# Compositional Semantics for Composable Continuations

## From Abortive to Delimited Control

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

## Abstract

Parigot's $\lambda\mu$-calculus, a system for computational reasoning about classical proofs, serves as a foundation for control operations embodied by operators like Scheme's callcc. We demonstrate that the call-by-value theory of the $\lambda\mu$-calculus contains a latent theory of delimited control, and that a known variant of $\lambda\mu$ which unshackles the syntax yields a calculus of composable continuations from the existing constructs and rules for classical control. To relate to the various formulations of control effects, and to continuation-passing style, we use a form of compositional program transformations which preserves the underlying structure of equational theories, contexts, and substitution. Finally, we generalize the call-by-name and call-by-value theories of the $\lambda\mu$-calculus by giving a single parametric theory that encompasses both, allowing us to generate a call-by-need instance that defines a calculus of classical and delimited control with lazy evaluation and sharing.

***Categories and Subject Descriptors*** F.3.3 [*Studies of Program Constructs*]: Control primitives

***Keywords*** Delimited Control; Equational Theory; Program Transformation; Continuation-passing Style; Evaluation Strategy

## 1. Introduction

Many programming languages give the programmer the ability to manipulate the flow of control during execution. For example, exception handling mechanisms allow for a faulty execution path to be aborted up to the nearest recovery point, and the callcc operator, which first appeared in Scheme, gives access to the current control state in the program represented as a first-class function called a *continuation*. Griffin [16] observed that certain manipulations of control flow correspond to reasoning in classical logic: in the same way that intuitionistic logic corresponds to the $\lambda$-calculus, adding classical axioms corresponds to adding control operators like callcc. In order to extend the same high-level reasoning tools that apply to open programs in the pure $\lambda$-calculus, Sabry and Felleisen developed equational theories [14, 28, 29] for callcc that not only describe *operational* rules [18] that could be used in an evaluator, but also *observational* guarantees that the evaluator must fulfill. Of note, Sabry's [28] theory of callcc makes use of *continuation variables* that have special properties and can only be instantiated by callcc. As we will see, this not only greatly aids in reasoning about terms with free variables, but also helps in relating the equational and operational interpretations of callcc.

In another line of work, Parigot's [25] $\lambda\mu$-calculus gives a system for computing with classical proofs. As opposed to the previous theories based on the $\lambda$-calculus with additional primitive constants, the $\lambda\mu$-calculus begins with continuation variables integrated into its core, so that control abstractions and applications have the same status as ordinary function abstraction. In this sense, the $\lambda\mu$-calculus presents a native language of control. Both $\lambda$- and $\lambda\mu$-based approaches have their advantages: we have much more experience programming with the $\lambda$-calculus model and the $\lambda\mu$-calculus reveals insights into reasoning about control. Therefore, we aim to present both approaches side-by-side. Typically, a call-by-value version of the $\lambda\mu$-calculus has been related [3] to a different presentation of control based on Felleisen's [14] $\mathcal{C}$ operator, but here we show that it is indeed isomorphic to the call-by-value $\lambda$-calculus with the ordinary callcc operator.

However, callcc isn't the only effectful operation we are interested in — callcc alone is not capable of giving a direct-style representation of other effects like exception handling and mutable references. Instead, there is a variant of this *classical* mode of control, exemplified by the shift and reset operators [7], called *delimited control* or *composable continuations* because the reach of a control operator can be delimited in scope and continuations can be composed like ordinary functions. Filinski [15] showed that delimited control is vastly more powerful than classical control: delimited control operators can give a direct-style representation of any monadic effect in a call-by-value language. The call-by-value $\lambda$-calculus extended with the shift and reset operators [7] has been particularly well-studied, as they have a simple definition in terms of continuation composition and an equational theory due to Kameyama and Hasegawa [19].

Back in the setting of the $\lambda\mu$-calculus, a simple variant of the call-by-name calculus, attributed to de Groote [10], takes a more relaxed view of the syntax. Although these two calculi have been considered the same for typed programs, Saurin [31] discovered that in the untyped setting, de Groote's relaxed variant of the $\lambda\mu$-calculus, there called $\Lambda\mu$, enjoys a form of observational completeness — if two terms cannot be proven equal then they exhibit observably different behavior — which Parigot's original $\lambda\mu$-calculus does not have. Even more, Herbelin and Ghilezan [17] discovered that the call-by-name $\Lambda\mu$-calculus provides a theory for delimited control in a non-strict functional language: the syntactic relaxation gives rise to a native form of composable continuations with a call-by-name evaluation order.

Here, we show that the call-by-value interpretation of the $\lambda\mu$-calculus contains a latent theory of delimited control in strict func-



*ICFP '14*, September 1–6, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2628136.2628147

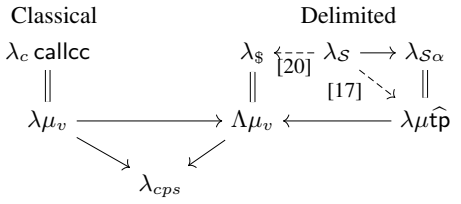


**Figure 1.** Correspondences of classical and delimited control.

tional languages as well. No new programming constructs are needed. No new rules are required. All we need to do is unshackle the syntactic restrictions on programs in the $\lambda\mu$-calculus. Key to this approach is the fact that the syntax of Parigot's $\lambda\mu$-calculus disallows programs from observing the result of invoking a continuation. Removing this restriction gives us an inherit interpretation of such programs from the existing theory, which turns out to be a delimited form of continuations. The call-by-value interpretation of the $\Lambda\mu$-calculus is particularly interesting because the role of the delimiter is more intricate than in the call-by-name setting, and has given rise to distinct control operators with different observational properties. In particular, we discover that the call-by-value $\Lambda\mu_v$-calculus is isomorphic to the call-by-value $\lambda$-calculus with the less-studied $\mathsf{shift}_0$ and $\mathsf{reset}_0$ delimited control operators, as described by Materzok's [20] $\lambda_\$$ theory. Therefore, the syntactic relaxation of the $\lambda\mu$-calculus gives us a canonical bridge from classical to delimited control in both the call-by-name and call-by-value settings.

