_{\text{refocus}} \\
\langle [\Delta]c, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle c, F^{2*} @ \Delta \rangle_{\text{refocus}} \\
\langle [\hat{\alpha}]_0 \Delta.t, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle F^{2*}, \hat{\alpha}, \Delta.t \rangle_{\text{lookup}} \\
\langle t t', F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle t, F^*[\square t'], F^{2*} \rangle_{\text{refocus}} \\
\langle \mu \alpha.c, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle \mu \alpha.c, F^*, F^{2*} \rangle_{\text{reduce}} \\
\langle \mu_0 \hat{\alpha}.c, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle c, F^{2*}[\hat{\alpha} \mapsto F^*] \rangle_{\text{refocus}} \\
\langle V, F^*, F^{2*} \rangle_{\text{refocus}} &\rightsquigarrow \langle F^*, V, F^{2*} \rangle_{\text{apply}} \\
\\
\langle F^*[\square t], V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle t, F^*[V \square], F^{2*} \rangle_{\text{refocus}} \\
\langle F^*[V' \square], V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle V' V, F^*, F^{2*} \rangle_{\text{reduce}} \\
\langle *, V, F^{2*} \rangle_{\text{apply}} &\rightsquigarrow \langle V \rangle_{\text{done}} \\
\\
\langle (\lambda x.t) V, F^*, F^{2*} \rangle_{\text{reduce}} &\rightsquigarrow \langle t\{V/x\}, F^*, F^{2*} \rangle_{\text{refocus}} \\
\langle \mu \alpha.c, F^*, F^{2*} \rangle_{\text{reduce}} &\rightsquigarrow \langle c\{F^*/\alpha\}, F^{2*} \rangle_{\text{refocus}} \\
\langle \Delta.t, E^2, F^*, F^{2*} \rangle_{\text{reduce}} &\rightsquigarrow \langle t\{E^2/\Delta\}, F^*, F^{2*} \rangle_{\text{refocus}} \\
\\
\langle F^{2*}[\hat{\alpha} \mapsto F^*], \hat{\alpha}, L \rangle_{\text{lookup}} &\rightsquigarrow \langle L, [], F^*, F^{2*} \rangle_{\text{collect}} \\
\langle F^{2*}[\hat{\beta} \mapsto F^*], \hat{\alpha}, L \rangle_{\text{lookup}} &\rightsquigarrow \langle F^{2*}, \hat{\alpha}, L[\hat{\beta} \mapsto F^*] \rangle_{\text{lookup}} \\
\langle F^{2*} @ E'^2, \hat{\alpha}, L \rangle_{\text{lookup}} &\rightsquigarrow \langle F^{2*} E'^2, \hat{\alpha}, L \rangle_{\text{lookup}} \\
\\
\langle \Delta.t, E^2, F^*, F^{2*} \rangle_{\text{collect}} &\rightsquigarrow \langle \Delta.t, E^2, F^*, F^{2*} \rangle_{\text{reduce}} \\
\langle L[\hat{\alpha} \mapsto F'^*], E^2, F^*, F^{2*} \rangle_{\text{collect}} &\rightsquigarrow \langle L, E^2[\hat{\alpha} \mapsto F'^*], F^*, F^{2*} \rangle_{\text{collect}}
\end{aligned}$$

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

$$\begin{aligned}
\#_0^{\hat{\alpha}} t &= \mu_0 \hat{\alpha}. [\hat{\alpha}] t \\
\mathcal{S}_0^{\hat{\alpha}} &= \lambda h. \mu \beta. [\hat{\alpha}]_0 \Delta. (h (\lambda x. \mu_0 \hat{\alpha}. [\Delta][\beta] x))
\end{aligned}$$

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

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

Example 11

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

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

$$\begin{aligned}
E &::= \square \mid E \ t \mid V \ E \\
D &::= \square \mid E[\#_0^{\hat{\alpha}} D] \\
D[E[(\lambda x.t) V]] &\mapsto D[E[t\{V/x\}]] \\
D[E[\#_0^{\hat{\alpha}} V]] &\mapsto D[E[V]] \\
D[E[\#_0^{\hat{\alpha}} D'_\alpha[E'[\mathcal{S}_0^{\hat{\alpha}} V]]]] &\mapsto D[E[V \ (\lambda x.\#_0^{\hat{\alpha}} D'_\alpha[E'[x]])]]
\end{aligned}$$

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

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

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

The first step is a β_v -reduction passing in the value V to the encoding of $\mathcal{S}_0^{\hat{\alpha}}$. The next two reductions capture the evaluation context immediately surrounding the call site of shift_0 , represented up 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 a reset_0 of $\hat{\alpha}$ 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 $\lambda\mu\hat{\text{tp}}$ -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, $\lambda\hat{\mu}$, $\lambda\mu\hat{\text{tp}}_0$, and $\lambda\hat{\mu}_0$, 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 \gg_T ³, with the types:

$$\text{return}_T : a \rightarrow T a \qquad \gg_T : T a \rightarrow (a \rightarrow T b) \rightarrow T b$$

The return_T function turns a value into a trivial T -computation that returns the value without performing any effects. The \gg_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 execution of the computation. Though not ordinary functions, these special operations for a particular monad T have the types:

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

In order to give the correct result, **reflect** acts as an ordinary call-by-value function and evaluates its argument first, whereas **reify** is a special form that first creates a barrier around t to isolate its effects before evaluating it. And furthermore, **reflect** and **reify** represent inverse operations from one another, and so they ought to obey the following laws for any monad:

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

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.

$$\begin{aligned} \mathbf{reify}_T t &= \#(\text{return}_T t) \\ \mathbf{reflect}_T t &= \mathcal{S}\lambda k.(t \gg_T k) \end{aligned}$$

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

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

$$\begin{aligned}\mathbf{reify}_T t &= \mu\hat{\text{tp}}.[\hat{\text{tp}}](\text{return}_T t) \\ \mathbf{reflect}_T t &= \mu\alpha.[\hat{\text{tp}}](t \gg\! =_T \lambda x.\mu\hat{\text{tp}}.[\alpha]x)\end{aligned}$$

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 **reflect**_T is used on some value V in a context where it has a surrounding reset.

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

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

$$\dots \#E[\mathcal{S}\lambda k.(V \gg\! =_T k)] \dots$$

Now, according to the definition of the $\gg\! =_T$ operator, V must have type $T a$ and the function k must have the type $a \rightarrow T b$. Since k is a functional representation of the evaluation context E , that means that E is expecting a value of type a , and so the entire expression **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 \mathcal{S} delivers the result of $V \gg\! =_T k$ 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 , **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 **reflect**_T, is expected to give a monadic value.

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

1. Define the type⁵ 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.
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

⁴ For a more formal discussion on type and effect systems for delimited control, see Danvy & Filinski (1989); Filinski (1999); 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.

encountered, the rest of the current computation is abandoned and the exception is raised.

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

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

$$\text{raise}' = \lambda z. \text{Exn } z$$

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

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

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 $\text{Error}_e a$ and checking for whether or not an exception was raised. An SML-like exception handling mechanism can be defined as:

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

Using the reduction semantics of $\lambda \mu_{\widehat{\text{tp}}}$, 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.

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

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:

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

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

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

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

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.

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

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

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

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

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

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:

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

The expression **alloc** V **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 $\lambda\mu\hat{\text{tp}}$, we get the following rules for mutable state:

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

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.

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

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.

$$\begin{aligned} \mathbf{alloc} V \mathbf{in} E[\mathbf{get}] &= (\mu\hat{\text{tp}}.[\hat{\text{tp}}]\text{return}_{\text{State}_s} E[\mu\alpha.[\hat{\text{tp}}]\lambda z.(\mu\hat{\text{tp}}.[\alpha]z) z]) V \\ &\rightarrow (\mu\hat{\text{tp}}.[\hat{\text{tp}}](\lambda z.(\mu\hat{\text{tp}}.[\hat{\text{tp}}]\text{return}_{\text{State}_s} E[z]) z)) V \\ &\rightarrow (\lambda z.(\mu\hat{\text{tp}}.[\hat{\text{tp}}]\text{return}_{\text{State}_s} E[z]) z) V \\ &\rightarrow (\mu\hat{\text{tp}}.[\hat{\text{tp}}]\text{return}_{\text{State}_s} E[V]) V \\ &= \mathbf{alloc} V \mathbf{in} E[V] \end{aligned}$$

$$\begin{aligned} \mathbf{alloc} V \mathbf{in} E[\mathbf{put} V'] &\rightarrow (\mu\hat{\text{tp}}.[\hat{\text{tp}}]\text{return}_{\text{State}_s} E[\mu\alpha.[\hat{\text{tp}}]\lambda z.(\mu\hat{\text{tp}}.[\alpha]()) V']) V \\ &\rightarrow (\mu\hat{\text{tp}}.[\hat{\text{tp}}]\lambda z.(\mu\hat{\text{tp}}.[\hat{\text{tp}}]\text{return}_{\text{State}_s} E[()] V')) V \\ &\rightarrow (\lambda z.(\mu\hat{\text{tp}}.[\hat{\text{tp}}]\text{return}_{\text{State}_s} E[()] V') V) V \\ &\rightarrow (\mu\hat{\text{tp}}.[\hat{\text{tp}}]\text{return}_{\text{State}_s} E[()]) V' \\ &= \mathbf{alloc} V' \mathbf{in} E[()] \end{aligned}$$

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

$$E ::= \square \mid E t \mid V E$$

$$D ::= \square \mid E[\mathbf{alloc} V \mathbf{in} D]$$

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

$$D[E[\mathbf{alloc} V \mathbf{in} V']] \mapsto D[E[(V, V')]]$$

$$D[E[\mathbf{alloc} V \mathbf{in} E'[\mathbf{get}]]] \mapsto D[E[\mathbf{alloc} V \mathbf{in} E'[V]]]$$

$$D[E[\mathbf{alloc} V \mathbf{in} E'[\mathbf{put} V']]] \mapsto D[E[\mathbf{alloc} V' \mathbf{in} E'[()]]]$$

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 $\lambda\hat{\mu}$. The key difference is to extend the **raise** and **handle** primitives from using only $\hat{\text{tp}}$ to instead allow any dynamic co-variable.

$$\begin{aligned} \text{raise } \hat{\alpha} &= \lambda z. \mu_{-}. [\hat{\alpha}] \text{Exn } z \\ t \text{ handle } \hat{\alpha} z \Rightarrow u &= \text{case } \mu_{\hat{\alpha}}. [\hat{\alpha}] \text{OK } t \text{ of OK } x \Rightarrow x \mid \text{Exn } z \Rightarrow u \end{aligned}$$

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

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

$$\begin{aligned} V \text{ handle } \hat{\alpha} z \Rightarrow t &\rightarrow V \\ E[\text{raise } \hat{\alpha} V] \text{ handle } \hat{\alpha} z \Rightarrow t &\rightarrow t\{V/z\} \\ E[\text{raise } \hat{\alpha} V] \text{ handle } \hat{\beta} z \Rightarrow t &\rightarrow \text{raise } \hat{\alpha} V \quad \text{where } \hat{\alpha} \neq \hat{\beta} \end{aligned}$$

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 $\lambda\mu\hat{\text{tp}}$. When an $\hat{\alpha}$ exception is raised inside a handler for $\hat{\alpha}$, the handler stops the exception and takes its recovery branch.

$$\begin{aligned} E[\text{raise } \hat{\alpha} V] \text{ handle } \hat{\alpha} z \Rightarrow t & \\ \rightarrow \text{case } (\mu_{\hat{\alpha}}. [\hat{\alpha}] \text{OK } E[\mu_{-}. [\hat{\alpha}] \text{Exn } V]) \text{ of OK } x \Rightarrow x \mid \text{Exn } z \Rightarrow t & \\ \rightarrow \text{case } (\mu_{\hat{\alpha}}. [\hat{\alpha}] \text{Exn } V) \text{ of OK } x \Rightarrow x \mid \text{Exn } z \Rightarrow t & \\ \rightarrow \text{case } \text{Exn } V \text{ of OK } x \Rightarrow x \mid \text{Exn } z \Rightarrow t & \\ \rightarrow t\{V/z\} & \end{aligned}$$

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

$$\begin{aligned} E[\text{raise } \hat{\alpha} V] \text{ handle } \hat{\beta} z \Rightarrow t & \\ \rightarrow \text{case } (\mu_{\hat{\beta}}. [\hat{\beta}] \text{OK } E[\mu_{-}. [\hat{\alpha}] \text{Exn } V]) \text{ of OK } x \Rightarrow x \mid \text{Exn } z \Rightarrow t & \\ \rightarrow \text{case } (\mu_{\hat{\beta}}. [\hat{\alpha}] \text{Exn } V) \text{ of OK } x \Rightarrow x \mid \text{Exn } z \Rightarrow t & \\ \rightarrow \text{case } (\mu_{-}. [\hat{\alpha}] \text{Exn } V) \text{ of OK } x \Rightarrow x \mid \text{Exn } z \Rightarrow t & \\ \rightarrow \mu_{-}. [\hat{\alpha}] \text{Exn } V & \\ = \text{raise } \hat{\alpha} V & \end{aligned}$$

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

$$\begin{aligned}
E &::= \square \mid E \ t \mid V \ E \mid \text{raise } \hat{\alpha} \ E & D &::= \square \mid E[D\mathbf{handle} \ \hat{\alpha} \ x \Rightarrow u] \\
D[E[(\lambda x.t) \ V]] &\mapsto D[E[t\{V/x\}]] \\
D[E[V \ \mathbf{handle} \ \hat{\alpha} \ z \Rightarrow u]] &\mapsto D[E[V]] \\
D[E[(D'_{\hat{\alpha}}[E'[\text{raise } \hat{\alpha} \ V]]) \ \mathbf{handle} \ \hat{\alpha} \ z \Rightarrow u]] &\mapsto D[E[u\{V/z\}]]
\end{aligned}$$

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 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, and
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:

$$\#_3 E''[\#_2 E'[\#_1 E[\mathcal{S}_2 V]]] \mapsto \#_3 E''[\#_2 V (\lambda x.\#_2 E'[\#_1 E[x]])]$$

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 are all compiled down to shift_0 and reset_0 , which can access arbitrarily deep into the stack of meta-continuations.

In their encoding, Materzok and Biernacki 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 $\lambda \mu \hat{\text{tp}}_0$ from Section 7:

$$\begin{aligned} t \$ u &= \mathbf{let} k = t \mathbf{in} \#_0(\mathbf{let} x = u \mathbf{in} \mathcal{S}_0(\lambda _ . k x)) \\ &= \mathbf{let} k = t \mathbf{in} (\mu_0 \hat{\text{tp}}. [\hat{\text{tp}}](\mathbf{let} x = u \mathbf{in} \mu _ . [\hat{\text{tp}}]_0 k x)) \end{aligned}$$

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 $\mathcal{C}_{\lambda \mu \hat{\text{tp}}_0}$ (Figure 43) or the double CPS transforms using environments or functions, $\mathcal{C}_{\lambda \mu \hat{\text{tp}}_0}^\times$ (Figure 44) or $\mathcal{C}_{\lambda \mu \hat{\text{tp}}_0}^2$ (Figure 49), respectively.

$$\begin{aligned} (\lambda x. x) \$ t &\rightarrow \mu_0 \hat{\text{tp}}. [\hat{\text{tp}}] \mathbf{let} x = t \mathbf{in} \mu _ . [\hat{\text{tp}}]_0 x \\ \mathcal{C}_{\lambda \mu \hat{\text{tp}}_0} \llbracket \mu_0 \hat{\text{tp}}. [\hat{\text{tp}}] \mathbf{let} x = t \mathbf{in} \mu _ . [\hat{\text{tp}}]_0 x \rrbracket &\rightarrow \mathcal{C}_{\lambda \mu \hat{\text{tp}}_0} \llbracket \mu_0 \hat{\text{tp}}. [\hat{\text{tp}}] t \rrbracket \\ &= \lambda k. \mathcal{C}_{\lambda \mu \hat{\text{tp}}_0} \llbracket t \rrbracket (\lambda x. \lambda k'. k' x) k \end{aligned}$$

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

$$\begin{aligned} V \$ V' &= \mathbf{let} k = V \mathbf{in} \mu_0 \hat{\text{tp}}. [\hat{\text{tp}}] \mathbf{let} x = V' \mathbf{in} \mu _ . [\hat{\text{tp}}]_0 k x \\ &\rightarrow \mu_0 \hat{\text{tp}}. [\hat{\text{tp}}] \mu _ . [\hat{\text{tp}}]_0 V V' \\ &\rightarrow \mu_0 \hat{\text{tp}}. [\hat{\text{tp}}]_0 V V' \\ &\rightarrow V V' \end{aligned}$$

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

$$\begin{aligned} V \$ E[\mathcal{S}_0 V'] &\rightarrow \mu_0 \hat{\text{tp}}. [\hat{\text{tp}}] \mathbf{let} x = E[\mu \alpha. [\hat{\text{tp}}]_0 V' (\lambda y. \mu_0 \hat{\text{tp}}. [\alpha] y)] \mathbf{in} \mu _ . [\hat{\text{tp}}]_0 V x \\ &\rightarrow \mu_0 \hat{\text{tp}}. [\hat{\text{tp}}]_0 V' (\lambda y. \mu_0 \hat{\text{tp}}. [\hat{\text{tp}}] \mathbf{let} x = E[y] \mathbf{in} \mu _ . [\hat{\text{tp}}]_0 V x) \\ &\rightarrow V' (\lambda y. \mu_0 \hat{\text{tp}}. [\hat{\text{tp}}] \mathbf{let} x = E[y] \mathbf{in} \mu _ . [\hat{\text{tp}}]_0 V x) \\ &= V' (\lambda y. V \$ E[y]) \end{aligned}$$

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

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

In the expression $\mathcal{S}_i V$, the \mathcal{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.

$$\begin{aligned} \mathcal{S} &= \lambda h. \mathcal{S}_0 \lambda k. \#_0 (h k') \\ \mathbf{where} \ k' &= \lambda x. \mathcal{S}_0 \lambda q_1. \mathcal{S}_0 \lambda q_2. (\lambda w. q_2 \$ q_1 \$ w) \$ k \$ x \\ \#_1 t &= \mathcal{S}_0 \lambda q_1. \mathcal{S}_0 \lambda q_2. (\lambda w. q_2 \$ q_1 \$ w) \$ \#_0 t \end{aligned}$$

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 $\mathcal{C}_{\lambda \mu \hat{\text{tp}}_0}^2$:

$$\mathcal{C}_{\lambda \mu \hat{\text{tp}}_0}^2 \llbracket \# \# t \rrbracket =_{\beta \eta} \mathcal{C}_{\lambda \mu \hat{\text{tp}}_0}^2 \llbracket \# t \rrbracket$$

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

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

Note that in order to evaluate a term t using this encoding, it is important to provide the appropriate number of bindings for $\hat{\text{tp}}$. 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 $\mu_0 \hat{\text{tp}}. [\hat{\text{tp}}] \square$:

$$[\otimes] [\hat{\text{tp}}] (\mu_0 \hat{\text{tp}}. [\hat{\text{tp}}].^n. (t))$$

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 $\hat{\text{tp}}$ 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 $\hat{\text{tp}}$.