The connections between the various control calculi are pictured in Figure 1, which contains both $\lambda$-based ($\lambda_c$ callcc [28] for callcc, $\lambda_\$$ [20] for $\mathsf{shift}_0$, $\lambda_\mathcal{S}$ [19] for shift) and $\lambda\mu$-based ($\lambda\mu_v$ [17], $\Lambda\mu_v$, $\lambda\mu\widehat{\mathsf{tp}}$ [17]) approaches. The syntactic embedding from the more restricted $\lambda\mu_v$-calculus to the more relaxed $\Lambda\mu_v$-calculus provides the connection between the classical and delimited worlds, while maintaining soundness and completeness with respect to a typical *continuation-passing style* (CPS) transformation. We also spell out the connection with shift and reset, which can be embedded into $\mathsf{shift}_0$ [22] and the $\lambda\mu\widehat{\mathsf{tp}}$-calculus [17]. In particular, the connection with the $\Lambda\mu_v$-calculus suggests that, similar to Sabry's [28] theory of callcc, Kameyama and Hasegawa's [19] $\lambda_\mathcal{S}$ theory of shift can also benefit from the use of continuation variables (here called $\lambda_{\mathcal{S}\alpha}$) to provide a more powerful tool for reasoning about open programs.

Along the way, we introduce a proof methodology for establishing an equational correspondence [29] between two theories that takes advantage of the typical *compositionality* and *hygiene* of program transformations in order to simplify the correspondence. In short, a general class of transformations preserves the basic structures of equality relations and substitution. Additionally, we connect the study of classical and delimited control in call-by-value and call-by-name languages by presenting a single *parametric* $\Lambda\mu$-calculus, similar to the parametric $\lambda$-calculus [27], that subsumes the equational theories for both evaluation strategies. The parametric calculus can serve as a bridge between the existing work on type systems and semantics for call-by-name $\Lambda\mu$-calculi and call-by-value control operators in the $\lambda$-calculus. We further exercise the parametric $\Lambda\mu$-calculus as a tool for studying evaluation strategies and control by deriving a call-by-need calculus with delimited control from the pure call-by-need $\lambda$-calculus [1].

In summary, our contributions are:

- We show that the call-by-value equational theory for the $\lambda\mu$-calculus that is sound and complete with respect to the usual CPS transformation remains sound and complete for the syntactically relaxed $\Lambda\mu$-calculus.

$$
\begin{aligned}
V \in Value &::= x \mid \lambda x.M \\
M, N \in Term &::= V \mid M\ N \\
E \in EvalCxt &::= \Box \mid E\ M \mid V\ E
\end{aligned}
$$

$$
\begin{array}{ll}
\beta_v & (\lambda x.M)\ V = M\{V/x\} \\
\eta_v & \lambda x.V\ x = V \\
\beta_\Omega & (\lambda x.E[x])\ M = E[M]
\end{array}
$$

**Figure 2.** The syntax and axioms of the pure $\lambda_c$-calculus.

- We demonstrate a general technique of using compositional and hygienic program transformations to more easily relate the syntactic theories of different languages.
- We classify various call-by-value control effects, expressed by primitive control operators, by their correspondence with subsets of the $\Lambda\mu$-calculus: callcc corresponds with Parigot's original $\lambda\mu$-calculus, shift and reset correspond with a subset of $\Lambda\mu$ equal to Ariola *et al.*'s $\lambda\mu\widehat{\mathsf{tp}}$-calculus, and $\mathsf{shift}_0$ and $\mathsf{reset}_0$ correspond to the full $\Lambda\mu$-calculus.
- We generalize the call-by-name and call-by-value theories of the $\Lambda\mu$-calculus into a single parametric theory for control that can be instantiated by different evaluation strategies.
- We generate a strategy for the parametric theory of $\Lambda\mu$ that corresponds with Ariola and Felleisen's [1] call-by-need $\lambda$-calculus, thereby giving a theory of both classical and delimited control with lazy evaluation and sharing (memoization).

Next, we begin by reviewing ways to reason about callcc in the call-by-value $\lambda$-calculus and how Parigot's $\lambda\mu$-calculus provides a well-behaved syntactic restriction of first-class control operators.

## 2. Callcc and classical control

To understand control operators in a strict functional language, we first look to the flow of control in the pure, call-by-value $\lambda_c$-calculus [26], shown in Figure 2. The syntax of $\lambda_c$ is the $\lambda$-calculus, which includes variables $x$, function abstractions $\lambda x.M$, and function calls $M\ N$. We can reason about the behavior of programs in this calculus by using the equational theory of Moggi's [23] computational $\lambda$-calculus, whose axioms are also given in Figure 2.[1] We can more specifically describe how to reduce a term to a value in terms of an operational semantics. Evaluation contexts $E$ point out the current position of evaluation where we need to reduce the program, signified by the "hole" $\Box$, and are defined as usual for the pure call-by-value $\lambda$-calculus. We write $E[M]$ for plugging a term $M$ into the context $E$ by replacing the "hole" $\Box$ with $M$. In particular, evaluation contexts are defined so that the arguments to a function call are evaluated first and a function only receives a value, as described by the $\beta_v$ axiom. Therefore, we can describe the operational semantics of the $\lambda_c$-calculus with the single rule:

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

It follows that $\beta_v$ is an *operational* [18] rule, since it is used during evaluation, whereas $\eta_v$ and $\beta_\Omega$ are instead *observational* properties that do not happen during evaluation, but also do not change the observable result of a program. Because the only operational step

[1] Note that the axioms must always be used in a way that is *hygienic*, so that they respect the static binding of variables given by the usual definitions of *free* and *bound* variables, and $M\{V/x\}$ stands for the usual capture-avoiding substitution of $V$ for $x$ in $M$. In the case of $\lambda_c$, $x$ must not be free in $V$ for $\eta_v$ to hold, and $x$ must not be free in $E$ for $\beta_\Omega$ to hold. Similar such side conditions apply for the axioms in the other theories that follow.

$$
\begin{array}{rr}
C_{lift} & E[\mathsf{callcc}\ M] = \mathsf{callcc}(\lambda\alpha.E[M\ (\lambda x.\alpha(E[x]))]) \\
C_{abort} & E[\alpha\ M] = \alpha\ M \\
C_{current} & \mathsf{callcc}(\lambda\alpha.\alpha\ M) = \mathsf{callcc}(\lambda\alpha.M) \\
C_{elim} & \mathsf{callcc}(\lambda\alpha.M) = M
\end{array}
$$

**Figure 3.** The axioms of control in the $\lambda_c$ callcc equational theory.

matches exactly the $\beta_v$ axiom, it is straightforward to check that the equational theory is as strong as the operational semantics: if the operational semantics reaches a value so does the equational theory.