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

Extending get and put to handle multiple cells, however, is a more delicate operation. Not only does the cell with the matching name have to be found, but we need to be careful to not disrupt any other mutable cells that may have been allocated more recently. Therefore, in the encoding of $\text{get } \hat{\alpha}$ and $\text{put } \hat{\alpha}$, 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 $\lambda\hat{\mu}_0$ -calculus.

$$\begin{aligned} \text{get } \hat{\alpha} &= \mu\beta.[\hat{\alpha}]_0\Delta.(\lambda s.(\mu_0\hat{\alpha}.\Delta)[\beta]s) s \\ \text{put } \hat{\alpha} &= \lambda s'.\mu\beta.[\hat{\alpha}]_0\Delta.(\lambda_.(\mu_0\hat{\alpha}.\Delta)[\beta]()) s' \end{aligned}$$

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

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

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 $\lambda\hat{\mu}_0$, the simpler $\lambda\hat{\mu}$ would do. However, when we are looking for an older state cell that is not the most recently allocated one, the simple lookup of $\lambda\hat{\mu}$ 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:

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

which demonstrates how the intermediate allocation of the $\hat{\beta}$ state cell is remembered during access to the $\hat{\alpha}$ cell. Once the value V allocated in the cell $\hat{\alpha}$ is found, then the $\hat{\beta}$ 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:

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

$$\begin{aligned}
& \mathbf{alloc} \hat{\alpha} := 10 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} \\
& \left((\mathbf{put} \hat{\alpha} (\mathbf{get} \hat{\alpha} * \mathbf{get} \hat{\beta}); \mathbf{raise} \hat{\varepsilon} (\mathbf{get} \hat{\alpha})) \mathbf{handle} \hat{\varepsilon} x \Rightarrow x + \mathbf{get} \hat{\alpha} \right) \\
\mapsto & \mathbf{alloc} \hat{\alpha} := 10 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} \\
& \left((\mathbf{put} \hat{\alpha} (10 * 2); \mathbf{raise} \hat{\varepsilon} (\mathbf{get} \hat{\alpha})) \mathbf{handle} \hat{\varepsilon} x \Rightarrow x + \mathbf{get} \hat{\alpha} \right) \\
\mapsto & \mathbf{alloc} \hat{\alpha} := 20 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} \\
& \left((\mathbf{raise} \hat{\varepsilon} (\mathbf{get} \hat{\alpha})) \mathbf{handle} \hat{\varepsilon} x \Rightarrow x + \mathbf{get} \hat{\alpha} \right) \\
\mapsto & \mathbf{alloc} \hat{\alpha} := 20 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} \\
& \left((\mathbf{raise} \hat{\varepsilon} 20) \mathbf{handle} \hat{\varepsilon} x \Rightarrow x + \mathbf{get} \hat{\alpha} \right) \\
\mapsto & \mathbf{alloc} \hat{\alpha} := 20 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} (20 + \mathbf{get} \hat{\alpha}) \\
\mapsto & \mathbf{alloc} \hat{\alpha} := 20 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} (20 + 20) \\
\mapsto & \mathbf{alloc} \hat{\alpha} := 20 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} 40
\end{aligned}$$

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

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

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

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

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

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 is given as the simple union of the independent semantics, which is validated by the encodings of state and exceptions into the $\lambda\hat{\mu}_0$ -calculus. First, we show the typical case, where the mutable cells are stored globally, “outside” of 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

$$\begin{aligned}
& \mathbf{alloc} \hat{\alpha} := 10 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} \\
& \left(\left(\mathbf{alloc} \hat{\alpha} := 100 \mathbf{in} \right. \right. \\
& \quad \left. \left. \mathbf{put} \hat{\alpha} (\mathbf{get} \hat{\alpha} * \mathbf{get} \hat{\beta}); \mathbf{raise} \hat{\varepsilon} (\mathbf{get} \hat{\alpha}) \right) \mathbf{handle} \hat{\varepsilon} x \Rightarrow x + \mathbf{get} \hat{\alpha} \right) \\
& \mapsto \left(\left(\mathbf{alloc} \hat{\alpha} := 100 \mathbf{in} \right. \right. \\
& \quad \left. \left. \mathbf{put} \hat{\alpha} (100 * 2); \mathbf{raise} \hat{\varepsilon} (\mathbf{get} \hat{\alpha}) \right) \mathbf{handle} \hat{\varepsilon} x \Rightarrow x + \mathbf{get} \hat{\alpha} \right) \\
& \mapsto \left(\left(\mathbf{alloc} \hat{\alpha} := 200 \mathbf{in} \right. \right. \\
& \quad \left. \left. \mathbf{raise} \hat{\varepsilon} (\mathbf{get} \hat{\alpha}) \right) \mathbf{handle} \hat{\varepsilon} x \Rightarrow x + \mathbf{get} \hat{\alpha} \right) \\
& \mapsto \left(\left(\mathbf{alloc} \hat{\alpha} := 200 \mathbf{in} \right. \right. \\
& \quad \left. \left. \mathbf{raise} \hat{\varepsilon} 200 \right) \mathbf{handle} \hat{\varepsilon} x \Rightarrow x + \mathbf{get} \hat{\alpha} \right) \\
& \mapsto \mathbf{alloc} \hat{\alpha} := 10 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} (200 + \mathbf{get} \hat{\alpha}) \\
& \mapsto \mathbf{alloc} \hat{\alpha} := 10 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} (200 + 10) \\
& \mapsto \mathbf{alloc} \hat{\alpha} := 10 \mathbf{in} \mathbf{alloc} \hat{\beta} := 2 \mathbf{in} 210
\end{aligned}$$

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

mutable cells to exist within the program itself. In this case, internal allocations are subject to alterations of control flow in the program. The example in Figure 66 shows the same program with an internal re-allocation of the $\hat{\alpha}$ mutable cell. In this program, the internal allocation of $\hat{\alpha}$ overrides the external one until an exception is raised. At that point, the internal allocation of $\hat{\alpha}$ 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 of the bounds of an exception behaves like the usual notion of global state, and corresponds to the Haskell monad given by `ErrorT e (StateT s m) a`. On the other hand, an allocated cell inside of the bounds of an exception is discarded, and corresponds to the monad given by `StateT s (ErrorT e m) a`.

10 Conclusion

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 rebindable top-level

to the $\lambda\mu$ calculus, giving us the $\lambda\mu\widehat{\text{tp}}$ 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, rebindable top-levels as a simple extension, $\lambda\widehat{\mu}$, and which allowed us to express exception handling with more than one named exception. We also considered a variant of delimited control with dynamic access to prompts, $\lambda\mu\widehat{\text{tp}}_0$, 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 of 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 which allows us to study delimited control with multiple prompts. To do this, we used an intermediate language of dynamic binding in order to define the semantics of multiple prompts. Kiselyov *et al.* (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 of ours. They directly define the dynamic binding in terms of delimited control with multiple prompts. On the other hand, we use the conceptually simpler notion of dynamic binding as a stepping stone for understanding delimited control with multiple prompts.

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:

$$\begin{aligned} &(\text{reset}[(\text{abort}(\text{raise } 0) * 5) \mathbf{handle } n \Rightarrow n + 1]) \\ &\mathbf{handle } n \Rightarrow n + 2 \end{aligned}$$

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{tp} . If we define abort as:

$$\text{abort} = \lambda x. \mu_{-} . [\widehat{tp}] x$$

then due to the call-by-value semantics of function application, first raise_0 is evaluated, clearing the context $(\text{abort } \square) * 5$ and invoking the innermost exception handler and giving the result 1. If we define abort as:

$$\text{abort } t = \mu_{-} . [\widehat{tp}] t$$

then first abort takes control and clears the context $\square * 5$, but then evaluates raise_0 still inside the innermost handler and also gives the result 1. If we instead define abort as:

$$\text{abort } t = \mu_{-} . [\widehat{tp}]_{0 \dots} t$$

then abort clears the context $\square * 5$ and also removes the innermost handler while searching for reset. After abort finds and removes the reset, 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 the 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 $\lambda \mu$, $\lambda \mu \widehat{tp}$, and $\lambda \mu \widehat{tp}_0$ 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 $\lambda \mu \widehat{tp}_0$ *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. This research has been developed under INRIA Équipe Associée SEMACODE.

Bibliography

- Ariola, Zena M., & Herbelin, Hugo. (2008). Control reduction theories: The benefit of structural substitution. *Journal of functional programming*, **18**(May), 373–419.
- Ariola, Zena M., Herbelin, Hugo, & Sabry, Amr. (2009). A type-theoretic foundation of delimited continuations. *Higher-order and symbolic computation*, **22**(3), 233–273.
- Ariola, Zena M., Herbelin, Hugo, Herman, David, & Keith, Daniel. (2011). A robust implementation of delimited control. *Page 6 of: First international workshop on the theory and practice of delimited continuations.*

- Biernacka, Malgorzata, Biernacki, Dariusz, & Danvy, Olivier. (2005). An operational foundation for delimited continuations in the cps hierarchy. *Logical methods in computer science*, **1**(2).
- Danvy, Olivier. (2004). On evaluation contexts, continuations, and the rest of the computation. *Pages 13–23 of: Acm sigplan continuations workshop*.
- Danvy, Olivier, & Filinski, Andrzej. (1989). *A functional abstraction of typed contexts*. Tech. rept. 89/12. DIKU, University of Copenhagen, Copenhagen, Denmark.
- Danvy, Olivier, & Filinski, Andrzej. (1990). Abstracting control. *Pages 151–160 of: Proceedings of the 1990 acm conference on lisp and functional programming*. ACM Press.
- Danvy, Olivier, & Nielsen, Lasse R. (2003). A first-order one-pass cps transformation. *Theoretical computer science*, **308**(1), 239–257.
- Downen, Paul, & Ariola, Zena M. (2012). A systematic approach to delimited control with multiple prompts. *Pages 234–253 of: Proceedings of the 21st european symposium on programming*. Springer.
- Dybvig, R. Kent, Jones, Simon Peyton, & Sabry, Amr. (2007). A monadic framework for delimited continuations. *Journal of functional programming*, **17**(06), 687–730.
- Felleisen, Matthias. (1988). The theory and practice of first-class prompts. *Pages 180–190 of: Principles of programming languages '88*.
- Felleisen, Matthias. (1991). On the expressive power of programming languages. *Science of computer programming*, **17**(1-3), 35–75.
- Felleisen, Matthias, & Friedman, Daniel P. (1987). A reduction semantics for imperative higher-order languages. *Pages 206–223 of: Parallel architectures and languages europe*.
- Felleisen, Matthias, & Hieb, Robert. (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, **103**(2), 235–271.
- Felleisen, Matthias, Friedman, Daniel P., Kohlbecker, Eugene E., & Duba, Bruce F. (1987). A syntactic theory of sequential control. *Theoretical computer science*, **52**, 205–237.
- Felleisen, Matthias, Wand, Mitchell, Friedman, Daniel P., & Duba, Bruce F. (1988). Abstract continuations: A mathematical semantics for handling full jumps. *Pages 52–62 of: Lisp and functional programming*.
- Filinski, Andrzej. (1994). Representing monads. *Pages 446–457 of: Principles of programming languages '94*. ACM.
- Filinski, Andrzej. (1999). Representing layered monads. *Pages 175–188 of: Principles of programming languages '99*.
- Flatt, Matthew, Yu, Gang, Findler, Robert Bruce, & Felleisen, Matthias. (2007). Adding delimited and composable control to a production programming environment. *Pages 165–176 of: Proceedings of the 12th acm sigplan international conference on functional programming*, vol. 1.
- Gunter, Carl A., Rémy, Didier, & Riecke, Jon G. (1995). A generalization of exceptions and control in ML-like languages. *Pages 12–23 of: Functional programming languages and computer architecture '95*. New York, NY, USA: ACM.
- Kameyama, Yukiyoshi, & Hasegawa, Masahito. (2003). A sound and complete axiomatization of delimited continuations. *Pages 177–188 of: Proceedings of the eighth acm sigplan international conference on functional programming*. ICFP '03. New York, NY, USA: ACM.

- Kiselyov, Oleg. (2010). Delimited control in OCaml, abstractly and concretely: System description. *Functional and logic programming*, 304–320.
- Kiselyov, Oleg, Shan, Chung-chieh, & Sabry, Amr. (2006). Delimited dynamic binding. *Pages 26–37 of: Proceedings of the eleventh acm sigplan international conference on functional programming*. ICFP '06. New York, NY, USA: ACM.
- Materzok, Marek, & Biernacki, Dariusz. (2011). Subtyping delimited continuations. *Pages 81–93 of: Proceeding of the 16th acm sigplan international conference on functional programming*. ICFP '11. New York, NY, USA: ACM.
- Materzok, Marek, & Biernacki, Dariusz. (2012). A dynamic interpretation of the cps hierarchy. *Pages 296–311 of: Programming languages and systems - 10th asian symposium, aplas 2012*.
- Moggi, Eugenio. (1989). Computational lambda-calculus and monads. *Pages 14–23 of: Proceedings of the fourth annual symposium on logic in computer science*. IEEE.
- Moggi, Eugenio. (1991). Notions of computation and monads. *Information and computation*, **93**(1), 55–92.
- Moreau, Luc. (1998). A syntactic theory of dynamic binding. *Higher-order and symbolic computation*, **11**(3), 233–279.
- Parigot, Michel. (1992). Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. *Pages 190–201 of: Logic programming and automated reasoning*. Springer.
- Plotkin, Gordon D. (1975). Call-by-name, call-by-value, and the λ -calculus. *Theoretical computer science*, **1**, 125–159.
- Reynolds, John C. (1972). Definitional interpreters for higher-order programming languages. *Pages 717–740 of: Proceedings of the 25th acm national conference*.
- Sabry, Amr, & Felleisen, Matthias. (1992). Reasoning about programs in continuation-passing style. *Pages 288–298 of: Lisp and functional programming*.
- Sabry, Amr, & Felleisen, Matthias. (1993). Reasoning about programs in continuation-passing style. *Lisp and symbolic computation*, **6**(3-4), 289–360.
- Shan, Chung-chieh. (2007). A static simulation of dynamic delimited control. *Higher-order and symbolic computation*, **20**(4), 371–401.
- Sitaram, Dorai, & Felleisen, Matthias. (1990a). Control delimiters and their hierarchies. *Lisp and symbolic computation*, **3**(1), 67–99.
- Sitaram, Dorai, & Felleisen, Matthias. (1990b). Reasoning with continuations II: Full abstraction for models of control. *Conference on lisp and functional programming*.

A Correctness of the semantics of control

We now show that semantics for the various languages of control are correct, in the sense that the different presentations of the semantics for a language correspond with one another. The reduction theories and operational semantics are already closely related, and similarly for the abstract machines and CPS transforms. Therefore, we focus on the relationship between the reduction theories and CPS transforms of the languages, since they are the furthest removed from one another. The first property that we consider is the *soundness* of the reduction theory with respect to the CPS transform, so that the reductions “make sense” according to the meaning given by transform. This property ensures that every reduction in the source language relates equal programs in the target language.

Property A.1 (Soundness)

If $M \rightarrow M'$ then $\llbracket M \rrbracket = \llbracket M' \rrbracket$.

In addition, we would also like to know that the reductions are *complete* in some sense, so that we have “enough” reductions to reach a final answer if there is one. This is an operational notion of completeness, which guarantees that whenever the CPS transform finds a final answer then the operational semantics is strong enough to also reach a final answer in the source language.

Property A.2 (Evaluation)

If $\llbracket M \rrbracket = V$ then there is a final answer M' such that $M \mapsto M'$ and $\llbracket M' \rrbracket = V$.

Note that we are considering languages with control effects that allow a program to jump in and out of a context during evaluation, and may appear to “return” a value multiple times to the same location. Therefore, we emphasize the fact that the answer is *final*, so that the program is committed to a single answer and may not change its mind by returning a different value at some later point in time. Finally, to finish the comparison between the semantics, we would also like to know the fact that the operational semantics defines a subset of the reduction theory.

Property A.3

If $M \mapsto M'$ then $M \rightarrow M'$.

Together, these two properties ensure that the reduction theory is strong enough to find a final answer if one is given by the CPS transform.

Since we are considering many different languages, each of which extends a previous one in some way, we develop the proofs of correctness according to a pattern that is repeated for each language. We therefore give the most attention the $\lambda\mu$ -calculus, the simplest language of control, establishing the general pattern for the proofs. Since the syntax, reduction theory, and CPS transform of the pure λ -calculus is a closed subset of the $\lambda\mu$ -calculus, Theorems 2.1 and 2.2 are a consequence of Theorems 3.1 and 3.2 for the $\lambda\mu$ -calculus. From there, we show how the previous proofs may be extended to show the correctness of the remaining languages of control, giving details where the language under consideration differs.

$$\begin{array}{ll}
 \mathcal{C}_{\lambda\mu}[[[q]t]] = \mathcal{C}_{\lambda\mu}[[t]] \mathcal{C}_{\lambda\mu}[[q]] & \mathcal{C}_{\lambda\mu}[[*]] = \bar{\lambda}x.x \\
 \mathcal{C}_{\lambda\mu}[[V]] = \bar{\lambda}k.k \mathcal{C}_{\lambda\mu}[[V]]^V & \mathcal{C}_{\lambda\mu}[[\alpha]] = \alpha \\
 \mathcal{C}_{\lambda\mu}[[t_1 t_2]] = \bar{\lambda}k.\mathcal{C}_{\lambda\mu}[[t_1]]\bar{\lambda}f.\mathcal{C}_{\lambda\mu}[[t_2]]\bar{\lambda}s.f s k & \mathcal{C}_{\lambda\mu}[[x]]^V = x \\
 \mathcal{C}_{\lambda\mu}[[\mu\alpha.c]] = \bar{\lambda}k.(\lambda\alpha.\mathcal{C}_{\lambda\mu}[[c]]) k & \mathcal{C}_{\lambda\mu}[[\lambda x.t]]^V = \lambda x.\mathcal{C}_{\lambda\mu}[[t]]
 \end{array}$$

Fig. A 1. The $\mathcal{C}_{\lambda\mu}$ transform with annotations on administrative $\bar{\lambda}$ -abstractions.

A.1 Soundness of the $\lambda\mu$ reduction theory

Not all reductions in the $\mathcal{C}_{\lambda\mu}$ transform correspond to a step in the $\lambda\mu$ calculus. Some of the additional steps performed in the CPS transformed program deal with more administrative concerns such as plugging a value into an evaluation context or returning the final answer, which are hidden at the level of the source language. In terms of the abstract machine for $\lambda\mu$, refocus and apply steps are administrative reductions in the CPS transform and reduce steps are non-administrative reductions.

We can keep track of which reductions are administrative in nature by marking the functions with a $\bar{\lambda}$, in the style of Sabry & Felleisen (1993), so that the administrative reductions are only those β -reductions of the following form:

$$\bar{\beta} : (\bar{\lambda}x.M) V \Rightarrow M\{V/x\}$$

Notation: We use \Rightarrow to denote an administrative reduction performed in an evaluation context, and use \Rightarrow and \Rightarrow to denote the reflexive, transitive closure of \Rightarrow and \Rightarrow , respectively.

We determine the administrative $\bar{\lambda}$ -abstractions by finding the abstractions that introduce new variables that were not present in the original source program. In the CPS transform for $\lambda\mu$ from Figure 12, four clauses introduce administrative $\bar{\lambda}$ -abstractions, and the annotated transform is given in Figure A 1. Observe that the translation $\mathcal{C}_{\lambda\mu}[[\mu\alpha.c]]$ introduces a non-administrative binding for its continuation k , since the co-variable naming it is present in the source program. However, many freshly transformed $\lambda\mu$ programs contain active administrative reductions that mask the real work that needs to be done. For the $\mathcal{C}_{\lambda\mu}$ transform, we have the following administrative reductions:

$$\begin{aligned}
 \mathcal{C}_{\lambda\mu}[[[*]V]] &= (\bar{\lambda}x.x) \mathcal{C}_{\lambda\mu}[[V]]^V \Rightarrow \mathcal{C}_{\lambda\mu}[[V]]^V \\
 \mathcal{C}_{\lambda\mu}[[V V']] &= \bar{\lambda}k.(\bar{\lambda}k'.k' \mathcal{C}_{\lambda\mu}[[V]]^V) (\bar{\lambda}f.(\bar{\lambda}k''.k'' \mathcal{C}_{\lambda\mu}[[V']]^V) (\bar{\lambda}s.f s k)) \\
 &\Rightarrow \bar{\lambda}k.\mathcal{C}_{\lambda\mu}[[V]]^V \mathcal{C}_{\lambda\mu}[[V']]^V k
 \end{aligned}$$

When tracking reduction in the CPS program, it's useful to have a transform that is free of administrative reductions. That is to say, an alternate CPS transform where immediately after transforming a $\lambda\mu$ program, no administrative reductions apply. The alternate transform, $\bar{\mathcal{C}}_{\lambda\mu}$, is achieved by taking the administrative-normal form of $\mathcal{C}_{\lambda\mu}$, by performing all available administrative reductions until there are none left.

$$\mathcal{C}_{\lambda\mu}[[c]] \Rightarrow \bar{\mathcal{C}}_{\lambda\mu}[[c]] \not\Rightarrow$$

$$\begin{aligned}
\mathcal{C}_{\lambda\mu}[[q]E] &= \mathcal{C}_{\lambda\mu}[E] \mathcal{C}_{\lambda\mu}[q] \\
\mathcal{C}_{\lambda\mu}[[\square]k] &= k \\
\mathcal{C}_{\lambda\mu}[E \ t]k &= \mathcal{C}_{\lambda\mu}[E] \bar{\lambda}f. \mathcal{C}_{\lambda\mu}[t] \bar{\lambda}s. f \ s \ k \\
\mathcal{C}_{\lambda\mu}[V \ E]k &= \mathcal{C}_{\lambda\mu}[E] \bar{\lambda}s. \mathcal{C}_{\lambda\mu}[V]^V \ s \ k
\end{aligned}$$

Fig. A 2. CPS transform of call-by-value evaluation contexts in the $\lambda\mu$ -calculus.

A one-pass, administrative-normal CPS transform for the pure λ -calculus has been given by Sabry & Felleisen (1993) as well as Danvy & Nielsen (2003). However, in contrast to some previous administrative-normal CPS transforms like the one by Sabry & Felleisen (1992), the transform we describe above is unable to eliminate every single administrative reduction throughout the future of the program. After performing some real computation, some administrative steps may later become available. For example, since we follow Plotkin's (1975) call-by-value CPS transform, functions take their argument first and continuation second. Therefore, in the CPS program:

$$\bar{\mathcal{C}}_{\lambda\mu}[(\lambda x.x) \ y]k = (\lambda x.\bar{\lambda}k.k \ x) \ y \ k$$

the λ -abstraction for the argument x blocks the administrative $\bar{\beta}$ reduction on the continuation k . Furthermore, we have the more fundamental issue that μ -abstractions in the source calculus introduce non-administrative bindings for continuations in the CPS program. This fact alone prevents us from eliminating certain future administrative reductions during the transformation, since any μ -abstraction in a source program blocks a continuation from seeing the value returned to it. For instance, consider the CPS transformation of the command $[\alpha](f(\mu\beta.[\beta]x))$:

$$\bar{\mathcal{C}}_{\lambda\mu}[[\alpha](f(\mu\beta.[\beta]x))] = (\lambda\beta.\beta \ x) (\bar{\lambda}y.f \ y \ \alpha)$$

We must first perform a non-administrative reduction to move the continuation to the place it is used before we are able to substitute x for y .

In order to show the soundness and completeness of evaluation of $\lambda\mu$ reductions with respect to the $\mathcal{C}_{\lambda\mu}$ transform, we need a couple properties of the transform. The first is a formal statement of the notion that in a CPS transformed program, the standard redex is lifted out of its evaluation context to the top of the term. Consider the CPS transform of evaluation contexts in $\lambda\mu$ shown in Figure A 2. We have that in the CPS transform, terms are lifted out of their evaluation context, after performing any necessary administrative reductions.

Lemma A.1.1 (Lifting)

$$\mathcal{C}_{\lambda\mu}[D[t]] \Rightarrow \mathcal{C}_{\lambda\mu}[t] \mathcal{C}_{\lambda\mu}[D] \text{ and } \mathcal{C}_{\lambda\mu}[E[t]] \ k \Rightarrow \mathcal{C}_{\lambda\mu}[t] (\mathcal{C}_{\lambda\mu}[E]k)$$

Proof

By induction on D and E .

- $D = [q]E$:

$$\mathcal{C}_{\lambda\mu}[[q](E[t])] = \mathcal{C}_{\lambda\mu}[E[t]] \mathcal{C}_{\lambda\mu}[q] \Rightarrow \mathcal{C}_{\lambda\mu}[t] (\mathcal{C}_{\lambda\mu}[[q]E])$$

- $E = \square$:

$$\mathcal{C}_{\lambda\mu} \llbracket t \rrbracket k = \mathcal{C}_{\lambda\mu} \llbracket t \rrbracket (\mathcal{C}_{\lambda\mu} \llbracket \square \rrbracket k)$$

- $E = E' t'$:

$$\mathcal{C}_{\lambda\mu} \llbracket (E' t') t' \rrbracket k \Rightarrow \mathcal{C}_{\lambda\mu} \llbracket E' t' \rrbracket (\bar{\lambda}f. \mathcal{C}_{\lambda\mu} \llbracket t' \rrbracket \bar{\lambda}s.f s k) \Rightarrow \mathcal{C}_{\lambda\mu} \llbracket t \rrbracket (\mathcal{C}_{\lambda\mu} \llbracket E' t' \rrbracket k)$$

- $E = V E'$:

$$\begin{aligned} \mathcal{C}_{\lambda\mu} \llbracket V (E' t') \rrbracket k &\Rightarrow \mathcal{C}_{\lambda\mu} \llbracket V \rrbracket (\bar{\lambda}f. \mathcal{C}_{\lambda\mu} \llbracket E' t' \rrbracket \bar{\lambda}s.f s k) \\ &\Rightarrow \mathcal{C}_{\lambda\mu} \llbracket E' t' \rrbracket (\bar{\lambda}s. \mathcal{C}_{\lambda\mu} \llbracket V \rrbracket^V s k) \\ &\Rightarrow \mathcal{C}_{\lambda\mu} \llbracket t \rrbracket (\mathcal{C}_{\lambda\mu} \llbracket V E' \rrbracket k) \end{aligned}$$

□

The second property we need for the $\mathcal{C}_{\lambda\mu}$ transform is that substitution in the $\lambda\mu$ calculus corresponds to substitution performed in the CPS language. Since we have three different notions of substitution in the $\lambda\mu$ calculus, we have three correlations. From now on, the meta-syntactic variable M ranges over terms and commands.

Lemma A.1.2 (Substitution)

- $\mathcal{C}_{\lambda\mu} \llbracket M \{V/x\} \rrbracket = \mathcal{C}_{\lambda\mu} \llbracket M \rrbracket \{\mathcal{C}_{\lambda\mu} \llbracket V \rrbracket^V / x\}$
- $\mathcal{C}_{\lambda\mu} \llbracket M \{q/\alpha\} \rrbracket = \mathcal{C}_{\lambda\mu} \llbracket M \rrbracket \{\mathcal{C}_{\lambda\mu} \llbracket q \rrbracket / \alpha\}$
- $\mathcal{C}_{\lambda\mu} \llbracket M \{D[t]/[\alpha]t\} \rrbracket \Rightarrow \mathcal{C}_{\lambda\mu} \llbracket M \rrbracket \{\mathcal{C}_{\lambda\mu} \llbracket D \rrbracket / \alpha\}$

Proof

By induction on M . The only interesting cases are the ones in which a (co-)variable is being replaced by substitution.

- $M = x, \{V/x\}$:

$$\mathcal{C}_{\lambda\mu} \llbracket x \{V/x\} \rrbracket^V = \mathcal{C}_{\lambda\mu} \llbracket V \rrbracket^V = \mathcal{C}_{\lambda\mu} \llbracket x \rrbracket^V \{\mathcal{C}_{\lambda\mu} \llbracket V \rrbracket^V / x\}$$

- $M = \alpha, \{q/\alpha\}$:

$$\mathcal{C}_{\lambda\mu} \llbracket \alpha \{q/\alpha\} \rrbracket = \mathcal{C}_{\lambda\mu} \llbracket q \rrbracket = \mathcal{C}_{\lambda\mu} \llbracket \alpha \rrbracket \{\mathcal{C}_{\lambda\mu} \llbracket q \rrbracket / \alpha\}$$

- $M = [\alpha]t, \{D[t]/[\alpha]t\}$:

$$\begin{aligned} \mathcal{C}_{\lambda\mu} \llbracket ([\alpha]t) \{D[t]/[\alpha]t\} \rrbracket &= \mathcal{C}_{\lambda\mu} \llbracket D \{D[t]/[\alpha]t\} \rrbracket \\ &\Rightarrow \mathcal{C}_{\lambda\mu} \llbracket t \rrbracket \{\mathcal{C}_{\lambda\mu} \llbracket D \rrbracket / \alpha\} \mathcal{C}_{\lambda\mu} \llbracket D \rrbracket \\ &= \mathcal{C}_{\lambda\mu} \llbracket [\alpha]t \rrbracket \{\mathcal{C}_{\lambda\mu} \llbracket D \rrbracket / \alpha\} \end{aligned}$$

□

With the lifting and substitution lemmas, proving the soundness of a single step of the $\lambda\mu$ reduction rules with respect to $\mathcal{C}_{\lambda\mu}$ is a straightforward calculation in the λ -calculus, using the usual notion of β equality.

Lemma A.1.3 (One-step soundness)

If $M \rightarrow M'$ then $\mathcal{C}_{\lambda\mu} \llbracket M \rrbracket =_{\beta} \mathcal{C}_{\lambda\mu} \llbracket M' \rrbracket$.

Proof

By cases on the reduction $M \rightarrow M'$ in the $\lambda\mu$ -calculus. Recall that the transform $\mathcal{C}_{\lambda\mu}[[t]]$ is always an administrative $\bar{\lambda}$ -abstraction of the form $\bar{\lambda}k.M$, for some CPS program M .

- $(\lambda x.t) V \rightarrow t\{V/x\}$:

$$\begin{aligned} \mathcal{C}_{\lambda\mu}[(\lambda x.t) V] &\Rightarrow \bar{\lambda}k.(\lambda x.\mathcal{C}_{\lambda\mu}[[t]]) \mathcal{C}_{\lambda\mu}[[V]]^V k \\ &\rightarrow \bar{\lambda}k.\mathcal{C}_{\lambda\mu}[[t]]\{\mathcal{C}_{\lambda\mu}[[V]]^V/x\} k \\ &= \bar{\lambda}k.\mathcal{C}_{\lambda\mu}[[t\{V/x\}]] k && \text{(by Lemma A.1.2)} \\ &\Rightarrow \mathcal{C}_{\lambda\mu}[[t\{V/x\}]] \end{aligned}$$

- $F[\mu\alpha.c] \rightarrow \mu\alpha'.c\{[\alpha'](F[t])/[\alpha]t\}$:

$$\begin{aligned} \mathcal{C}_{\lambda\mu}[[F[\mu\alpha.c]]] &\Rightarrow \bar{\lambda}k.(\lambda\alpha.\mathcal{C}_{\lambda\mu}[[c]]) (\mathcal{C}_{\lambda\mu}[[F]]k) && \text{(by Lemma A.1.1)} \\ &\rightarrow \bar{\lambda}k.\mathcal{C}_{\lambda\mu}[[c]]\{\mathcal{C}_{\lambda\mu}[[F]]k/\alpha\} \\ &\leftarrow \bar{\lambda}k.(\lambda\alpha'.\mathcal{C}_{\lambda\mu}[[c]]\{\mathcal{C}_{\lambda\mu}[[\alpha']F]/\alpha\}) k \\ &\Leftarrow \bar{\lambda}k.(\lambda\alpha'.\mathcal{C}_{\lambda\mu}[[c\{[\alpha'](F[t])/[\alpha]t\}]] k && \text{(by Lemma A.1.2)} \\ &= \mathcal{C}_{\lambda\mu}[[\mu\alpha'.c\{[\alpha']F[t]/[\alpha]t\}]] \end{aligned}$$

- $[q]\mu\alpha.c \rightarrow c\{q/\alpha\}$:

$$\begin{aligned} \mathcal{C}_{\lambda\mu}[[[q]\mu\alpha.c]] &\Rightarrow (\lambda\alpha.\mathcal{C}_{\lambda\mu}[[c]]) \mathcal{C}_{\lambda\mu}[[q]] \\ &\rightarrow \mathcal{C}_{\lambda\mu}[[c]]\{\mathcal{C}_{\lambda\mu}[[q]]/\alpha\} \\ &= \mathcal{C}_{\lambda\mu}[[c\{q/\alpha\}]] && \text{(by Lemma A.1.2)} \end{aligned}$$

Note that the administrative reduction:

$$\bar{\lambda}k.\mathcal{C}_{\lambda\mu}[[t]] k \Rightarrow \mathcal{C}_{\lambda\mu}[[t]]$$

follows from the $\bar{\beta}$ rule since the transformation of every term t , shown in Figure A 1, has the form $\bar{\lambda}k.M$, for some CPS term M .

Since the CPS transform $\mathcal{C}_{\lambda\mu}$ is compositional, every context in $\lambda\mu$ is transformed into a context in the resulting λ -calculus term, and therefore the $\lambda\mu$ reduction rules apply in every context. \square

Observe that the reductions for the $\lambda\mu$ -calculus are conceptually more fine-grained than the β reductions produced by the $\mathcal{C}_{\lambda\mu}$ transform. Specifically, using the $\lambda\mu$ reduction rules, the term $\mu\alpha.c$ captures its evaluation context one small piece at a time. However, in the CPS program, the translation of $\mu\alpha.c$ captures the continuation representing its complete evaluation context in one step. Therefore, to simulate the more fine-grained reduction in the CPS program, we must “backtrack” capturing the tail of the continuation, so that only the first piece is taken. Soundness for the general $\lambda\mu$ reduction theory, as stated in Theorem 3.1, follows by straightforward induction on the reduction sequence.

Theorem A.1.1 (Soundness)

If $M \twoheadrightarrow M'$ then $\mathcal{C}_{\lambda\mu}[[M]] =_{\beta} \mathcal{C}_{\lambda\mu}[[M']]$.