Languages like Scheme and SML/NJ allow programmers the ability to manipulate the control flow of their programs by giving them access to *continuations* — a representation of an evaluation context — as an object on par with first-class functions. One way to model such control effects in strict functional programming languages is to extend the $\lambda_c$-calculus with primitive operators that create or use continuations. The classic control operator to consider is callcc, first introduced in Scheme. Informally, when we call callcc in Scheme with a (higher-order) function $h$, the operator: (1) takes a snapshot of the current evaluation context, (2) wraps the evaluation context inside of a first-class function (the continuation) which jumps back to the currently active context when called, and (3) calls $h$ with the continuation as its argument.

We can explain the behavior of first-class control more formally by extending the $\lambda_c$-calculus with callcc as a primitive function constant. To account for the continuation functions generated by callcc, we denote the continuation value captured in the context $E$ as $[E]$, giving us the following operational steps:

$$
\begin{array}{rl}
E[(\lambda x.M)\ V] & \mapsto E[M\ \{V/x\}] \\
E[\mathsf{callcc}\ V] & \mapsto E[V\ [E]] \\
E[[E']\ V] & \mapsto E'[V]
\end{array}
$$

The second two rules explain how to evaluate callcc and continuations. To perform a call to callcc in an evaluation context $E$, wrap a copy of the context into the function $[E]$ and pass it along to the argument of callcc. To call $[E]$ at some later point during evaluation, "jump" out of the current evaluation context and restore the state of the program to $E$, plugging the argument into the hole of the context.

*Example* 1. To illustrate the behavior of callcc, consider the following term that captures its (empty) evaluation context and returns a function that, when given an input $x$, ends the program by "jumping" out of its calling context with the constant function $\lambda\_.x$.

$$
\begin{array}{rl}
\mathsf{callcc}(\lambda k.\lambda x.k\ (\lambda\_.x)) & \mapsto (\lambda k.\lambda x.k\ (\lambda\_.x))\ [\Box] \\
& \mapsto \lambda x.[\Box]\ (\lambda\_.x)
\end{array}
$$

If we use the function $\lambda x.[\Box]\ (\lambda\_.x)$ in a larger program, we can see how the continuation $[\Box]$ immediately ends the program and returns a constant function as the final result.

$$
\begin{array}{rl}
((\lambda x.[\Box]\ (\lambda\_.x))\ 2) + 10 & \mapsto ([\Box]\ (\lambda\_.2)) + 10 \\
& \mapsto \lambda\_.2 \qquad \textit{End example } 1.
\end{array}
$$

Like with the pure calculus, we would like to also describe the behavior of callcc with an equational theory. However, because the operational steps for callcc and continuations manipulate their *entire* evaluation context, we cannot just apply them in an arbitrary sub-term of a program. This sort of issue is addressed by the equational theories for control developed by Felleisen and Sabry [14, 28, 29]. In particular, we will use Sabry's [28] theory of control, here called $\lambda_c$ callcc, which extends the $\lambda_c$ theory with the axioms of callcc shown in Figure 3. This theory extends the $\lambda_c$-calculus with the primitive function callcc and makes use of a distinguished set of *continuation variables*, $\alpha$, $\beta$, *etc*. Continuation variables may only be introduced by the form $\mathsf{callcc}(\lambda\alpha.M)$, so that a term like $(\lambda\alpha.\alpha\ y)\ (\lambda x.x)$ is syntactically illegal. Note that this means there are two different syntactic forms that refer to callcc, either as an ordinary function application $\mathsf{callcc}\ (\lambda x.M)$ or the special form $\mathsf{callcc}(\lambda\alpha.M)$, which are equated by $C_{lift}$.

Unfortunately, unlike the case with the pure $\lambda_c$-calculus, it is no longer so clear how the equational theory relates to the operational semantics. In actuality, the equational theory is no longer as strong as the operational semantics. The program in Example 1 shows the problem, where there is no way to eliminate the top-most application of callcc in the term $\mathsf{callcc}(\lambda k.\lambda x.k\ (\lambda\_.x))$ so that it cannot be equated to a value. Therefore, for general terms there is a disconnect between the operational and equational accounts of callcc. However, is it possible that we can find some specific subset of "well-behaved" terms where the two semantics always coincide?

*Example* 2. Let's extend the term in Example 1 so that rather than returning its final result, it instead finishes by "jumping" to some continuation $\alpha$. The operational interpretation of the term is similar to before, where we now supply $\alpha$ with a function that forms a constant function and jumps out of its calling context back to $\alpha$.

$$
\begin{array}{rl}
\alpha\ (\mathsf{callcc}(\lambda k.\lambda x.k\ (\lambda\_.x))) & \mapsto \alpha\ ((\lambda k.\lambda x.k\ (\lambda\_.x))[\alpha\ \Box]) \\
& \mapsto \alpha\ (\lambda x.[\alpha\ \Box]\ (\lambda\_.x))
\end{array}
$$

If we use this function in a larger program, it behaves similarly to the function from Example 1, except that we now end the program by supplying a result to $\alpha$ rather than implicitly returning a value.

$$
((\lambda x.[\alpha\ \Box]\ (\lambda\_.x))\ 2) + 10 \mapsto\!\!\!\!\rightarrow \alpha\ (\lambda\_.2)
$$

The equational theory in Figure 3 is capable of simulating this operational evaluation. In particular, the first step, which completely eliminates callcc, is achieved with the $C_{lift}$, $C_{abort}$, and $C_{elim}$ axioms, resulting in a jump that passes a similar function to $\alpha$. This sequence of steps is possible because we know that $\alpha$ will inevitably jump out of its calling context, so that any surrounding evaluation context is redundant garbage.

$$
\begin{array}{ll}
& \alpha\ (\mathsf{callcc}(\lambda k.\lambda x.k\ (\lambda b.lankx))) \\
=_{C_{lift}} & \mathsf{callcc}(\lambda\beta.\alpha\ ((\lambda k.\lambda x.k\ (\lambda\_.x))\ (\lambda y.\beta\ (\alpha\ y)))) \\
=_{C_{abort}} & \mathsf{callcc}(\lambda\beta.\alpha\ ((\lambda k.\lambda x.k\ (\lambda\_.x))\ (\lambda y.\alpha\ y))) \\
=_{C_{elim}} & \alpha\ ((\lambda k.\lambda x.k\ (\lambda\_.x))\ (\lambda y.\alpha\ y)) \\
=_{\beta_v} & \alpha\ (\lambda x.(\lambda y.\alpha\ y)\ (\lambda\_.x))
\end{array}
$$

Thus, calling a continuation variable like $\alpha$ serves as a marker that signifies the end of a usable evaluation context, and allows us to recognize the rest of the calling context as garbage and end the call to callcc early. The function that $\alpha$ receives behaves exactly like the one that was created by the operational semantics. For example, if we call it in the same context as before, we still end up with a jump that returns the constant function $\lambda\_.2$ to $\alpha$.

$$
\begin{array}{rl}
((\lambda x.(\lambda y.\alpha\ y)\ (\lambda\_.x))\ 2) + 10 & =_{\beta_v} ((\lambda y.\alpha\ y)\ (\lambda\_.2)) + 10 \\
& =_{\beta_v} (\alpha\ (\lambda\_.2)) + 10 \\
& =_{C_{abort}} \alpha\ (\lambda\_.2)
\end{array}
$$

*End example* 2.

Example 2 shows how "jumps" are important for understanding callcc. By starting with a jump instead of a general term, we fix the mismatch between the operational semantics and equational theory. As it turns out [3], this concept is explicitly expressed by Parigot's $\lambda\mu$-calculus [25], in which jumps, referred to as *commands*, are given prominent status. A call-by-value version of the $\lambda\mu$-calculus is given in Figure 4, where we extend the syntax of the $\lambda$-calculus

$$V \in Value ::= x \mid \lambda x.M$$
$$M, N \in Term ::= V \mid M\ N \mid \mu\alpha.c$$
$$c \in Command ::= [\alpha]M$$
$$E \in EvalCxt ::= \Box \mid E\ M \mid V\ E$$

$$\mu_v \qquad [\alpha](E[\mu\beta.c]) = c\{[\alpha](E[M])/[\beta]M\}$$
$$\eta_\mu \qquad \mu\alpha.[\alpha]M = M$$
$$\beta_\mu \qquad (\lambda x.\mu\alpha.[\beta]M)\ N = \mu\alpha.[\beta](\lambda x.M)\ N$$

**Figure 4.** The syntax and axioms of control in the $\lambda\mu_v$-calculus.

with co-variables $\alpha, \beta, \ldots$ along with their abstraction $\mu\alpha.c$ and application $[\alpha]M$. Intuitively, the calculus can be viewed as imposing a syntactic restriction on the use of the callcc operator, where it must always be applied to a $\lambda$-abstraction whose body immediately jumps, as in $\mathsf{callcc}(\lambda\alpha.\beta\ M)$.

The $\lambda\mu_v$ equational theory extends $\lambda_c$ with the axioms of control given in Figure 4. Note that the axiom describing the behavior of a $\mu$-abstraction makes use of *structural substitution* [2] to perform a pattern-matching substitution on terms, following the normal criteria for static scope and avoiding variable capture. The substitution $\{[\alpha]E[N]/[\beta]N\}$ replaces every command of the form $[\beta]N$, for an arbitrary term $N$, with $[\alpha]E[N]$. We can also describe the $\lambda\mu_v$-calculus in terms of an operational semantics, similar to $\lambda_c$ and $\lambda_c$ callcc. In particular, we always evaluate a command of the form $[\alpha]M$ according to the following two rules:

$$[\alpha]E[(\lambda x.M)\ V] \mapsto [\alpha]E[M\{V/x\}]$$
$$[\alpha]E[\mu\beta.c] \mapsto c\{[\alpha](E[M])/[\beta]M\}$$

Notice that the first rule is exactly the $\beta_v$ axiom, as usual, and the second rule is exactly the $\mu_v$ axiom. Therefore, it is straightforward to show how the operational semantics and equational theory coincide. The $\mu_v$ axiom is the operational rule for implementing $\mu$-abstractions, whereas $\eta_\mu$ and $\beta_\mu$ express observational properties of control.

*Example* 3. We can transcribe the $\lambda_c$ callcc term from Example 2 into a command in the $\lambda\mu_v$-calculus by explicating some of the implicit behavior of the callcc operator. In particular, we need to explicitly say that we are returning a result to our current context, and state that invoking the bound co-variable will abort with the form $\mu\_.[\beta]M$. This gives us a command that produces a similar result as Example 2.

$$[\alpha]\mu\beta.[\beta](\lambda x.\mu\_.[\beta](\lambda\_.x)) \mapsto [\alpha](\lambda x.\mu\_.[\alpha](\lambda\_.x))$$

As before, when we call the function passed to $\alpha$ in a similar context, we see the similar result $[\alpha]\lambda\_.2$.

$$[\delta](((\lambda x.\mu\_.[\alpha](\lambda\_.x))\ 2) + 10) \mapsto [\delta]((\mu\_.[\alpha](\lambda\_.2)) + 10)$$
$$\mapsto [\alpha](\lambda\_.2)$$

The $\lambda\mu_v$ equational theory gives us the same result as the above: every operational step is an application of either the $\beta_v$ or $\mu_v$ axioms. *End example* 3.