Proof

By Lemma A.1.3 and induction on the definition of $M \twoheadrightarrow M'$ as the reflexive, transitive closure of \rightarrow . \square

$$\begin{aligned}
 c_{fa} \in \text{FinalAnswer} &::= [*]V \\
 c_{whnf} \in \text{WHNF} &::= c_{fa} \mid [\alpha]V \mid D[x V]
 \end{aligned}$$

Fig. A 3. Final answers and weak head-normal forms in the $\lambda\mu$ -calculus.

A.2 Operational completeness of the $\lambda\mu$ reduction theory

We would also like to show that the operational semantics (and therefore the reduction rules) are strong enough to reach any answer that the CPS is capable of producing. The set of commands that constitutes some form of “answer” is given in Figure A 3. In $\lambda\mu$, a *final answer* is a command of the form $[*]V$. Intuitively, the answer V is considered “final” because the program is now committed, and cannot change its mind by using control effects to jump somewhere else and produce a different answer. Additionally, a *weak head-normal form* (whnf) is either a final answer or a stuck command of the form $[\alpha]V$ or $D[x V]$, where x and α are free (co-)variables. Observe that with this definition of weak head-normal forms and complete evaluation contexts, we have the unique decomposition property for the $\lambda\mu$ -calculus.

Property A.2.1 (Unique decomposition)

Every $\lambda\mu$ command is either a weak head-normal form, or it decomposes into a complete evaluation context D surrounding a redex, *i.e.* $D[(\lambda x.t) V]$, or a μ -abstraction, *i.e.* $D[\mu\alpha.c]$.

To start, we need a few facts about the administrative subset of the λ -calculus.

Lemma A.2.1

1. Administrative reduction in the λ -calculus is confluent.
2. If $M_1 \Rightarrow M_2$ and $M_1 \mapsto M'_1$ by a non-administrative reduction, then there is a M'_2 such that $M_2 \mapsto M'_2$ by a non-administrative reduction and $M'_1 \Rightarrow M'_2$.

Proof

For the first lemma, confluence follows from confluence of basic λ -calculus, see Sabry & Felleisen (1993) for details. For the second lemma, we have that $M_1 = E[(\lambda x.M) V]$ and the following diagram commutes (purely administrative reduction sequences are labeled as $\bar{\beta}$):

$$\begin{array}{ccc}
 M_1 = E[(\lambda x.M) V] & \longmapsto & M'_1 = E[M\{V/x\}] \\
 \downarrow \bar{\beta} & & \downarrow \bar{\beta} \\
 M_2 = E[(\lambda x.M') V'] & \longmapsto & M'_2 = E[M'\{V'/x\}]
 \end{array}$$

□

We also need a stronger correspondence of substitution in the source and target programs according to the administrative-normal CPS transform $\mathcal{C}_{\lambda\mu}$.

Lemma A.2.2 (Administrative-normal substitution)

- $\mathcal{C}_{\lambda\mu}[M\{V/x\}] \Leftarrow \mathcal{C}_{\lambda\mu}[M]\{\mathcal{C}_{\lambda\mu}[V]^V/x\}$
- $\mathcal{C}_{\lambda\mu}[M\{q/\alpha\}] \Leftarrow \mathcal{C}_{\lambda\mu}[M]\{\mathcal{C}_{\lambda\mu}[q]/\alpha\}$
- $\mathcal{C}_{\lambda\mu}[M\{D[t]/[\alpha]t\}] \Leftarrow \mathcal{C}_{\lambda\mu}[M]\{\mathcal{C}_{\lambda\mu}[D]/\alpha\}$

Proof

We first show the case for structural substitution of evaluation contexts. By definition of $\overline{\mathcal{C}}_{\lambda\mu}$, Lemma A.1.2, and confluence of administrative reduction (Lemma A.2.1), there is an M' such that the following diagram commutes (where all reductions are administrative only):

$$\begin{array}{ccc} \mathcal{C}_{\lambda\mu} \llbracket M\{D[t]/[\alpha]t\} \rrbracket & \longrightarrow & \mathcal{C}_{\lambda\mu} \llbracket M \rrbracket \{ \mathcal{C}_{\lambda\mu} \llbracket D \rrbracket / \alpha \} \\ \downarrow & & \downarrow \\ \overline{\mathcal{C}}_{\lambda\mu} \llbracket M\{D[t]/[\alpha]t\} \rrbracket & \longrightarrow & M' \longleftarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket M \rrbracket \{ \overline{\mathcal{C}}_{\lambda\mu} \llbracket D \rrbracket / \alpha \} \end{array}$$

Furthermore, since $\overline{\mathcal{C}}_{\lambda\mu} \llbracket M\{D[t]/[\alpha]t\} \rrbracket$ is an administrative normal form by definition, we have that $\overline{\mathcal{C}}_{\lambda\mu} \llbracket M\{D[t]/[\alpha]t\} \rrbracket = M' \longleftarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket M \rrbracket \{ \overline{\mathcal{C}}_{\lambda\mu} \llbracket D \rrbracket / \alpha \}$.

The cases for substitution of values and co-terms follow analogously. \square

We now show that non-administrative reductions in the CPS actually correspond to reductions in $\lambda\mu$, and that end states (values and whnfs) in the CPS correspond to end states in $\lambda\mu$.

Lemma A.2.3 (One-step evaluation)

- If $\overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket$ is a value then c is a final answer.
- If $\overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket$ is a whnf then c is a whnf.
- If $\overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket \mapsto M'$ then there is a $\lambda\mu$ command c' such that $c \mapsto c'$ and $M' \Rightarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket c' \rrbracket$.

Proof

By cases on c , using the unique decomposition property.

- $c = [*]V$:

$$\overline{\mathcal{C}}_{\lambda\mu} \llbracket [*]V \rrbracket = \overline{\mathcal{C}}_{\lambda\mu} \llbracket V \rrbracket^V$$

- $c = [\alpha]V$:

$$\overline{\mathcal{C}}_{\lambda\mu} \llbracket [\alpha]V \rrbracket = \alpha \overline{\mathcal{C}}_{\lambda\mu} \llbracket V \rrbracket^V$$

- $c = D[x V]$:

$$\overline{\mathcal{C}}_{\lambda\mu} \llbracket D[x V] \rrbracket = x \overline{\mathcal{C}}_{\lambda\mu} \llbracket V \rrbracket^V \overline{\mathcal{C}}_{\lambda\mu} \llbracket D \rrbracket \quad (\text{by Lemma A.1.1})$$

- $c = D[(\lambda x.t) V]$:

$$\begin{aligned} \overline{\mathcal{C}}_{\lambda\mu} \llbracket D[(\lambda x.t) V] \rrbracket &= (\lambda x. \overline{\mathcal{C}}_{\lambda\mu} \llbracket t \rrbracket) \overline{\mathcal{C}}_{\lambda\mu} \llbracket V \rrbracket^V \overline{\mathcal{C}}_{\lambda\mu} \llbracket D \rrbracket && (\text{by Lemma A.1.1}) \\ &\mapsto \overline{\mathcal{C}}_{\lambda\mu} \llbracket t \rrbracket \{ \overline{\mathcal{C}}_{\lambda\mu} \llbracket V \rrbracket^V / x \} \overline{\mathcal{C}}_{\lambda\mu} \llbracket D \rrbracket \\ &\Rightarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket t \{ V/x \} \rrbracket \overline{\mathcal{C}}_{\lambda\mu} \llbracket D \rrbracket && (\text{by Lemma A.2.2}) \\ &\Rightarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket D[t \{ V/x \}] \rrbracket \end{aligned}$$

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

- $c = D[\mu\alpha.c]$:

$$\begin{aligned} \overline{\mathcal{C}}_{\lambda\mu} \llbracket D[\mu\alpha.c] \rrbracket &= (\lambda\alpha. \overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket) \overline{\mathcal{C}}_{\lambda\mu} \llbracket D \rrbracket && \text{(by Lemma A.1.1)} \\ &\mapsto \overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket \{ \overline{\mathcal{C}}_{\lambda\mu} \llbracket D \rrbracket / \alpha \} \\ &\Rightarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket c \{ D[t] / [\alpha]t \} \rrbracket && \text{(by Lemma A.2.2)} \end{aligned}$$

$$D[\mu\alpha.c] \mapsto c \{ D[t] / [\alpha]t \}$$

□

From the fact that steps and end states in the CPS program correspond properly in the source program, we can prove the main evaluation lemma for operational completeness of the call-by-value $\lambda\mu$ -calculus.

Lemma A.2.4

If $M \Rightarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket$ and $M \mapsto M'$, then there is a $\lambda\mu$ command c' such that $c \mapsto c'$ and $M' \Rightarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket c' \rrbracket$. Furthermore, if M' is a whnf then c' is a whnf, and if M' is a value then c' is a final answer.

Proof

By induction on the non-administrative reductions in $M \mapsto M'$.

- In the case that $M \mapsto M'$ by administrative reductions only, then we have $c' = c$ and $M' \Rightarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket c' \rrbracket$.

Additionally by Lemma A.2.3, we have that if M' is a whnf then $\overline{\mathcal{C}}_{\lambda\mu} \llbracket c' \rrbracket$ is also a whnf, and so c' is a whnf. Also, if M' is a value then $\overline{\mathcal{C}}_{\lambda\mu} \llbracket c' \rrbracket$ is also a value, and so c' is a final answer.

- Otherwise, we have the case where $M \mapsto M_1$ and $M_1 \mapsto M'_1$ by a non-administrative reduction. We have that $M_1 \Rightarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket$ by Lemma A.2.1 (confluence) and the fact that $\overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket$ is an administrative-normal form. Also by Lemma A.2.1, we have that there is an M'_2 such that $\overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket \mapsto M'_2$ and $M'_1 \Rightarrow M'_2$. By Lemma A.2.3, we know that there is a c'' such that $c \mapsto c''$ and $M'_2 \Rightarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket c'' \rrbracket$. Applying the induction hypothesis to $M'_1 \Rightarrow M'_2 \Rightarrow \overline{\mathcal{C}}_{\lambda\mu} \llbracket c'' \rrbracket$ and $M'_1 \mapsto M'$ finishes the proof.

This reasoning is summed up by the following commutative diagram (purely administrative reduction sequences are labeled as $\overline{\beta}$):

$$\begin{array}{ccccc} c & \xrightarrow{\quad} & c'' & \xrightarrow{\quad} & c' \\ \parallel & & \parallel & & \parallel \\ \overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket & \xrightarrow{\quad} & \overline{\mathcal{C}}_{\lambda\mu} \llbracket c'' \rrbracket & & \overline{\mathcal{C}}_{\lambda\mu} \llbracket c' \rrbracket \\ \uparrow \overline{\beta} & & \uparrow \overline{\beta} & & \uparrow \overline{\beta} \\ \overline{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket & \xrightarrow{\quad} & M'_2 & & \\ \uparrow \overline{\beta} & \swarrow \overline{\beta} & \uparrow \overline{\beta} & & \\ M & \xrightarrow{\quad} & M_1 & \xrightarrow{\quad} & M'_1 & \xrightarrow{\quad} & M' \end{array}$$

$$\begin{aligned} \mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2 \llbracket \mu\hat{\tau}\hat{\rho}.c \rrbracket &= \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2 \llbracket c \rrbracket \lambda x.k x \gamma & \mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2 \llbracket \hat{\tau}\hat{\rho} \rrbracket &= \bar{\lambda}x.\bar{\lambda}\gamma.\gamma x \\ \mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2 \llbracket * \rrbracket &= \bar{\lambda}x.\bar{\lambda}\gamma.x \end{aligned}$$

Fig. A 4. The administrative $\bar{\lambda}$ -abstractions of the $\mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2$ transform.

□

Theorem A.2.1 (Evaluation)

If $\mathcal{C}_{\lambda\mu} \llbracket c \rrbracket =_{\beta} V$ then there is a final answer c' such that $c \mapsto c'$ and $\mathcal{C}_{\lambda\mu} \llbracket c' \rrbracket =_{\beta} V$.

Proof

First, recall that $\mathcal{C}_{\lambda\mu} \llbracket c \rrbracket \Rightarrow \bar{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket$, so that $\bar{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket =_{\beta} V$. By confluence and the fact that values are closed under reduction in the λ -calculus, we know that there is a V' such that $\bar{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket \twoheadrightarrow V'$. Furthermore, by standardization of reduction in the λ -calculus (both call-by-name or call-by-value standard reduction is the same for CPS programs), there is a V'' such that $\bar{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket \mapsto V''$. From applying Lemma A.2.4 to $\mathcal{C}_{\lambda\mu} \llbracket c \rrbracket \Rightarrow \bar{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket$ and $\bar{\mathcal{C}}_{\lambda\mu} \llbracket c \rrbracket \mapsto V''$, we have a final answer c' such that $c \mapsto c'$ and $V'' \Rightarrow \mathcal{C}_{\lambda\mu} \llbracket c' \rrbracket$. Finally, by transitivity of β -equivalence in the λ -calculus we have that $\mathcal{C}_{\lambda\mu} \llbracket c' \rrbracket =_{\beta} V'' =_{\beta} V$. □

Finally, the fine-grained reductions presented in Figure 10 are capable of simulating the operational semantics of $\lambda\mu$ given in Figure 11.

Theorem A.2.2

If $c \mapsto c'$ then $c \twoheadrightarrow c'$.

Proof

By induction on reduction sequence $c \mapsto c'$ as the reflexive, transitive closure of \mapsto . The result is immediate for β -reduction on functions by applying the reduction rule inside the evaluation context. In the case that a μ -abstraction captures its context,

$$D[\mu\alpha.c] \mapsto c\{D[t]/[\alpha]t\}$$

the simulation is obtained by induction on D . □

Therefore, from Theorems A.2.1 and A.2.2, we have that the reduction theory is capable of finding any final answer given by the CPS transform.

A.3 Soundness and operational completeness of the $\lambda\mu\hat{\tau}\hat{\rho}$ reduction theory

The administrative $\bar{\lambda}$ -abstractions in the $\mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2$ transformation are those from Figure A 1 as well as the new ones shown in Figure A 4. Observe that on the one hand, the clauses of the $\mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2$ transform inherited from the $\mathcal{C}_{\lambda\mu}$ transform in Figure A 1 do not make use of meta-continuation, γ , but just implicitly pass it along. On the other hand, the clauses of $\mathcal{C}_{\lambda\mu\hat{\tau}\hat{\rho}}^2$ from Figure A 4 explicitly introduce and manipulate a meta-continuation due to the behavior of the dynamic $\hat{\tau}\hat{\rho}$. We could η -expand the $\mathcal{C}_{\lambda\mu}$ transform to remove this difference in CPS programs, but choose not to so that the CPS transform of purely $\lambda\mu$ programs remains unchanged. Instead, in order to mediate between implicit and explicit uses of the meta-continuation, we use η -reduction while reasoning about CPS programs.

In fact, we only need a very restricted form of usual notion of η -reduction that eliminates trivial $\bar{\lambda}$ -abstractions in the place of a meta-continuation:

$$\bar{\eta} : \bar{\lambda}k.\bar{\lambda}\gamma.M \gamma \Rightarrow \bar{\lambda}k.M$$

where γ is not a free variable of M . It follows that the reduction rules of $\lambda\mu\hat{\text{tp}}$, which include the reductions from Figure 16 along with the additional rule:

$$\mu\hat{\text{tp}}.[*]V \rightarrow \mu_{-}.[*]V$$

are sound with respect to $\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2$ up to $\beta\eta$ -equality in the λ -calculus.

Lemma A.3.1 (One-step soundness)

If $M \rightarrow M'$ then $\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket M \rrbracket =_{\beta\eta} \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket M' \rrbracket$.

Proof

By cases on the reduction $M \rightarrow M'$ in the $\lambda\mu\hat{\text{tp}}$ -calculus:

- $\mu\hat{\text{tp}}.[\hat{\text{tp}}]V \rightarrow V$:

$$\begin{aligned} \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket \mu\hat{\text{tp}}.[\hat{\text{tp}}]V \rrbracket &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket [\hat{\text{tp}}] \rrbracket \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket V \rrbracket^V (\lambda x.k x \gamma) \\ &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\lambda x.k x \gamma) \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket V \rrbracket^V \\ &\rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.k \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket V \rrbracket^V \gamma \\ &\Rightarrow_{\bar{\eta}} \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket V \rrbracket \end{aligned}$$

- $\mu\hat{\text{tp}}.[*]V \rightarrow \mu_{-}.[*]V$:

$$\begin{aligned} \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket \mu\hat{\text{tp}}.[*]V \rrbracket &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket [*] \rrbracket \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket V \rrbracket^V (\lambda x.k x \gamma) \\ &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket V \rrbracket^V \\ &\Leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket [*] \rrbracket \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket V \rrbracket^V \gamma \\ &\Leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\lambda_{-}.\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket [*] \rrbracket \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket V \rrbracket^V) k \gamma \\ &\Rightarrow_{\bar{\eta}} \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket \mu_{-}.[*]V \rrbracket \end{aligned}$$

The rest of the cases are the same as for the proof of Lemma A.1.3 for $\lambda\mu$. \square

Note how $\bar{\eta}$ reduction is used to prove soundness of the $\lambda\mu\hat{\text{tp}}$ reduction theory. This allows us to equate CPS terms that do not mention a meta-continuation with those that pass along their meta-continuation unchanged, as in $\bar{\lambda}\gamma.k \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket V \rrbracket^V \gamma$. Soundness of the reduction theory, as stated in Theorem 4.1, follows by induction.

Theorem A.3.1 (Soundness)

If $M \twoheadrightarrow M'$ then $\mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket M \rrbracket =_{\beta\eta} \mathcal{C}_{\lambda\mu\hat{\text{tp}}}^2 \llbracket M' \rrbracket$.