## 3. Compositional equational correspondence

We have mentioned that Parigot's $\lambda\mu$-calculus models the $\lambda$-calculus with callcc, but how can we relate the two equational theories? To answer this question, we will turn to Sabry and Felleisen's [29] concept of *equational correspondence*. In the general case, we have two equational theories — a theory $Eq_S$ for equating programs in language $L_S$ (for source) and a theory $Eq_T$ for equating programs in language $L_T$ (for target). We can say that $Eq_S$ is in equational correspondence with $Eq_T$ if and only if there are two translations, $\mathcal{T}^{\mathcal{S}}$ from $L_S$ to $L_T$ and $\mathcal{S}^{\mathcal{T}}$ from $L_T$ back to $L_S$, that satisfy the following two criteria:[2]

(1) $\mathcal{T}^{\mathcal{S}}$ and $\mathcal{S}^{\mathcal{T}}$ are *inverses* up to $Eq_S$ and $Eq_T$: $Eq_S \vdash \mathcal{S}^{\mathcal{T}}[\![\mathcal{T}^{\mathcal{S}}[\![M]\!]]\!] = M$ and $Eq_T \vdash \mathcal{T}^{\mathcal{S}}[\![\mathcal{S}^{\mathcal{T}}[\![P]\!]]\!] = P$.

(2) $Eq_S$ and $Eq_T$ are *sound* with respect to each other: $Eq_S \vdash M = N$ implies $Eq_T \vdash \mathcal{T}^{\mathcal{S}}[\![M]\!] = \mathcal{T}^{\mathcal{S}}[\![N]\!]$ and $Eq_T \vdash P = Q$ implies $Eq_S \vdash \mathcal{S}^{\mathcal{T}}[\![P]\!] = \mathcal{S}^{\mathcal{T}}[\![Q]\!]$.

Intuitively, criteria (1) guarantees that both languages are big enough to reflect each other, and criteria (2) guarantees that the translations preserve the intended semantics. When taken together, both criteria are enough to prove the additional property:

(3) $Eq_S$ and $Eq_T$ are *complete* with respect to each other: $Eq_T \vdash \mathcal{T}^{\mathcal{S}}[\![M]\!] = \mathcal{T}^{\mathcal{S}}[\![N]\!]$ implies $Eq_S \vdash M = N$ and $Eq_S \vdash \mathcal{S}^{\mathcal{T}}[\![P]\!] = \mathcal{S}^{\mathcal{T}}[\![Q]\!]$ implies $Eq_T \vdash P = Q$.

The *soundness and completeness* of a transformation is a weaker result than an equational correspondence: it expresses an embedding of the source language into the target such that the full source language corresponds to the image of the transformation. An equational correspondence adds the additional guarantee that the transformation covers the full target language, forming an isomorphism up to the two equational theories.

In order to establish an equational correspondence, we must surely show that the axioms of the two theories are inter-derivable. However, knowing that the axioms work out is not enough to show that every equation is preserved by the transformations — we also have to take care of the additional inference principles of the equational theories. Uses of reflexivity, transitivity, and symmetry in the two theories coincides for any pair of transformations. The biggest obstacle is *congruence*, the property that equality may be lifted into any context $C$ so that $M = N$ implies $C[M] = C[N]$. However, we can guarantee that even congruence is preserved for a wide class of transformations. Therefore, we introduce a general proof methodology that takes advantage of typical properties of transformations to simplify establishing the correspondence.

Fortunately, many program transformations are *compositional*, meaning that the transformation of a term depends only on the transformation of its subterms. Compositionality of a transform, $[\![\_]\!]$, implies a form of context-free interpretation, so that for any context $C$ in the source language, there exists a unique context in the target, denoted $[\![C]\!]$, such that $[\![C[M]]\!] \triangleq [\![C]\!][[\![M]\!]]$ for any $M$.[3] Crucially, there is a well-defined meaning for every context that can be given in isolation of both its input and its surrounding context. This property guarantees that the transformations respect congruence: if we apply the equation $Eq_S \vdash M = N$ inside a context $C$, then by congruence in the target language we also have that $Eq_T \vdash [\![C]\!][[\![M]\!]] = [\![C]\!][[\![N]\!]]$, where the transformation of $C$ is not affected by the fact that we are plugging in a different term. Therefore, when working with an equational correspondence by compositional transformations, it is enough to show that the axioms of one theory are provable by the other — the other properties of equality follow.

[2] The notation $Eq_S \vdash M = N$ means that $M$ and $N$ are equated by $Eq_S$.

[3] We can plug one context into another, $C[C']$, in the obvious way by treating $C'$ as a term. This means that contexts form a monoid with the empty context, $\Box$, as the identity element and plugging-in as the composition operation. Therefore, a compositional transformation can be seen as a monoid homomorphism that respects the compositional nature of contexts: $[\![\Box]\!] = \Box$ and $[\![C[C']]\!] = [\![C]\!][[\![C']\!]]$. In a language with different syntactic types, like $\lambda\mu_v$, contexts instead form a category and a compositional transformation is a functor between contexts in the source and target.

**Proposition 1.** *Compositional transformations preserve the structure of an equality relation: reflexivity, transitivity, symmetry, and congruence.*

Another obstacle that compositionality allows us to avoid is handling substitution. Since some axioms, like $\beta_v$, may rely on substitution, we end up having to show that substitution in the source and target language are compatible with one another. Since we are working with call-by-value languages that only allow substitution for a subset of terms, we would need a property of the form $[\![M]\!]\left\{[\![V]\!]^V/x\right\} = [\![M\left\{V/x\right\}]\!]$, where $V$ is a value and $[\![V]\!]^V$ is the embedding of $V$ into a target-level value.[4] First, we point out that ordinary capture-avoiding substitution can be implemented by an iterative procedure that replaces the free occurrences of the variable one by one. For example, when we are substituting $V$ for $x$ in a term $M$, we can identify a particular occurrence of the free variable $x$ in $M = C[x]$ and replace $x$ with $V$. If we can't find such an $x$ then the substitution does nothing. Second, we also require that the transformation is *hygienic*, so that it does not capture or escape the static variables of its input. In other words, hygiene guarantees that $C$ captures the variable $x$ if and only if $[\![C]\!]$ does. Third, if source variables are values and are embedded directly into target variables, so that $[\![x]\!]^V \triangleq x$, then this procedure maps directly to the target language:

$$[\![C[x]]\!]\left\{[\![V]\!]^V/x\right\} = [\![C]\!][[\![x]\!]^V]\left\{[\![V]\!]^V/x\right\}$$
$$= [\![C]\!][[\![V]\!]^V]\left\{[\![V]\!]^V/x\right\} = [\![C[V]]\!]\left\{[\![V]\!]^V/x\right\}$$

Structural substitution, which replaces one context with another, follows similarly.

**Proposition 2.** *Let $[\![\_]\!]$ be a hygienic, compositional transformation, and $[\![\_]\!]^V$ be the transformation of substitutable terms. If $[\![x]\!]^V \triangleq x$ then $[\![M]\!]\left\{[\![V]\!]^V/x\right\} \triangleq [\![M\left\{V/x\right\}]\!]$. Furthermore, $[\![M]\!]\left\{[\![C']\!]/[\![C]\!]\right\} \triangleq [\![M\left\{C'/C\right\}]\!]$.*

Therefore, in order to establish an equational correspondence with hygienic, compositional transformations, we only need to show that (1) the two transformations are inverses, and (2) the axioms of the two theories, which may rely on substitution, are inter-derivable. The rest of the correspondence follows generically from Propositions 1 and 2. Furthermore, one nice property about an equational correspondence, as well as soundness and completeness, is that it composes transitively [30]: if $Eq_S$ is in equational correspondence with $Eq_T$ and $Eq_T$ is in equational correspondence with $Eq_U$, then $Eq_S$ is also in equational correspondence with $Eq_U$. This fact lets us relate two theories in several steps by going through some intermediate languages.

## 4. Correspondence of $\lambda\mu_v$ and callcc

We can apply the proof technique described in Section 3 to formally relate the $\lambda\mu_v$-calculus to callcc in a call-by-value $\lambda$-calculus. First, we need to explain how to translate one language into the other in terms of two inverse, compositional transformations, spelling out the details of the transcription process used in Example 3. The interesting points of the syntactic translations are given in Figure 5, and the rest is defined homomorphically. With these two transformations, the equational correspondence follows straightforwardly from the outlined methodology. Compositionality and hygiene of the transformations is confirmed by observing that every clause in the grammars of $\lambda\mu_v$ and $\lambda_c$ callcc is exactly defined by one clause

[4] Note that in a typical call-by-value language where values are implicitly included in general terms, we may have that the more general transformation $[\![V]\!]$ introduces a non-trivial context $C$, so that $[\![V]\!] \triangleq C[[\![V]\!]^V]$.

$$\mathcal{K}[\![\mu\alpha.c]\!] \triangleq \mathsf{callcc}(\lambda\alpha.\mathcal{K}[\![c]\!])$$
$$\mathcal{K}[\![[\alpha]M]\!] \triangleq \alpha\,\mathcal{K}[\![M]\!]$$

$$\mathcal{K}^{-1}[\![\mathsf{callcc}]\!] \triangleq \lambda h.\mu\alpha.[\alpha]h\ (\lambda x.\mu\_.[\alpha]x)$$
$$\mathcal{K}^{-1}[\![\mathsf{callcc}(\lambda\alpha.M)]\!] \triangleq \mu\alpha.[\alpha]\mathcal{K}^{-1}[\![M]\!]$$
$$\mathcal{K}^{-1}[\![\alpha]\!] \triangleq \lambda x.\mu\_.[\alpha]x$$

**Figure 5.** The isomorphism between $\lambda\mu_v$ and $\lambda_c$ callcc.

$$\mathcal{C}[\![x]\!]^V \triangleq x$$
$$\mathcal{C}[\![\lambda x.M]\!]^V \triangleq \lambda(x,\alpha).\mathcal{C}[\![M]\!]^M\ \alpha$$
$$\mathcal{C}[\![V]\!]^M \triangleq \lambda\alpha.\alpha\ \mathcal{C}[\![V]\!]^V$$
$$\mathcal{C}[\![M\ N]\!]^M \triangleq \lambda\alpha.\mathcal{C}[\![M]\!]^M\ \lambda x.\mathcal{C}[\![N]\!]^M\lambda y.x\ (y,\alpha)$$
$$\mathcal{C}[\![\mu\alpha.c]\!]^M \triangleq \lambda\alpha.\mathcal{C}[\![c]\!]^c$$
$$\mathcal{C}[\![[\alpha]M]\!]^c \triangleq \mathcal{C}[\![M]\!]^M\ \alpha$$

**Figure 6.** CPS transformation $\mathcal{C}$ from $\lambda\mu_v$ to the $\lambda$-calculus.

in the transformation, that there are no additional parameters to the transformations, and that the transformations do not cause capture or escape of free static variables.

**Theorem 1.** *$\lambda_c$ callcc is in equational correspondence with $\lambda\mu_v$.*

*Proof.* The equational correspondence is formed by the $\mathcal{K}$ and $\mathcal{K}^{-1}$ transformations. Since the transformations are compositional and hygienic, we only need to show (1) that they are inverses of one another and (2) that the axioms of the two theories are inter-derivable. Criteria (1) follows by mutual induction on terms, values, *etc*. Criteria (2) follows by cases on the axioms. □

Crucially, the $C_{abort}$ [28] axiom lets us recognize the abortive nature of continuations, even for free continuation variables. This lets us bridge the gap between Felleisen's [14] $\mathcal{C}$ operator and callcc. Intuitively, $\mathcal{C}(\lambda\alpha.M)$ may be encoded as $\mathsf{callcc}(\lambda\alpha.\mathsf{tp}\ M)$, where tp is a free continuation variable that stands for the "top-level." We then run a term $M$ by explicitly marking the top-level of the program as in the jump $\mathsf{tp}\ M$. This same free tp co-variable also shows up in the correspondence [3] between $\mathcal{C}$ and $\lambda\mu_v$.