Since $\lambda\mu\hat{\text{tp}}$ operates on meta-commands, final answers in $\lambda\mu\hat{\text{tp}}$ are correspondingly meta-commands of the form $D^2 \llbracket [*]V \rrbracket$, as shown in Figure A.5. We must also expand the set of weak head normal forms to accommodate the new syntax, so that the whnfs of $\lambda\mu\hat{\text{tp}}$ are final answers or meta-commands of the form $\llbracket \bullet \rrbracket [\hat{\text{tp}}]V$, $D^2 \llbracket [\alpha]V \rrbracket$, or $D^2 \llbracket D[x V] \rrbracket$, where α and x are free (co-)variables. Likewise, the unique decomposition property is also lifted into meta-commands in the $\lambda\mu\hat{\text{tp}}$ -calculus.

$$c_{fa} \in \text{FinalAnswer} ::= D^2[[*]V]$$

$$c_{whnf} \in \text{WHNF} ::= c_{fa} \mid [\bullet][\widehat{\text{tp}}]V \mid D^2[[\alpha]V] \mid D^2[D[xV]]$$

Fig. A 5. Final answers and weak head-normal forms in the $\lambda\mu\widehat{\text{tp}}$ -calculus.

$$\begin{aligned} \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[q^2]E^2] &= \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[E^2] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[q^2] \\ \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[\square]\gamma &= \gamma \\ \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[D[\mu\widehat{\text{tp}}.E^2]]\gamma &= \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[E^2]\lambda x. \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[D] x \gamma \end{aligned}$$

Fig. A 6. CPS transform of call-by-value evaluation meta-contexts in the $\lambda\mu\widehat{\text{tp}}$ -calculus.*Property A.3.1 (Unique decomposition)*

Every $\lambda\mu\widehat{\text{tp}}$ meta-command is either a weak head-normal form, or it decomposes into a complete evaluation meta-context and complete evaluation context, D^2 and D , surrounding a redex, i.e. $D^2[D[(\lambda x.t) V]]$, or a μ -abstraction, i.e. $D^2[D[\mu\alpha.c]]$.

Note that the reduction

$$\mu\widehat{\text{tp}}.[*]V \rightarrow \mu_.[*]V$$

does not occur during standard reduction, but instead equates final answers by discarding unnecessary information in the meta-context.

Just like we needed to use the fact that terms are lifted out of their evaluation context in the $\mathcal{C}_{\lambda\mu}$ transform, we also need to show that commands are lifted out of their evaluation meta-contexts in the $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2$ transform. Evaluation meta-contexts are transformed as shown in Figure A 6. This gives us a similar result for evaluation meta-contexts in the $\lambda\mu\widehat{\text{tp}}$ -calculus.

Lemma A.3.2 (Meta-lifting)

$$\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[D^2[c]] \Rightarrow \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[c] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[D^2] \text{ and } \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[E^2[c]]\gamma \Rightarrow \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[c] (\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[E^2]\gamma)$$

Proof

By induction on D^2 and E^2 .

- $D^2 = [q^2]E^2$:

$$\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[q^2]E^2[c]] = \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[E^2[c]] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[q^2] \Rightarrow \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[c] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[q^2]E^2]$$

- $E^2 = \square$:

$$\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[c]\gamma = \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[c] (\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[\square]\gamma)$$

- $E^2 = D[\mu\widehat{\text{tp}}.E'^2]$:

$$\begin{aligned} \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[D[\mu\widehat{\text{tp}}.E'^2][c]]\gamma &\Rightarrow \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[\mu\widehat{\text{tp}}.E'^2[c]] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[D] \gamma \\ &\Rightarrow \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[E'^2[c]]\lambda x. \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[D] x \gamma \\ &\Rightarrow \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[c] (\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[D[\mu\widehat{\text{tp}}.E'^2]]\gamma) \end{aligned}$$

□

The proof that the $\lambda\mu\hat{\text{tp}}$ operational and reduction rules are complete with respect to evaluation of $\mathcal{C}_{\lambda\mu\hat{\text{tp}}}$ follows the same strategy as for $\lambda\mu$.

Lemma A.3.3 (One-step evaluation)

- If $\overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[c^2]]$ is a value then c^2 is a final answer.
- If $\overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[c^2]]$ is a whnf then c^2 is a whnf.
- If $\overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[c^2]] \mapsto M'$ by a non-administrative reduction then there is a $\lambda\mu\hat{\text{tp}}$ meta-command c'^2 such that $c^2 \mapsto c'^2$ and $M' \Rightarrow \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[c'^2]]$.

Proof

By cases on c^2 , using the unique decomposition property.

- $c^2 = D^2[[*]V]$:

$$\overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[D^2[[*]V]]] = \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[V]]^V$$

- $c^2 = [\bullet][\hat{\text{tp}}]V$:

$$\overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[[\bullet][\hat{\text{tp}}]V]] = \gamma_0 \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[V]]^V$$

- $c^2 = D^2[D[\mu\hat{\text{tp}}.[\hat{\text{tp}}]V]]$:

$$\begin{aligned} & \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[D^2[D[\mu\hat{\text{tp}}.[\hat{\text{tp}}]V]]]] \\ &= (\lambda x. \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[D[x]]] \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[D^2]]) \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[V]]^V \quad (\text{by Lemmas A.1.1 and A.3.2}) \\ &= (\lambda x. \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[D^2[D[x]]]]) \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[V]]^V \quad (\text{by Lemma A.3.2}) \\ &\mapsto \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[D^2[D[x]]]] \{ \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[V]]^V / x \} \\ &\Rightarrow \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[D^2[D[V]]]] \quad (\text{by Lemma A.2.2}) \end{aligned}$$

$$D^2[D[\mu\hat{\text{tp}}.[\hat{\text{tp}}]V]] \mapsto D^2[D[V]]$$

The rest of the cases are the same as the proof of Lemma A.2.3 for $\lambda\mu$, except lifted into an arbitrary evaluation meta-context D^2 . \square

Lemma A.3.4

If $M \Rightarrow \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[c^2]]$ and $M \mapsto M'$, then there is a $\lambda\mu\hat{\text{tp}}$ meta-command c'^2 such that $c^2 \mapsto c'^2$ and $M' \Rightarrow \overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[c'^2]]$. Furthermore, if M' is a whnf then c'^2 is a whnf, and if M' is a value then c'^2 is a final answer.

Proof

The same as the proof of Lemma A.2.4 for $\lambda\mu$. \square

Finally, we have a proof of the operational completeness of $\lambda\mu\hat{\text{tp}}$ as a strengthened version of Theorem 4.2.

Theorem A.3.2 (Evaluation)

If $\overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[c^2]] =_{\beta} V$ then there is a final answer c'^2 such that $c^2 \mapsto c'^2$ and $\overline{\mathcal{C}}_{\lambda\mu\hat{\text{tp}}}^2[[c'^2]] =_{\beta} V$.

Proof

$$\begin{aligned} \mathcal{D}_{\hat{\lambda}} \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}}[\mu\hat{\alpha}.c] &= \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{D}_{\hat{\lambda}} \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}}[c]\gamma[\hat{\alpha} \mapsto k] & \mathcal{D}_{\hat{\lambda}} \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}}[\hat{\alpha}] &= \bar{\lambda}x.\bar{\lambda}\gamma.\gamma(\hat{\alpha}) x \\ \gamma[\hat{\alpha} \mapsto k] &= \bar{\lambda}p.\mathbf{if} p \equiv \ulcorner \hat{\alpha} \urcorner \mathbf{then} \lambda x.k x \mathbf{else} \gamma p \end{aligned}$$

Fig. A 7. The administrative $\bar{\lambda}$ -abstractions in the $\lambda\hat{\mu}$ transform.

The same as the proof of Theorem A.2.1 for $\lambda\mu$. \square

Theorem A.3.3

If $c^2 \mapsto c'^2$ then $c^2 \twoheadrightarrow c'^2$.

Proof

By induction on reduction sequence $c^2 \mapsto c'^2$ as the reflexive, transitive closure of \mapsto . The result is immediate for reduction of $\mu\hat{\text{tr}}.[\hat{\text{tr}}]V$. The rest of the proof is the same as the proof of Theorem A.2.2 for $\lambda\mu$. \square

A.4 Soundness and operational completeness of the $\lambda\hat{\mu}$ reduction theory

The administrative $\bar{\lambda}$ -abstractions in the $\lambda\hat{\mu}$ two-pass, $\mathcal{D}_{\hat{\lambda}} \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}}$ transform are the same ones given for $\lambda\mu\hat{\text{tr}}$ in Figure A 4 with the changes shown in Figure A 7. In addition, for purposes of classifying reductions in the CPS program, reducing the if-then-else expression is also considered administrative.

The reduction rules for $\lambda\hat{\mu}$ are sound with respect to the transform $\mathcal{D}_{\hat{\lambda}} \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}}$, and the operational semantics is complete with respect to evaluation of the transform.

Lemma A.4.1 (One-step soundness)

If $M \rightarrow M'$ then $\mathcal{D}_{\hat{\lambda}} \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}}[M] =_{\beta\eta} \mathcal{D}_{\hat{\lambda}} \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}}[M']$.

Proof

By cases on the reduction $M \rightarrow M'$ in the $\lambda\hat{\mu}$ -calculus. For the purpose of conciseness, in this proof the transform $\mathcal{D}_{\hat{\lambda}} \hat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}}[M]$ is referred to as just $\llbracket M \rrbracket$.

- $\mu\hat{\alpha}.[\hat{\alpha}]V \rightarrow V$:

$$\begin{aligned} \llbracket \mu\hat{\alpha}.[\hat{\alpha}]V \rrbracket &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\gamma[\hat{\alpha} \mapsto k])(\hat{\alpha}) \llbracket V \rrbracket^V \\ &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\lambda x.k x \gamma) \llbracket V \rrbracket^V \\ &\rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.k \llbracket V \rrbracket^V \gamma \\ &\Rightarrow_{\eta} \llbracket V \rrbracket \end{aligned}$$

- $\mu\hat{\alpha}.[\hat{\beta}]V \rightarrow \mu_{-}.[\hat{\beta}]V$:

$$\begin{aligned} \llbracket \mu\hat{\alpha}.[\hat{\beta}]V \rrbracket &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\llbracket \hat{\beta} \rrbracket \llbracket V \rrbracket^V (\gamma[\hat{\alpha} \mapsto k]) \\ &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\gamma[\hat{\alpha} \mapsto k])(\hat{\beta}) \llbracket V \rrbracket^V \\ &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\gamma(\hat{\beta}) \llbracket V \rrbracket^V \\ &\Leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.\llbracket \hat{\beta} \rrbracket \llbracket V \rrbracket^V \gamma \\ &\leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\lambda_{-}.\llbracket \hat{\beta} \rrbracket \llbracket V \rrbracket^V) k \gamma \\ &\Rightarrow_{\eta} \llbracket \mu_{-}.[\hat{\beta}]V \rrbracket \end{aligned}$$

$$c_{fa} \in \text{FinalAnswer} ::= D^2[[*]V]$$

$$c_{whnf} \in \text{WHNF} ::= c_{fa} \mid D_{\hat{\alpha}}^2[[\hat{\alpha}]V] \mid D^2[[\alpha]V] \mid D^2[D[x V]]$$

Fig. A 8. Final answers and weak head-normal forms in the $\lambda\hat{\mu}$ -calculus.

$$\mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[[q^2]E^2] = \mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[E^2] \mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[q^2]$$

$$\mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[\square]\gamma = \gamma$$

$$\mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[D[\mu\hat{\alpha}.E^2]]\gamma = \mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[E^2] \gamma[\hat{\alpha} \mapsto \mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[D]]$$

Fig. A 9. CPS transform of call-by-value evaluation contexts of the $\lambda\hat{\mu}$ -calculus.

- $\mu\hat{\alpha}.[*]V \rightarrow \mu_{-}.[*]V$:

$$\begin{aligned} [[\mu\hat{\alpha}.[*]V]] &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.[*] [[V]]^V (\gamma[\hat{\alpha} \mapsto k]) \\ &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.[V]^V \\ &\Leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.[*] [[V]]^V \gamma \\ &\Leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\lambda_{-}.[*] [[V]]^V) k \gamma \\ &\Rightarrow_{\bar{\eta}} [[\mu_{-}.[*]V]] k \gamma \end{aligned}$$

The rest of the cases are the same as for the proof of Lemma A.3.1 for $\lambda\mu\hat{\text{tp}}$. \square

Theorem A.4.1 (Soundness)

If $M \twoheadrightarrow M'$ then $\mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[M] =_{\beta\eta} \mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[M']$.

The final answers and weak head-normal forms of $\lambda\hat{\mu}$ are shown in Figure A 8 and are the same as those from Figure A 5 for $\lambda\mu\hat{\text{tp}}$, except that whnfs $D_{\hat{\alpha}}^2[[\hat{\alpha}]V]$ are generalized to any meta-context $D_{\hat{\alpha}}^2$ which does not bind $\hat{\alpha}$.

To accommodate the addition of multiple dynamic co-variables in the $\mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}$ transform, we need to generalize the transform of meta-contexts to dynamic binding environments, as shown in Figure A 9. With this change, we still have the normal lifting property, as well as the fact that dynamic co-variable lookup behaves as expected.

Lemma A.4.2 (Meta-lifting)

For the transform $\mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}, [D^2[c]] \mapsto [c] [D^2]$ and $[E^2[c]]\gamma \mapsto [c] ([E^2]\gamma)$.

Proof

By induction on D^2 and E^2 . The proof follows the same form as the meta-lifting lemma for $\lambda\mu\hat{\text{tp}}$. \square

Lemma A.4.3 (Lookup)

If E^2 does not contain a binding for $\hat{\alpha}$, then $\mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[D^2[E^2]](\hat{\alpha}) \mapsto \mathcal{D}_{\lambda}\hat{\mathcal{C}}_{\lambda\hat{\mu}}[D^2](\hat{\alpha})$.

Proof

By induction on E^2 . \square

With these lemmas, proving completeness of the operational rules with respect to evaluation in the transform follows as it did for $\lambda\mu\hat{\text{tp}}$.

Lemma A.4.4 (One-step evaluation)

- If $\overline{\mathcal{D}}_{\lambda} \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c^2]$ is a value then c^2 is a final answer.
- If $\overline{\mathcal{D}}_{\lambda} \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c^2]$ is a whnf then c^2 is a whnf.
- If $\overline{\mathcal{D}}_{\lambda} \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c^2] \mapsto M'$ by a non-administrative reduction then there is a $\lambda\hat{\mu}$ meta-command c'^2 such that $c^2 \mapsto c'^2$ and $M' \Rightarrow \overline{\mathcal{D}}_{\lambda} \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c'^2]$.

Proof

By cases on c^2 , using the unique decomposition property. For the purpose of conciseness, in this proof the transform $\overline{\mathcal{D}}_{\lambda} \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[M]$ is referred to as just $\llbracket M \rrbracket$.

- $c^2 = [\bullet]E^2[\llbracket \hat{\alpha} \rrbracket V]$, where E^2 does not contain a binding for $\hat{\alpha}$:

$$\llbracket [\bullet]E^2[\llbracket \hat{\alpha} \rrbracket V] \rrbracket = \gamma_0(\hat{\alpha}) \llbracket V \rrbracket^V \quad (\text{by Lemmas A.4.2 and A.4.3})$$

- $c^2 = D^2[D[\mu\hat{\alpha}.E^2[\llbracket \hat{\alpha} \rrbracket V]]]$, where E^2 does not contain a binding for $\hat{\alpha}$:

$$\begin{aligned} & \llbracket D^2[D[\mu\hat{\alpha}.E^2[\llbracket \hat{\alpha} \rrbracket V]]] \rrbracket \\ &= \llbracket \hat{\alpha} \rrbracket \llbracket V \rrbracket^V (\llbracket D^2 \rrbracket [\hat{\alpha} \mapsto \llbracket D \rrbracket]) \\ &= (\llbracket D^2 \rrbracket [\hat{\alpha} \mapsto \llbracket D \rrbracket]) (\hat{\alpha}) \llbracket V \rrbracket^V \quad (\text{by Lemmas A.1.1 and A.4.2}) \\ &= (\lambda x. \llbracket D \rrbracket x \llbracket D^2 \rrbracket) \llbracket V \rrbracket^V \\ &= (\lambda x. \llbracket D^2[D[x]] \rrbracket) \llbracket V \rrbracket^V \quad (\text{by Lemmas A.1.1 and A.4.2}) \\ &\mapsto \llbracket D^2[D[x]] \rrbracket \{ \llbracket V \rrbracket^V / x \} \\ &\Rightarrow \llbracket D^2[D[V]] \rrbracket \quad (\text{by Lemma A.1.2}) \end{aligned}$$

$$D^2[D[\mu\hat{\alpha}.E^2[\llbracket \hat{\alpha} \rrbracket V]]] \mapsto D^2[D[V]]$$

The rest of the cases are the same as for the proof of Lemma A.3.3 for $\lambda\mu\hat{\text{tp}}$. \square

Lemma A.4.5

If $M \Rightarrow \overline{\mathcal{D}}_{\lambda} \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c^2]$ and $M \mapsto M'$, then there is a $\lambda\mu\hat{\text{tp}}$ meta-command c'^2 such that $c^2 \mapsto c'^2$ and $M' \Rightarrow \overline{\mathcal{D}}_{\lambda} \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c'^2]$. Furthermore, if M' is a whnf then c'^2 is a whnf, and if M' is a value then c'^2 is a final answer.

Proof

The same as the proof of Lemma A.2.4 for $\lambda\mu$. \square

Theorem A.4.2 (Evaluation)

If $\overline{\mathcal{D}}_{\lambda} \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c^2] =_{\beta} V$ then there is a final answer c'^2 such that $c^2 \mapsto c'^2$ and $\overline{\mathcal{D}}_{\lambda} \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c'^2] =_{\beta} V$.

Proof

The same as the proof of Theorem A.2.1 for $\lambda\mu$. \square

Theorem A.4.3

If $c^2 \mapsto c'^2$ then $c^2 \twoheadrightarrow c'^2$.

$$\begin{aligned}
 \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket \mu_0\hat{\text{tp}}.c \rrbracket &= \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket c \rrbracket \lambda u.u k \gamma \\
 \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket [\hat{\text{tp}}]_0 t \rrbracket &= \bar{\lambda}\gamma.\gamma \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket t \rrbracket \\
 \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket [*] \rrbracket &= \bar{\lambda}x.\bar{\lambda}\gamma.x
 \end{aligned}$$

Fig. A 10. The administrative $\bar{\lambda}$ -abstractions of the $\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2$ transform.