We can also use a similar methodology to verify that the equational theory of the $\lambda\mu_v$-calculus makes sense — that there are no equalities between programs that behave differently, or that we are not missing some crucial axiom that we would need to relate two similar programs. A now common technique for getting around this dilemma is to reduce the theory in question to another, more well-understood theory. This allows for the established understanding of the target theory to be reflected back to the source. In particular, it is typical to translate the language through a *continuation-passing style* (CPS) transformation into the $\lambda$-calculus (or a variant thereof), and to use the induced semantics that comes from the resulting $\lambda$-calculus terms as a definition of the semantics of the source language. The goal, then, is to establish an equational correspondence between the source language and the image of the CPS transformation in the target language.

Figure 6 gives a CPS transformation for terms, values, and commands of the $\lambda\mu_v$-calculus, based on Plotkin's [26] call-by-value CPS transformation. The CPS transformation is defined so that it does not depend on the evaluation strategy of the target $\lambda$-calculus

$$c \in Command ::= [q]M \qquad q \in CoTerm ::= \alpha \mid \tilde{\mu}x.c$$

$$\begin{array}{llll} \beta_v & (\lambda x.M)\ V = M\,\{V/x\} & \eta_v & \lambda x.V\ x = V \\ \mu_q & [q]\mu\alpha.c = c\,\{q/\alpha\} & \eta_\mu & \mu\alpha.[\alpha]M = M \\ \tilde{\mu}_v & [\tilde{\mu}x.c]V = c\,\{V/x\} & \eta_{\tilde{\mu}} & \tilde{\mu}x.[q]x = q \\ & \varsigma_v \quad E[M] = \mu\alpha.[\tilde{\mu}x.[\alpha]E[x]]M & & \end{array}$$

**Figure 7.** The syntax and axioms of control in the $\lambda\mu\tilde{\mu}_v$-calculus.

$$\mathcal{C}[\![[q]M]\!]^c \triangleq \mathcal{C}[\![M]\!]^M\ \mathcal{C}[\![q]\!]^q \quad \mathcal{C}[\![\alpha]\!]^q \triangleq \alpha \quad \mathcal{C}[\![\tilde{\mu}x.c]\!]^q \triangleq \lambda x.\mathcal{C}[\![c]\!]^c$$

**Figure 8.** CPS transformation $\mathcal{C}$ for co-terms of $\lambda\mu\tilde{\mu}_v$.

— the argument to every function call is a value, so that the resulting term behaves the same according to both a call-by-name and call-by-value evaluation strategy. Intuitively, the transformation turns every term of the $\lambda\mu_v$-calculus into a function that accepts a continuation as a description of the entire future of the computation. The way to read the term $\mathcal{C}[\![M]\!]^M\ \alpha$ is "run $M$, and when it is done, the value it returns is passed to $\alpha$." For instance, a value immediately returns, so the transformation of a value immediately calls the continuation and allows it to take over. The most complicated case is for function application $M\ N$, which can be read as: in a calling context $\alpha$, first (1) run $M$ and wait for it to return a value $x$, second (2) run $N$ and wait for it to return a value $y$, and third (3) call $x$ with $y$ inside the original calling context $\alpha$. Notice that the control constructs of $\lambda\mu_v$ translate directly to the concept of continuations used by the transformation: a $\mu$-abstraction gives a name to the continuation and a command runs a term in a chosen continuation. Also note that, similar to the axioms of $\lambda\mu_v$, the CPS transformation must be hygienic and take care to respect the static binding of variables. For instance, in the transformation $\mathcal{C}[\![\lambda x.M]\!]^V \triangleq \lambda(x,\alpha).\mathcal{C}[\![M]\!]\ \alpha$, the $x$ introduced by the $\lambda$-abstraction on the left must be the *same* $x$ that is referenced by $\mathcal{C}[\![M]\!]$ on the right, but the $\alpha$ that is introduced must *not* be free in $\mathcal{C}[\![M]\!]$ to avoid unintentional variable capture.

There are some complications that arise when trying to form the equational correspondence between the $\lambda\mu_v$ and $\lambda$ theories, which is typical when reasoning about CPS transformations. The primary issue is that we need to translate CPS terms back to the source calculus. For instance, if we see the CPS term $\alpha\ V$, we want to turn it into some command $[\alpha]V'$. However, for other cases, the reverse CPS transformation is not so obvious. Therefore, we introduce an intermediate calculus that is isomorphic to $\lambda\mu_v$, but is more closely connected with the CPS transformation. Borrowing syntax for the sequent calculus from Curien and Herbelin [6], we extend the $\lambda\mu_v$-calculus with $\tilde{\mu}$-abstractions as the opposite of $\mu$-abstractions, giving us the $\lambda\mu\tilde{\mu}_v$-calculus in Figure 7. Note that this calculus no longer relies on structural substitution to capture an evaluation context, but instead first converts an evaluation context into a $\tilde{\mu}$-abstraction (by the $\varsigma_v$ axiom) as in the following example:

$$\begin{aligned} f\ (\mu\alpha.[\beta]\lambda x.\mu\_.[\alpha]x) &=_{\varsigma_v} \mu\alpha.'[\tilde{\mu}y.[\alpha'](f\ y)]\mu\alpha.[\beta]\lambda x.\mu\_.[\alpha]x \\ &=_{\mu_q} \mu\alpha.'[\beta]\lambda x.\mu\_.[\tilde{\mu}y.[\alpha'](f\ y)]x \\ &=_{\tilde{\mu}_v} \mu\alpha.'[\beta]\lambda x.\mu\_.[\alpha'](f\ x) \end{aligned}$$

Intuitively, the $\varsigma_v$ axiom recognizes the fact that evaluation contexts are *transformations* from inputs to outputs, and gives a name both to the input end, $x$, and the output end, $\alpha$. In other words, an evaluation context is not only missing an input denoted by $\square$, it is *also* missing an "output" that says what to do with the result. We

$$M^c ::= V^M\ V^q \mid V^q\ V^V \mid V^V\ (V^V, V^q) \quad V^M ::= \lambda\alpha.M^c$$
$$V^q ::= \alpha \mid \lambda x.M^c \quad V^V ::= x \mid \lambda(x,\alpha).M^c$$