Proof

By induction on reduction sequence $c^2 \mapsto c'^2$ as the reflexive, transitive closure of \mapsto . In the case that a value is given to a dynamic co-variable,

$$D^2[D[\mu\hat{\alpha}.E_{\hat{\alpha}}^2 \llbracket [\hat{\alpha}]V \rrbracket]] \mapsto D^2[D[V]]$$

the simulation is obtained by induction on $E_{\hat{\alpha}}^2$. The rest of the proof is the same as the proof of Theorem A.2.2 for $\lambda\mu$. \square

A.5 Soundness and operational completeness of the $\lambda\mu\hat{\text{tp}}_0$ reduction theory

The new administrative $\bar{\lambda}$ -abstractions introduced in the $\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2$ transform are those of the $\mathcal{C}_{\lambda\mu}$ transform from Figure A 1 along with the new ones shown in Figure A 10. The reduction rules of $\lambda\mu\hat{\text{tp}}_0$ are sound with respect to the transform $\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2$.

Lemma A.5.1 (One-step soundness)

If $M \rightarrow M'$ then $\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket M \rrbracket =_{\beta\eta} \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket M' \rrbracket$.

Proof

By cases on the reduction $M \rightarrow M'$ in the $\lambda\mu\hat{\text{tp}}_0$ -calculus. Recall that the transform $\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket t \rrbracket$ is always an administrative $\bar{\lambda}$ -abstraction of the form $\bar{\lambda}k.M$, for some CPS program M .

- $\mu_0\hat{\text{tp}}.[\hat{\text{tp}}]_0 t \rightarrow t$:

$$\begin{aligned}
 \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket \mu_0\hat{\text{tp}}.[\hat{\text{tp}}]_0 t \rrbracket &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\lambda u.u k \gamma) \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket t \rrbracket \\
 &\rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket t \rrbracket k \gamma \\
 &\Rightarrow_{\bar{\eta}} \lambda k.\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket t \rrbracket k \\
 &\Rightarrow \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket t \rrbracket
 \end{aligned}$$

- $\mu_0\hat{\text{tp}}.[*]V \rightarrow \mu_{-}.[*]V$:

$$\begin{aligned}
 \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket \mu_0\hat{\text{tp}}.[*]V \rrbracket &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket [*] \rrbracket \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket V \rrbracket^V (\lambda u.u k \gamma) \\
 &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket V \rrbracket^V \\
 &\Leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket [*] \rrbracket \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket V \rrbracket^V \gamma \\
 &\Leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\lambda_{-}.\mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket [*] \rrbracket \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket V \rrbracket^V) k \gamma \\
 &\Rightarrow_{\bar{\eta}} \mathcal{C}_{\lambda\mu\hat{\text{tp}}_0}^2 \llbracket \mu_{-}.[*]V \rrbracket
 \end{aligned}$$

The rest of the cases are the same as for the proof of Lemma A.3.1 for $\lambda\mu\hat{\text{tp}}$. \square

$$\begin{aligned}
c_{fa} \in \mathit{FinalAnswer} &::= D^2[[*]V] \\
c_{whnf} \in \mathit{WHNF} &::= c_{fa} \mid [\bullet][\widehat{\text{tp}}]_0 V \mid D^2[[\alpha]V] \mid D^2[D[x V]]
\end{aligned}$$

Fig. A 11. Final answers and weak head-normal forms in the $\lambda\mu\widehat{\text{tp}}_0$ -calculus.

$$\begin{aligned}
\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[q^2]E^2] &= \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[E^2]] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[q^2]] \\
\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[\square]]\gamma &= \gamma \\
\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[D[\mu_0\widehat{\text{tp}}.E^2]]]\gamma &= \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[E^2]]\lambda t.t \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[D]]\gamma
\end{aligned}$$

Fig. A 12. CPS transform of call-by-value evaluation contexts of $\lambda\mu\widehat{\text{tp}}_0$.

Theorem A.5.1 (Soundness)

If $M \rightarrow M'$ then $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[M]] =_{\beta\eta} \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[M']]$.

In $\lambda\mu\widehat{\text{tp}}_0$, the final answers are the same as from Figure A 5 for $\lambda\mu\widehat{\text{tp}}$, and the set of weak head normal forms instead includes meta-commands of the form $[\bullet][\widehat{\text{tp}}]_0 t$, as shown in Figure A 11. The transform of evaluation meta-contexts is correspondingly modified from Figure A 6 to match the change in the CPS transform for $\mu\widehat{\text{tp}}.c$, as given in Figure A 12.

Lemma A.5.2 (Meta-lifting)

$\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[D^2[c]]] \Rightarrow \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[c]] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[D^2]]$ and $\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[E^2[c]]]\gamma \Rightarrow [[c]] (\mathcal{C}_{\lambda\mu\widehat{\text{tp}}_0}^2[[E^2]]\gamma)$

Proof

By induction on D^2 and E^2 . The proof follows the same form as the proof of Lemma A.3.2 for $\lambda\mu\widehat{\text{tp}}$. \square

Lemma A.5.3 (One-step evaluation)

- If $\overline{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}_0}^2[[c]]$ is a value then c^2 is a final answer.
- If $\overline{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}_0}^2[[c]]$ is a whnf then c^2 is a whnf.
- If $\overline{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}_0}^2[[c]] \mapsto M'$ by a non-administrative reduction then there is a $\lambda\mu\widehat{\text{tp}}_0$ meta-command c'^2 such that $c^2 \mapsto c'^2$ and $M' \Rightarrow \overline{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}_0}^2[[c'^2]]$.

Proof

By cases on c^2 , using the unique decomposition property.

- $c^2 = [\bullet][\widehat{\text{tp}}]_0 t$:

$$\overline{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}_0}^2[[[\bullet][\widehat{\text{tp}}]_0 t]] = \gamma_0 \overline{\mathcal{C}}_{\lambda\mu\widehat{\text{tp}}_0}^2[[t]]$$

- $c^2 = D^2[D[\mu_0\hat{\text{tp}}.\hat{\text{tp}}]_{0t}]$:

$$\begin{aligned} \overline{\mathcal{C}}^2[D^2[D[\mu_0\hat{\text{tp}}.\hat{\text{tp}}]_{0t}]] &= (\lambda u.u \overline{\mathcal{C}}^2[D] \overline{\mathcal{C}}^2[D^2]) \overline{\mathcal{C}}^2_{\lambda\mu\hat{\text{tp}}_0}[t] \\ &\text{(by Lemmas A.1.1 and A.5.2)} \\ &\mapsto \overline{\mathcal{C}}^2[t] \overline{\mathcal{C}}^2[D] \overline{\mathcal{C}}^2[D^2] \\ &\Rightarrow \overline{\mathcal{C}}^2[D^2[D[t]]] \end{aligned}$$

$$D^2[D[\mu_0\hat{\text{tp}}.\hat{\text{tp}}]_{0t}] \mapsto D^2[D[t]]$$

The rest of the cases are the same as for the proof of Lemma A.3.3 for $\lambda\mu\hat{\text{tp}}$. \square

Lemma A.5.4

If $M \Rightarrow \overline{\mathcal{C}}^2_{\lambda\mu\hat{\text{tp}}_0}[[c^2]]$ and $M \mapsto M'$, then there is a $\lambda\mu\hat{\text{tp}}_0$ meta-command c'^2 such that $c^2 \mapsto c'^2$ and $M' \Rightarrow \overline{\mathcal{C}}^2_{\lambda\mu\hat{\text{tp}}_0}[[c'^2]]$. Furthermore, if M' is a whnf then c'^2 is a whnf, and if M' is a value then c'^2 is a final answer.

Proof

The same as the proof of Lemma A.2.4 for $\lambda\mu$. \square

Theorem A.5.2 (Evaluation)

If $\overline{\mathcal{C}}^2_{\lambda\mu\hat{\text{tp}}_0}[[c^2]] =_{\beta} V$ then there is a final answer c'^2 such that $c^2 \mapsto c'^2$ and $\overline{\mathcal{C}}^2_{\lambda\mu\hat{\text{tp}}_0}[[c'^2]] =_{\beta} V$.

Proof

The same as the proof of Lemma A.2.1 for $\lambda\mu$. \square

Theorem A.5.3

If $c^2 \mapsto c'^2$ then $c^2 \twoheadrightarrow c'^2$.

Proof

By induction on reduction sequence $c^2 \mapsto c'^2$ as the reflexive, transitive closure of \mapsto . The result is immediate for reduction of $\mu_0\hat{\text{tp}}.\hat{\text{tp}}]_{0t}$. The rest of the proof is the same as the proof of Theorem A.2.2 for $\lambda\mu$. \square

A.6 Soundness and operational completeness of the $\lambda\hat{\mu}_0$ reduction theory

Like with $\lambda\hat{\mu}$ the reduction rules and operational semantics are sound and complete with respect to the transform $\mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda\hat{\mu}_0}$. The administrative $\bar{\lambda}$ -abstractions for $\lambda\hat{\mu}_0$ are those of $\lambda\mu$ from Figure A 1 along with the ones shown in Figure A 13. For the purpose of classifying reductions, reducing the if-then-else expression is an administrative reduction.

Since $\lambda\hat{\mu}_0$ introduces a new notion of substitution (structural substitution for a meta-context variable Δ), we need to check that our substitution lemma for $\lambda\hat{\mu}_0$ still holds for the $\mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda\hat{\mu}_0}$ transform. The new syntax for extending the meta-context, $[\Delta]c$, adds another case, shown in Figure A 14, to the transform of meta-context given for $\lambda\hat{\mu}$ from Figure A 9.

Lemma A.6.1 (Meta-lifting)

For the transform $\mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda\hat{\mu}_0}$, $[[D^2[c]]] \Rightarrow [[c]] [[D^2]]$ and $[[E^2[c]]\gamma] \Rightarrow [[c]] ([[E^2]]\gamma)$

$$\begin{aligned}
\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [\hat{\alpha}]_0 \Delta.t \rrbracket &= \bar{\lambda} \gamma. \gamma(\hat{\alpha}) (\lambda \Delta. \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t \rrbracket) \\
\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket \mu_0 \hat{\alpha}.c \rrbracket &= \bar{\lambda} k. \bar{\lambda} \gamma. \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket c \rrbracket \gamma[\hat{\alpha} \mapsto k] \\
\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket * \rrbracket &= \bar{\lambda} x. \bar{\lambda} \gamma. x
\end{aligned}$$

$$\begin{aligned}
\gamma[\hat{\alpha} \mapsto k] &= \bar{\lambda} p. \mathbf{if} \, p \equiv \ulcorner \hat{\alpha} \urcorner \\
&\quad \mathbf{then} \, (\bar{\lambda} g. g (\bar{\lambda} \gamma'. \gamma') \, k \, \gamma) \\
&\quad \mathbf{else} \, (\bar{\lambda} g. \gamma \, p \, (\bar{\lambda} \Delta. g \, (\bar{\lambda} \gamma'. (\Delta \, \gamma') [\hat{\alpha} \mapsto k])))
\end{aligned}$$

Fig. A 13. The administrative $\bar{\lambda}$ -abstractions of the $\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0}$ transform.

$$\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [\Delta] E^2 \rrbracket \gamma = \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket E^2 \rrbracket (\Delta \, \gamma)$$

Fig. A 14. CPS transform of call-by-value evaluation contexts of the $\lambda \hat{\mu}_0$ -calculus.

Proof

Analogous to the proof of Lemma A.4.2 for $\lambda \hat{\mu}$. \square

This allows us to show the correspondence of structural substitution of the meta-context between the source and target languages.

Lemma A.6.2 (Substitution)

$$\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket M \{ E^2[c] / [\Delta]c \} \rrbracket \Rightarrow \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket M \{ ([E^2]\gamma) / (\Delta \, \gamma) \} \rrbracket$$

Proof

By induction on M . The only new interesting case is for structural substitution on Δ . For conciseness, the transform $\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket M \rrbracket$ is written as just $\llbracket M \rrbracket$.

$$\begin{aligned}
\llbracket ([\Delta]c) \{ E^2[c'] / [\Delta]c' \} \rrbracket \gamma &= \llbracket E^2[c \{ E^2[c'] / [\Delta]c' \}] \rrbracket \gamma \\
&\Rightarrow \llbracket c \{ E^2[c'] / [\Delta]c' \} \rrbracket ([E^2]\gamma) && \text{(by Lemma A.6.1)} \\
&\Rightarrow \llbracket c \rrbracket \{ ([E^2]\gamma) / (\Delta \, \gamma) \} ([E^2]\gamma) && \text{(by Lemma A.6.1 and I.H.)} \\
&= \llbracket c \rrbracket (\Delta \, \gamma) \{ ([E^2]\gamma) / (\Delta \, \gamma) \} \\
&= \llbracket [\Delta]c \rrbracket \gamma \{ ([E^2]\gamma) / (\Delta \, \gamma) \}
\end{aligned}$$

\square

Likewise, we have a stronger substitution lemma for the administrative-normal CPS transform $\overline{\mathcal{D}}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0}$.

Lemma A.6.3

$$\overline{\mathcal{D}}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket M \{ E^2[c] / [\Delta]c \} \rrbracket \Leftarrow \overline{\mathcal{D}}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\hat{\lambda}\hat{\mu}_0} \llbracket M \rrbracket \{ ([E^2]\gamma) / (\Delta \, \gamma) \}$$

Proof

Same as the proof of Lemma A.2.2 for $\lambda \mu$. \square

The proof of soundness of the $\lambda \hat{\mu}_0$ reduction rules follows from the extended substitution Lemma A.6.2.

$$c_{fa} \in \text{FinalAnswer} ::= D^2[[*]V]$$

$$c_{whnf} \in \text{WHNF} ::= c_{fa} \mid [\bullet]E_{\hat{\alpha}}^2[[\hat{\alpha}]V] \mid D^2[[\Delta']E_{\hat{\alpha}}^2[[\hat{\alpha}]V]] \mid D^2[[\alpha]V] \mid D^2[D[x V]]$$

Fig. A 15. Final answers and weak head-normal forms in the $\lambda\hat{\mu}_0$ -calculus.

Lemma A.6.4 (One-step soundness)

If $M \rightarrow M'$ then $\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\lambda\hat{\mu}_0} \llbracket M \rrbracket =_{\beta\eta} \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\lambda\hat{\mu}_0} \llbracket M' \rrbracket$.

Proof

By cases on the reduction $M \rightarrow M'$ in the $\lambda\hat{\mu}_0$ -calculus. Recall that the transform $\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\lambda\hat{\mu}_0} \llbracket t \rrbracket$ is always an administrative $\bar{\lambda}$ -abstraction of the form $\bar{\lambda}k.M$, for some CPS program M . For conciseness, in this proof the transform $\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\lambda\hat{\mu}_0} \llbracket M \rrbracket$ is written as just $\llbracket M \rrbracket$.

- $\mu_0\hat{\alpha}.\llbracket \hat{\alpha} \rrbracket_0 \Delta.t \rightarrow t\{c/[\Delta]c\}$:

$$\begin{aligned} \llbracket \mu_0\hat{\alpha}.\llbracket \hat{\alpha} \rrbracket_0 \Delta.t \rrbracket &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\gamma[\hat{\alpha} \mapsto k])(\hat{\alpha}) (\lambda\Delta.\llbracket t \rrbracket) \\ &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\lambda\Delta.\llbracket t \rrbracket) (\bar{\lambda}\gamma'.\gamma') k \gamma \\ &\rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\llbracket t \rrbracket \{(\bar{\lambda}\gamma'.\gamma')/\Delta\} k \gamma \\ &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\llbracket t \rrbracket \{\gamma'/(\Delta\gamma')\} k \gamma \\ &\Leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.\llbracket t\{c/[\Delta]c\} \rrbracket k \gamma && \text{(by Lemma A.6.2)} \\ &\Rightarrow_{\bar{\eta}} \bar{\lambda}k.\llbracket t\{c/[\Delta]c\} \rrbracket k \\ &\Rightarrow \llbracket t\{c/[\Delta]c\} \rrbracket \end{aligned}$$

- $\mu_0\hat{\beta}.\llbracket \hat{\alpha} \rrbracket_0 \Delta.t \rightarrow \mu\beta.\llbracket \hat{\alpha} \rrbracket_0 \Delta.t\{[\Delta](\llbracket \beta \rrbracket \mu_0\hat{\beta}.c)/[\Delta]c\}$:

$$\begin{aligned} \llbracket \mu_0\hat{\beta}.\llbracket \hat{\alpha} \rrbracket_0 \Delta.t \rrbracket &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\gamma[\hat{\beta} \mapsto k])(\hat{\alpha}) (\lambda\Delta.\llbracket t \rrbracket) \\ &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\gamma(\hat{\alpha}) (\bar{\lambda}\Delta'.(\lambda\Delta.\llbracket t \rrbracket) (\bar{\lambda}\gamma'.(\Delta'\gamma')[\hat{\beta} \mapsto k])) \\ &\rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\gamma(\hat{\alpha}) (\bar{\lambda}\Delta'.\llbracket t \rrbracket \{(\bar{\lambda}\gamma'.(\Delta'\gamma')[\hat{\beta} \mapsto k])/\Delta\}) \\ &\Rightarrow \bar{\lambda}k.\bar{\lambda}\gamma.\gamma(\hat{\alpha}) (\bar{\lambda}\Delta'.\llbracket t \rrbracket \{((\Delta'\gamma')[\hat{\beta} \mapsto k])/\Delta\}) \\ &\Leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\lambda\beta.\bar{\lambda}\gamma.\gamma(\hat{\alpha}) (\bar{\lambda}\Delta'.\llbracket t \rrbracket \{((\Delta'\gamma')[\hat{\beta} \mapsto \beta])/\Delta\})) k \gamma \\ &\Leftarrow \bar{\lambda}k.\bar{\lambda}\gamma.(\lambda\beta.\bar{\lambda}\gamma.\gamma(\hat{\alpha}) (\bar{\lambda}\Delta'.\llbracket t\{[\Delta']\llbracket \beta \rrbracket \mu_0\hat{\beta}.c/[\Delta]c\} \rrbracket})) k \gamma \\ & && \text{(by Lemma A.6.2)} \\ &\Rightarrow_{\bar{\eta}} \llbracket \mu\beta.\llbracket \hat{\alpha} \rrbracket_0 \Delta'.t\{[\Delta'](\llbracket \beta \rrbracket \mu_0\hat{\beta}.c)/[\Delta]c\} \rrbracket \end{aligned}$$

The rest of the cases are the same as for the proofs of Lemmas A.4.4 and A.5.3 for $\lambda\hat{\mu}$ and $\lambda\mu\hat{t}\hat{p}_0$. \square

Theorem A.6.1 (Soundness)

If $M \twoheadrightarrow M'$ then $\mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\lambda\hat{\mu}_0} \llbracket M \rrbracket =_{\beta\eta} \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{C}}_{\lambda\hat{\mu}_0} \llbracket M' \rrbracket$.

The final answers of $\lambda\hat{\mu}_0$ are the same as the final answers from Figure A 8 for $\lambda\hat{\mu}$. The weak head-normal forms of $\lambda\hat{\mu}_0$ still include $D^2[D[x V]]$ and $D^2[[\alpha]V]$, as well as $D^2[E_{\hat{\alpha}}^2[[\hat{\alpha}]_0 \Delta.t]]$ with $D^2 = [\bullet]\square$ or $D^2 = D^2[[\Delta']\square]$ and where $E_{\hat{\alpha}}^2$ does not bind $\hat{\alpha}$. Looking

up a dynamic co-variable still behaves as expected, except that now the prefix of the environment that is skipped over is collected and returned to the calling continuation.

Lemma A.6.5 (Lookup)

For the transform $\mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda \hat{\mu}_0}, \llbracket D^2[E_{\hat{\alpha}}^2] \rrbracket(\hat{\alpha}) \Rightarrow \bar{\lambda}g. \llbracket D^2 \rrbracket(\hat{\alpha}) (\bar{\lambda}\Delta.g \bar{\lambda}\gamma. \llbracket [\Delta]E_{\hat{\alpha}}^2 \rrbracket \gamma)$ when $E_{\hat{\alpha}}^2 \neq \square$.

Proof

By induction on $E_{\hat{\alpha}}^2$. For conciseness, the transform $\mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda \hat{\mu}_0} \llbracket M \rrbracket$ is written as just $\llbracket M \rrbracket$.

- $E_{\hat{\alpha}}^2 = D[\mu_0 \hat{\beta}. \square]$:

$$\begin{aligned} \llbracket D^2[D[\mu_0 \hat{\beta}. \square]] \rrbracket(\hat{\alpha}) &= \llbracket D^2 \rrbracket[\hat{\beta} \mapsto \llbracket D \rrbracket](\hat{\alpha}) \\ &\Rightarrow \bar{\lambda}g. \llbracket D^2 \rrbracket(\hat{\alpha}) \bar{\lambda}\Delta.g \bar{\lambda}\gamma. (\Delta \gamma) [\hat{\beta} \mapsto \llbracket D \rrbracket] \\ &= \bar{\lambda}g. \llbracket D^2 \rrbracket(\hat{\alpha}) \bar{\lambda}\Delta.g \bar{\lambda}\gamma. \llbracket [\Delta]D[\mu_0 \hat{\beta}. \square] \rrbracket \gamma \end{aligned}$$

- $E_{\hat{\alpha}}^2 = D[\mu_0 \hat{\beta}. E_{\hat{\alpha}}'^2]$, where $E_{\hat{\alpha}}'^2 \neq \square$:

$$\begin{aligned} \llbracket D^2[D[\mu_0 \hat{\beta}. E_{\hat{\alpha}}'^2]] \rrbracket(\hat{\alpha}) &\Rightarrow \bar{\lambda}g. \llbracket D^2[D[\mu_0 \hat{\beta}. \square]] \rrbracket(\hat{\alpha}) (\bar{\lambda}\Delta.g \bar{\lambda}\gamma. \llbracket [\Delta]E_{\hat{\alpha}}'^2 \rrbracket \gamma) \quad (\text{by I.H.}) \\ &= \bar{\lambda}g. \llbracket D^2 \rrbracket[\hat{\beta} \mapsto \llbracket D \rrbracket](\hat{\alpha}) (\bar{\lambda}\Delta.g \bar{\lambda}\gamma. \llbracket E_{\hat{\alpha}}'^2 \rrbracket (\Delta \gamma)) \\ &\Rightarrow \bar{\lambda}g. \llbracket D^2 \rrbracket(\hat{\alpha}) (\bar{\lambda}\Delta.g \bar{\lambda}\gamma. \llbracket E_{\hat{\alpha}}'^2 \rrbracket (\Delta \gamma) [\hat{\beta} \mapsto \llbracket D \rrbracket]) \\ &= \bar{\lambda}g. \llbracket D^2 \rrbracket(\hat{\alpha}) (\bar{\lambda}\Delta.g \bar{\lambda}\gamma. \llbracket [\Delta]D[\mu_0 \hat{\beta}. E_{\hat{\alpha}}'^2] \rrbracket \gamma) \end{aligned}$$

□

Finally, completeness with respect to evaluation in the CPS transform follows as expected.

Lemma A.6.6 (One-step evaluation)

If $\mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda \hat{\mu}_0} \llbracket c^2 \rrbracket \Rightarrow M$ then

- If $\mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda \hat{\mu}_0} \llbracket c^2 \rrbracket$ is a value then c^2 is a final answer.
- If $\mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda \hat{\mu}_0} \llbracket c^2 \rrbracket$ is a whnf then c^2 is a whnf.
- If $\mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda \hat{\mu}_0} \llbracket c^2 \rrbracket \mapsto M'$ by a non-administrative reduction then there is a $\lambda \hat{\mu}_0$ meta-command c'^2 such that $c^2 \mapsto c'^2$ and $M' \Rightarrow \mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda \hat{\mu}_0} \llbracket c'^2 \rrbracket$.

Proof

By cases on c^2 using the unique decomposition property. For conciseness, in this proof the transform $\mathcal{D}_{\lambda_0} \widehat{\mathcal{C}}_{\lambda \hat{\mu}_0} \llbracket M \rrbracket$ is written as just $\llbracket M \rrbracket$.

- $c^2 = [\bullet]E_{\hat{\alpha}}^2 \llbracket [\hat{\alpha}]_0 \Delta.t \rrbracket$:

$$\llbracket [\bullet]E_{\hat{\alpha}}^2 \llbracket [\hat{\alpha}]_0 \Delta.t \rrbracket \rrbracket = \gamma_0(\hat{\alpha}) g \quad (\text{by Lemmas A.6.1 and A.6.5})$$

$$\text{where } g = \begin{cases} \lambda \Delta. \llbracket t \rrbracket & \text{if } E_{\hat{\alpha}}^2 = \square \\ \bar{\lambda} \Delta'. (\lambda \Delta. \llbracket t \rrbracket) \llbracket [\Delta']E_{\hat{\alpha}}^2 \rrbracket & \text{otherwise} \end{cases}$$

- $c^2 = D^2[[\Delta]E_{\hat{\alpha}}^2[[\hat{\alpha}]_0\Delta.t]]$:

$$[[D^2[[\Delta]E_{\hat{\alpha}}^2[[\hat{\alpha}]_0\Delta.t]]] = (\Delta [[D^2]]) (\hat{\alpha}) g \quad (\text{by Lemmas A.6.1 and A.6.5})$$

$$\text{where } g = \begin{cases} \lambda\Delta.[t] & \text{if } E_{\hat{\alpha}}^2 = \square \\ \bar{\lambda}\Delta'.(\lambda\Delta.[t]) [[\Delta']E_{\hat{\alpha}}^2] & \text{otherwise} \end{cases}$$

- $c^2 = D^2[D[\mu_0\hat{\alpha}.E_{\hat{\alpha}}^2[[\hat{\alpha}]_0\Delta.t]]$, when $E_{\hat{\alpha}}^2 \neq \square$:

$$\begin{aligned} & [[D^2[D[\mu_0\hat{\alpha}.E_{\hat{\alpha}}^2[[\hat{\alpha}]_0\Delta.t]]]] \\ &= (D^2[\hat{\alpha} \mapsto [D]]) (\hat{\alpha}) (\bar{\lambda}\Delta'.(\lambda\Delta.[t]) [[\Delta']E_{\hat{\alpha}}^2]) \quad (\text{by Lemmas A.1.1, A.6.1 and A.6.5}) \\ &= (\lambda\Delta.[t]) [[E_{\hat{\alpha}}^2]] [D] [D^2] \\ &\mapsto [t] \{ [E_{\hat{\alpha}}^2] / \Delta \} [D] [D^2] \\ &\Rightarrow [t] \{ ([E_{\hat{\alpha}}^2]\gamma) / (\Delta \gamma) \} [D] [D^2] \\ &\Rightarrow [t] \{ E_{\hat{\alpha}}^2[c] / [\Delta]c \} [D] [D^2] \quad (\text{by Lemma A.6.3}) \\ &= [[D^2[D[t\{E_{\hat{\alpha}}^2[c] / [\Delta]c\}]]] \quad (\text{by Lemmas A.1.1 and A.6.1}) \end{aligned}$$

$$D^2[D[\mu_0\hat{\alpha}.E_{\hat{\alpha}}^2[[\hat{\alpha}]_0\Delta.t]] \mapsto D^2[D[t\{E_{\hat{\alpha}}^2[c] / [\Delta]c\}]]$$

And similarly when $E_{\hat{\alpha}}^2 = \square$.

The rest of the cases are the same as for the proofs of Lemmas A.4.4 and A.5.3 for $\lambda\hat{\mu}$ and $\lambda\mu\hat{\text{tp}}_0$. \square

Lemma A.6.7

If $M \Rightarrow \overline{\mathcal{D}}_{\lambda_0} \widehat{\mathcal{E}}_{\lambda\hat{\mu}_0} \llbracket c^2 \rrbracket$ and $M \mapsto M'$, then there is a $\lambda\hat{\mu}_0$ meta-command c'^2 such that $c^2 \mapsto c'^2$ and $M' \Rightarrow \overline{\mathcal{D}}_{\lambda_0} \widehat{\mathcal{E}}_{\lambda\hat{\mu}_0} \llbracket c'^2 \rrbracket$. Furthermore, if M' is a whnf then c'^2 is a whnf, and if M' is a value then c'^2 is a final answer.

Proof

The same as the proof of Lemma A.2.4 for $\lambda\mu$. \square

Theorem A.6.2 (Evaluation)

If $\overline{\mathcal{D}}_{\lambda_0} \widehat{\mathcal{E}}_{\lambda\hat{\mu}_0} \llbracket c^2 \rrbracket =_{\beta} V$ then there is a final answer c'^2 such that $c^2 \mapsto c'^2$ and $\overline{\mathcal{D}}_{\lambda_0} \widehat{\mathcal{E}}_{\lambda\hat{\mu}_0} \llbracket c'^2 \rrbracket =_{\beta} V$.

Proof

The same as the proof of Lemma A.2.1 for $\lambda\mu$. \square

Theorem A.6.3

If $c^2 \mapsto c'^2$ then $c^2 \twoheadrightarrow c'^2$.

Proof

By induction on reduction sequence $c^2 \mapsto c'^2$ as the reflexive, transitive closure of \mapsto . In the case that a prefix of the meta-context is captured by a command of the form $[\hat{\alpha}]_0\Delta.t$,

$$D^2[D[\mu_0\hat{\alpha}.E_{\hat{\alpha}}^2[[\hat{\alpha}]_0\Delta.t]] \mapsto D^2[D[t\{E_{\hat{\alpha}}^2[c] / [\Delta]c\}]]$$

the simulation is obtained by induction on the context $E_{\hat{\alpha}}^2$, where reduction is lifted into the context $D^2[D]$.

- $E_{\hat{\alpha}}^2 = \square$:

$$\mu_0 \hat{\alpha}. [\hat{\alpha}]_0 \Delta. t \rightarrow t \{c / [\Delta] c\}$$

- $E_{\hat{\alpha}}^2 = E_{\hat{\alpha}}'^2 [D'[\mu \hat{\beta}. \square]]$:

$$\begin{aligned} \mu_0 \hat{\alpha}. E_{\hat{\alpha}}'^2 [D'[\mu \hat{\beta}. [\hat{\alpha}]_0 \Delta. t]] &\rightarrow \mu_0 \hat{\alpha}. E_{\hat{\alpha}}'^2 [D'[\mu \gamma. [\hat{\alpha}]_0 \Delta. t \{[\Delta][\gamma] \mu \hat{\beta}. c / [\Delta] c\}]] \\ &\twoheadrightarrow \mu_0 \hat{\alpha}. E_{\hat{\alpha}}'^2 [[\hat{\alpha}]_0 \Delta. t \{[\Delta] D'[\mu \hat{\beta}. c] / [\Delta] c\}] && \text{(by Theorem A.2.2)} \\ &\twoheadrightarrow t \{E_{\hat{\alpha}}'^2 [D'[\mu \hat{\beta}. c] / [\Delta] c\} && \text{(by I.H.)} \end{aligned}$$

The rest of the proof is the same as the proof of Theorem A.2.2 for $\lambda \mu$. \square

B List of definitions

B.1 Languages of Control

In the following, we summarize the syntax and semantics of the various languages of control by listing their reduction theories and CPS transforms in their entirety.

Parigot's $\lambda \mu$ -calculus:

$$\begin{aligned} t \in \text{Term} &::= V \mid t_1 t_2 \mid \mu \alpha. c & c \in \text{Command} &::= [q] t \\ V \in \text{Value} &::= x \mid \lambda x. t & q \in \text{CoTerm} &::= \alpha \mid * \end{aligned}$$

Fig. B 1. The syntax of the $\lambda \mu$ -calculus.

$$F \in \text{Frame} ::= \square t \mid V \square$$

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

Fig. B 2. Call-by-value reduction theory of the $\lambda \mu$ -calculus.

$$\begin{aligned} \mathcal{C}_{\lambda \mu} \llbracket [q] t \rrbracket &= \mathcal{C}_{\lambda \mu} \llbracket t \rrbracket \mathcal{C}_{\lambda \mu} \llbracket q \rrbracket & \mathcal{C}_{\lambda \mu} \llbracket * \rrbracket &= \lambda x. x \\ \mathcal{C}_{\lambda \mu} \llbracket V \rrbracket &= \lambda k. k \mathcal{C}_{\lambda \mu} \llbracket V \rrbracket^V & \mathcal{C}_{\lambda \mu} \llbracket \alpha \rrbracket &= \alpha \\ \mathcal{C}_{\lambda \mu} \llbracket t_1 t_2 \rrbracket &= \lambda k. \mathcal{C}_{\lambda \mu} \llbracket t_1 \rrbracket \lambda f. \mathcal{C}_{\lambda \mu} \llbracket t_2 \rrbracket \lambda s. f s k & \mathcal{C}_{\lambda \mu} \llbracket x \rrbracket^V &= x \\ \mathcal{C}_{\lambda \mu} \llbracket \mu \alpha. c \rrbracket &= \lambda k. (\lambda \alpha. \mathcal{C}_{\lambda \mu} \llbracket c \rrbracket) k & \mathcal{C}_{\lambda \mu} \llbracket \lambda x. t \rrbracket^V &= \lambda x. \mathcal{C}_{\lambda \mu} \llbracket t \rrbracket \end{aligned}$$

Fig. B 3. Call-by-value CPS transform for the $\lambda \mu$ -calculus.

$\lambda \mu \hat{\text{tp}}$ -calculus:

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

Fig. B 4. The syntax of the $\lambda\mu\widehat{\text{tp}}$ -calculus.

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

Fig. B 5. Call-by-value reduction theory of the $\lambda\mu\widehat{\text{tp}}$ -calculus.

$$\begin{aligned}
 \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[q^2]c] &= \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[c]] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[q^2]] & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\bullet]] &= \gamma_0 \quad \text{where } \gamma_0 \text{ free} \\
 \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[q]t] &= \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[t]] \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[q]] & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\otimes]] &= \lambda x.x \\
 \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[V]] &= \lambda k.k \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[V]]^V & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\alpha]] &= \alpha \\
 \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[t_1 t_2]] &= \lambda k.\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[t_1]]\lambda f.\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[t_2]]\lambda s.f s k & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\widehat{\text{tp}}]] &= \lambda x.\lambda\gamma.\gamma x \\
 \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\mu\alpha.c]] &= \lambda k.(\lambda\alpha.\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[c]]) k & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[*]] &= \lambda x.\lambda\gamma.x \\
 \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\mu\widehat{\text{tp}}.c]] &= \lambda k.\lambda\gamma.\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[c]]\lambda x.k x \gamma & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[x]]^V &= x \\
 & & \mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[\lambda x.t]]^V &= \lambda x.\mathcal{C}_{\lambda\mu\widehat{\text{tp}}}^2[[t]]
 \end{aligned}$$

Fig. B 6. Call-by-value double CPS transform of the $\lambda\mu\widehat{\text{tp}}$ -calculus.

$\lambda\widehat{\mu}$ -calculus:

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

Fig. B 7. The syntax of the $\lambda\widehat{\mu}$ -calculus.

$$\begin{aligned}
 (\lambda x.t) V &\rightarrow t\{V/x\} & \mu\widehat{\alpha}.[\widehat{\alpha}]V &\rightarrow V \\
 F[\mu\alpha.c] &\rightarrow \mu\alpha.c\{[\alpha](F[t])/[\alpha]t\} & \mu\widehat{\alpha}.[\widehat{\beta}]V &\rightarrow \mu_.[\widehat{\beta}]V \quad \text{where } \widehat{\alpha} \neq \widehat{\beta} \\
 [q]\mu\alpha.c &\rightarrow c\{q/\alpha\} & \mu\widehat{\alpha}.[*]V &\rightarrow \mu_.[*]V
 \end{aligned}$$

Fig. B 8. Call-by-value reduction theory for dynamic co-variables in the $\lambda\widehat{\mu}$ -calculus.

$$\begin{aligned}
\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[[q^2]c] &= \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c] \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[[q^2]] & \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[\bullet] &= \gamma_0 \quad \text{where } \gamma_0 \text{ free} \\
\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[[q]t] &= \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[t] \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[[q]] & \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[\alpha] &= \alpha \\
\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[V] &= \lambda k.k \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[V]^V & \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[\hat{\alpha}] &= \lambda x.\lambda \gamma.\gamma(\hat{\alpha}) x \\
\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[t_1 t_2] &= \lambda k.\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[t_1] \lambda f.\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[t_2] \lambda s.f s k & \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[*] &= \lambda x.\lambda \gamma.x \\
\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[\mu \alpha.c] &= \lambda k.(\lambda \alpha.\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c]) k & \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[x]^V &= x \\
\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[\mu \hat{\alpha}.c] &= \lambda k.\lambda \gamma.\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[c](\gamma[\hat{\alpha} \mapsto k]) & \mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[\lambda x.t]^V &= \lambda x.\mathcal{D}_\lambda \widehat{\mathcal{C}}_{\lambda\hat{\mu}}[t] \\
\end{aligned}$$

$$\begin{aligned}
\gamma(\hat{\alpha}) &= \gamma \ulcorner \hat{\alpha} \urcorner \\
\gamma[\hat{\alpha} \mapsto k] &= \lambda p.\text{if } p \equiv \ulcorner \hat{\alpha} \urcorner \text{ then } (\lambda x.k x \gamma) \text{ else } \gamma p
\end{aligned}$$

Fig. B 9. Call-by-value composed CPS transform of the $\lambda\hat{\mu}$ -calculus. **$\lambda\mu\hat{\text{tr}}_0$ -calculus:**

$$\begin{aligned}
c^2 &\in \text{Command}^2 ::= [q^2]c \\
c &\in \text{Command} ::= [q]t \mid [\hat{\text{tr}}]_0 t \\
t \in \text{Term} &::= V \mid t_1 t_2 \mid \mu \alpha.c \mid \mu_0 \hat{\text{tr}}.c & q^2 \in \text{CoTerm}^2 &::= \bullet \\
V \in \text{Value} &::= x \mid \lambda x.t & q \in \text{CoTerm} &::= \alpha \mid *
\end{aligned}$$

Fig. B 10. The syntax of the $\lambda\mu\hat{\text{tr}}_0$ -calculus.

$$\begin{aligned}
(\lambda x.t) V &\rightarrow t\{V/x\} & \mu_0 \hat{\text{tr}}.[\hat{\text{tr}}]_0 t &\rightarrow t \\
F[\mu \alpha.c] &\rightarrow \mu \alpha.c\{[\alpha](F[t])/[\alpha]t\} & \mu_0 \hat{\text{tr}}.[*]V &\rightarrow \mu.[*]V \\
[q]\mu \alpha.c &\rightarrow c\{q/\alpha\}
\end{aligned}$$

Fig. B 11. Call-by-value reduction theory of the $\lambda\mu\hat{\text{tr}}_0$ -calculus.

$$\begin{aligned}
\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[[q^2]c] &= \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[c] \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[[q^2]] & \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[\bullet] &= \gamma_0 \quad \text{where } \gamma_0 \text{ free} \\
\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[[q]t] &= \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[t] \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[[q]] & \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[\alpha] &= \alpha \\
\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[[\hat{\text{tr}}]_0 t] &= \lambda \gamma.\gamma \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[t] & \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[*] &= \lambda x.\lambda \gamma.x \\
\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[V] &= \lambda k.k \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[V]^V & \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[x]^V &= x \\
\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[t_1 t_2] &= \lambda k.\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[t_1] \lambda f.\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[t_2] \lambda s.f s k & \mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[\lambda x.t]^V &= \lambda x.\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[t] \\
\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[\mu \alpha.c] &= \lambda k.(\lambda \alpha.\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[c]) k \\
\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[\mu_0 \hat{\text{tr}}.c] &= \lambda k.\lambda \gamma.\mathcal{C}_{\lambda\mu\hat{\text{tr}}_0}^2[c] \lambda u.u k \gamma
\end{aligned}$$

Fig. B 12. Call-by-value double CPS transform of the $\lambda\mu\hat{\text{tr}}_0$ -calculus.

$\lambda\hat{\mu}_0$ -calculus:

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

Fig. B 13. The syntax of the $\lambda\hat{\mu}_0$ -calculus.

$$\begin{aligned}
 (\lambda x.t) V &\rightarrow t\{V/x\} & \mu_0\hat{\alpha}.\widehat{[\hat{\alpha}]}_0\Delta.t &\rightarrow t\{c/[\Delta]c\} \\
 F[\mu\alpha.c] &\rightarrow \mu\alpha.c\{[\alpha](F[t])/[\alpha]t\} & \mu_0\hat{\alpha}.\widehat{[\hat{\beta}]}_0\Delta.t &\rightarrow \mu\alpha.\widehat{[\hat{\beta}]}_0\Delta.t\{[\Delta][\alpha](\mu_0\hat{\alpha}.c)/[\Delta]c\} \\
 [q]\mu\alpha.c &\rightarrow c\{q/\alpha\} & & \text{where } \hat{\alpha} \neq \hat{\beta} \\
 & & \mu_0\hat{\alpha}.\widehat{[*]}V &\rightarrow \mu_.\widehat{[*]}V
 \end{aligned}$$

Fig. B 14. Call-by-value reduction theory of the $\lambda\hat{\mu}_0$ -calculus.

$$\begin{aligned}
 \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket x \rrbracket^V &= x & \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [\alpha] \rrbracket &= \alpha & \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [\bullet] \rrbracket &= \gamma_0 \\
 \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket \lambda x.t \rrbracket^V &= \lambda x. \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t \rrbracket & \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [*] \rrbracket &= \lambda x. \lambda \gamma. x & & \text{where } \gamma_0 \text{ free}
 \end{aligned}$$

$$\begin{aligned}
 \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket V \rrbracket &= \lambda k. k \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket V \rrbracket^V \\
 \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t_1 t_2 \rrbracket &= \lambda k. \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t_1 \rrbracket \lambda f. \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t_2 \rrbracket \lambda s. f s k \\
 \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket \mu\alpha.c \rrbracket &= \lambda k. (\lambda \alpha. \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket c \rrbracket) k \\
 \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket \mu_0\hat{\alpha}.c \rrbracket &= \lambda k. \lambda \gamma. \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket c \rrbracket (\gamma[\hat{\alpha} \mapsto k])
 \end{aligned}$$

$$\begin{aligned}
 \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [q^2]c \rrbracket &= \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket c \rrbracket \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [q^2] \rrbracket \\
 \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [q]t \rrbracket &= \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t \rrbracket \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [q] \rrbracket \\
 \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [\hat{\alpha}]_0\Delta.t \rrbracket &= \lambda \gamma. \gamma(\hat{\alpha}) (\lambda \Delta. \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket t \rrbracket) \\
 \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket [\Delta]c \rrbracket &= \lambda \gamma. \mathcal{D}_{\hat{\lambda}_0} \widehat{\mathcal{E}}_{\hat{\lambda}\hat{\mu}_0} \llbracket c \rrbracket (\Delta \gamma)
 \end{aligned}$$

$$\begin{aligned}
 \gamma(\hat{\alpha}) &= \gamma \ulcorner \hat{\alpha} \urcorner \\
 \gamma[\hat{\alpha} \mapsto k] &= \lambda p. \mathbf{if} p \equiv \ulcorner \hat{\alpha} \urcorner \\
 & \quad \mathbf{then} (\lambda g. g (\lambda \gamma'. \gamma') k \gamma) \\
 & \quad \mathbf{else} (\lambda g. \gamma p (\lambda \Delta. g (\lambda \gamma'. (\Delta \gamma')[\hat{\alpha} \mapsto k])))
 \end{aligned}$$

Fig. B 15. Call-by-value composed CPS transform of the $\hat{\lambda}_0$ -calculus.

B.2 Languages of Dynamic Binding

In the following, we summarize the syntax and semantics, in terms of environment-passing style transforms, of the intermediate languages of dynamic binding.

$\lambda \widehat{\text{tp}}$ -calculus:

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

Fig. B 16. The syntax of the $\lambda \widehat{\text{tp}}$ -calculus.

$$\begin{aligned} \mathcal{D}_{\lambda \widehat{\text{tp}}}[[e]t] &= \mathcal{D}_{\lambda \widehat{\text{tp}}}[[t]] \mathcal{D}_{\lambda \widehat{\text{tp}}}[[e]] & \mathcal{D}_{\lambda \widehat{\text{tp}}}[\bullet] &= \gamma_0 \quad \text{where } \gamma_0 \text{ free} \\ \mathcal{D}_{\lambda \widehat{\text{tp}}}[[V]] &= \lambda \gamma. \mathcal{D}_{\lambda \widehat{\text{tp}}}[[V]]^V & \mathcal{D}_{\lambda \widehat{\text{tp}}}[[x]]^V &= x \\ \mathcal{D}_{\lambda \widehat{\text{tp}}}[[t_1 t_2]] &= \lambda \gamma. (\mathcal{D}_{\lambda \widehat{\text{tp}}}[[t_1]] \gamma) (\mathcal{D}_{\lambda \widehat{\text{tp}}}[[t_2]] \gamma) \gamma & \mathcal{D}_{\lambda \widehat{\text{tp}}}[[\lambda x.t]]^V &= \lambda x. \mathcal{D}_{\lambda \widehat{\text{tp}}}[[t]] \\ \mathcal{D}_{\lambda \widehat{\text{tp}}}[[\widehat{\text{tp}} t]] &= \lambda \gamma. \gamma(\widehat{\text{tp}}) (\mathcal{D}_{\lambda \widehat{\text{tp}}}[[t]] \gamma) & \mathcal{D}_{\lambda \widehat{\text{tp}}}[[\lambda \widehat{\text{tp}}.t]]^V &= \lambda v. \lambda \gamma. \mathcal{D}_{\lambda \widehat{\text{tp}}}[[t]] (\gamma[\widehat{\text{tp}} \mapsto v]) \\ \gamma(\widehat{\text{tp}}) &= \gamma & \gamma[\widehat{\text{tp}} \mapsto v] &= \lambda x. v x \gamma \end{aligned}$$

Fig. B 17. Environment-passing style transform of the $\lambda \widehat{\text{tp}}$ -calculus.

$\widehat{\lambda}$ -calculus:

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

Fig. B 18. The syntax of the $\widehat{\lambda}$ -calculus.

$$\begin{aligned} \mathcal{D}_{\widehat{\lambda}}[[e]t] &= \mathcal{D}_{\widehat{\lambda}}[[t]] \mathcal{D}_{\widehat{\lambda}}[[e]] & \mathcal{D}_{\widehat{\lambda}}[\bullet] &= \gamma_0 \quad \text{where } \gamma_0 \text{ free} \\ \mathcal{D}_{\widehat{\lambda}}[[V]] &= \lambda \gamma. \mathcal{D}_{\widehat{\lambda}}[[V]]^V & \mathcal{D}_{\widehat{\lambda}}[[x]]^V &= x \\ \mathcal{D}_{\widehat{\lambda}}[[t_1 t_2]] &= \lambda \gamma. (\mathcal{D}_{\widehat{\lambda}}[[t_1]] \gamma) (\mathcal{D}_{\widehat{\lambda}}[[t_2]] \gamma) \gamma & \mathcal{D}_{\widehat{\lambda}}[[\lambda x.t]]^V &= \lambda x. \mathcal{D}_{\widehat{\lambda}}[[t]] \\ \mathcal{D}_{\widehat{\lambda}}[[\widehat{\text{tp}} t]] &= \lambda \gamma. \gamma(\widehat{\text{tp}}) (\mathcal{D}_{\widehat{\lambda}}[[t]] \gamma) & \mathcal{D}_{\widehat{\lambda}}[[\lambda \widehat{\text{tp}}.t]]^V &= \lambda v. \lambda \gamma. \mathcal{D}_{\widehat{\lambda}}[[t]] (\gamma[\widehat{\text{tp}} \mapsto v]) \\ \gamma(\widehat{x}) &= \gamma \ulcorner \widehat{x} \urcorner & \gamma[\widehat{x} \mapsto v] &= \lambda p. \text{if } p \equiv \ulcorner \widehat{x} \urcorner \text{ then } (\lambda x. v x \gamma) \text{ else } \gamma p \end{aligned}$$

Fig. B 19. Environment-passing style transform of the $\widehat{\lambda}$ -calculus.

$\lambda \widehat{\text{tp}}_0$ -calculus:

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

Fig. B 20. The syntax of the $\lambda\hat{\text{tp}}_0$ -calculus.

$$\begin{aligned}
 \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [e]t \rrbracket &= \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket t \rrbracket \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket e \rrbracket & \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket \bullet \rrbracket &= \gamma_0 \quad \text{where } \gamma_0 \text{ free} \\
 \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [V] \rrbracket &= \lambda \gamma. \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [V] \rrbracket^V & \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [x] \rrbracket^V &= x \\
 \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [t_1 t_2] \rrbracket &= \lambda \gamma. (\mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [t_1] \rrbracket \gamma) (\mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [t_2] \rrbracket \gamma) \gamma & \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [\lambda x.t] \rrbracket^V &= \lambda x. \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [t] \rrbracket \\
 \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [t \hat{\text{tp}}] \rrbracket &= \lambda \gamma. \gamma(\hat{\text{tp}}) (\mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [t] \rrbracket \gamma) & \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [\lambda \hat{\text{tp}}.t] \rrbracket^V &= \lambda v. \lambda \gamma. \mathcal{D}_{\lambda\hat{\text{tp}}_0} \llbracket [t] \rrbracket (\gamma[\hat{\text{tp}} \mapsto v]) \\
 \gamma(\hat{\text{tp}}) &= \gamma & \gamma[\hat{\text{tp}} \mapsto v] &= \lambda g. g \ v \ \gamma
 \end{aligned}$$

Fig. B 21. Environment-passing style transform of the $\lambda\hat{\text{tp}}$ -calculus.

$\hat{\lambda}_0$ -calculus:

$$\begin{aligned}
 c \in \text{Closure} &::= [e]t \\
 t \in \text{Term} &::= V \mid t_1 t_2 \mid t \hat{x} \mid [\Delta]t & e \in \text{Environment} &::= \bullet \\
 V \in \text{Value} &::= x \mid \lambda \tilde{x}.t \mid \lambda \langle \Delta, x \rangle. t & \tilde{x} \in \text{Var} &::= x \mid \hat{x}
 \end{aligned}$$

Fig. B 22. The syntax of the $\hat{\lambda}_0$ -calculus.

$$\begin{aligned}
 \mathcal{D}_{\hat{\lambda}_0} \llbracket [e]t \rrbracket &= \mathcal{D}_{\hat{\lambda}_0} \llbracket [t] \rrbracket \mathcal{D}_{\hat{\lambda}_0} \llbracket [e] \rrbracket & \mathcal{D}_{\hat{\lambda}_0} \llbracket \bullet \rrbracket &= \gamma_0 \quad \text{where } \gamma_0 \text{ free} \\
 \mathcal{D}_{\hat{\lambda}_0} \llbracket [V] \rrbracket &= \lambda \gamma. \mathcal{D}_{\hat{\lambda}_0} \llbracket [V] \rrbracket^V & \mathcal{D}_{\hat{\lambda}_0} \llbracket [x] \rrbracket^V &= x \\
 \mathcal{D}_{\hat{\lambda}_0} \llbracket [t_1 t_2] \rrbracket &= \lambda \gamma. (\mathcal{D}_{\hat{\lambda}_0} \llbracket [t_1] \rrbracket \gamma) (\mathcal{D}_{\hat{\lambda}_0} \llbracket [t_2] \rrbracket \gamma) \gamma & \mathcal{D}_{\hat{\lambda}_0} \llbracket [\lambda x.t] \rrbracket^V &= \lambda x. \mathcal{D}_{\hat{\lambda}_0} \llbracket [t] \rrbracket \\
 \mathcal{D}_{\hat{\lambda}_0} \llbracket [t \hat{x}] \rrbracket &= \lambda \gamma. \gamma(\hat{x}) (\mathcal{D}_{\hat{\lambda}_0} \llbracket [t] \rrbracket \gamma) & \mathcal{D}_{\hat{\lambda}_0} \llbracket [\lambda \hat{x}.t] \rrbracket^V &= \lambda v. \lambda \gamma. \mathcal{D}_{\hat{\lambda}_0} \llbracket [t] \rrbracket (\gamma[\hat{x} \mapsto v]) \\
 \mathcal{D}_{\hat{\lambda}_0} \llbracket [[\Delta]t] \rrbracket &= \lambda \gamma. \mathcal{D}_{\hat{\lambda}_0} \llbracket [t] \rrbracket (\Delta \ \gamma) \\
 \gamma(\hat{x}) &= \gamma \uparrow \hat{x}^{-1} \\
 \gamma[\hat{x} \mapsto v] &= \lambda p. \text{if } p \equiv \uparrow \hat{x}^{-1} \\
 & \quad \text{then } (\lambda g. g \ (\lambda \gamma'. \gamma') \ v \ \gamma) \\
 & \quad \text{else } (\lambda g. \gamma \ p \ (\lambda \Delta. g \ (\lambda \gamma'. (\Delta \ \gamma')[\hat{x} \mapsto v])))
 \end{aligned}$$

Fig. B 23. Environment-passing style transform of the $\hat{\lambda}_0$ -calculus.