**Figure 9.** The image of the CPS transformation of $\lambda\mu\tilde{\mu}_v$ by $\mathcal{C}$.

$$\begin{array}{rl} \mathcal{C}^{-1}[\![V^M\ V^q]\!]^c \triangleq & [\mathcal{C}^{-1}[\![V^q]\!]^q]\mathcal{C}^{-1}[\![V^M]\!]^M \\ \mathcal{C}^{-1}[\![V^q\ V^V]\!]^c \triangleq & [\mathcal{C}^{-1}[\![V^q]\!]^q]\mathcal{C}^{-1}[\![V^V]\!]^V \\ \mathcal{C}^{-1}[\![V^V\ (V'^V, V^q)]\!]^c \triangleq & [\mathcal{C}^{-1}[\![V^q]\!]^q]\mathcal{C}^{-1}[\![V^V]\!]^V\ \mathcal{C}^{-1}[\![V'^V]\!]^V \\ \mathcal{C}^{-1}[\![\lambda\alpha.M^c]\!]^M \triangleq & \mu\alpha.\mathcal{C}^{-1}[\![M^c]\!]^c \end{array}$$
$$\begin{array}{rlrl} \mathcal{C}^{-1}[\![\alpha]\!]^q \triangleq & \alpha & \mathcal{C}^{-1}[\![\lambda x.M^c]\!]^q \triangleq & \tilde{\mu}x.\mathcal{C}^{-1}[\![M^c]\!]^c \\ \mathcal{C}^{-1}[\![x]\!]^V \triangleq & x & \mathcal{C}^{-1}[\![\lambda(x,\alpha).M^c]\!]^V \triangleq & \lambda x.\mu\alpha.\mathcal{C}^{-1}[\![M^c]\!]^c \end{array}$$

**Figure 10.** The inverse of $\mathcal{C}$ on the image of $\lambda\mu\tilde{\mu}_v$.

also extend the CPS transformation as shown in Figure 8 to account for the new syntax, showing how the $\tilde{\mu}$-abstractions correspond exactly with $\lambda$-abstraction continuations. That way, it is relatively easy to translate a continuation of the form $\lambda x.M$ created by the CPS transformation back into the $\lambda\mu\tilde{\mu}_v$-calculus.

The second issue is that the $\mathcal{C}$ transformation does not "reach" every program in the $\lambda$-calculus, even if we restrict ourselves to terms where function arguments are always values. Because of this fact, it is impossible to translate the entire $\lambda$-calculus back into the $\lambda\mu\tilde{\mu}_v$-calculus (or $\lambda\mu_v$-calculus) in a meaningful way. Therefore, we need to determine the *image* of the CPS transformation of $\lambda\mu\tilde{\mu}_v$ — that is, the smallest subset of the $\lambda$-calculus that includes the transformation of every $\lambda\mu\tilde{\mu}_v$ term and is closed under reduction. The general target of the transformation is the $\lambda$-calculus extended with pairs constructed as $(M, N)$ and deconstructed by $\lambda$-abstractions of the form $\lambda(x,y).M$. For our purposes, we reason about CPS terms with the usual $\beta$ and $\eta$ axioms of the $\lambda$-calculus along with the following two rules for pairs:

$$\begin{array}{ll} \beta^\times & (\lambda(x,y).M)\ (N_1, N_2) = M\,\{N_1, N_2/x, y\} \\ \eta^\times & \lambda(x,y).M\ (x,y) = M \end{array}$$

The subset of this calculus reachable by the $\mathcal{C}$ transform is shown in Figure 9. Intuitively, $M^c$, $V^M$, $V^q$, and $V^V$ are the image of commands, terms, co-terms and values of the $\lambda\mu\tilde{\mu}_v$-calculus, respectively. Notice that $V^V$ is always a function deconstructing a pair, and all instances of $\lambda(x,y).M$ are in $V^V$, so that we only need to apply the $\eta^\times$ axiom to values in $V^V$.

We can now provide an inverse translation from this subset of the $\lambda$-calculus back to the $\lambda\mu\tilde{\mu}_v$-calculus, as shown in Figure 10. Note that sending a $\lambda\mu\tilde{\mu}_v$ term through both transformations changes the term. Most of the changes, for values and for functions, just introduce an extra $\eta_\mu$ expansion. The worst case is if we start with the application $M\ N$, where we end up with

$$\mathcal{C}^{-1}[\![\mathcal{C}[\![M\ N]\!]^M]\!]^M \triangleq \mu\alpha.[\tilde{\mu}x.[\tilde{\mu}y.[\alpha]x\ y]N']M'$$

This term is reminiscent of Sabry's [28] technique of *continuation-grabbing style*, where we write a CPS term in more abstract syntax as a way to relate to languages with control effects. In that light, we are relying on $\mu$-abstractions to grab the continuation, on co-terms to represent continuations, and on commands to represent sending a continuation to a term.

With both transformations, it is now relatively straightforward to show the soundness and completeness of the $\lambda\mu\tilde{\mu}_v$ equational theory in terms of its CPS transformation using the methodology

outlined in Section 3. We can check that the axioms of the two theories are provable with respect to the transformations. The $\beta_v$, $\mu_q$, and $\tilde{\mu}_v$ axioms correspond with the various forms of $\beta$ equality in the image of the CPS transformation, and the $\eta_v$, $\eta_\mu$, and $\eta_{\tilde{\mu}}$ axioms correspond with the $\eta$ equalities. The last axiom, $\varsigma_v$, is necessary for proving that the $\mathcal{C}$ and $\mathcal{C}^{-1}$ transforms are inverses for function application, by simplifying the expanded term back down to the form $M\ N$.

**Theorem 2.** *The $\lambda\mu\tilde{\mu}_v$ equational theory is sound and complete with respect to the $\beta\eta$ theory of the $\lambda$-calculus with pairs.*

*Proof.* This follows from the fact that the $\mathcal{C}$ and $\mathcal{C}^{-1}$ transformations form an equational correspondence between $\lambda\mu\tilde{\mu}_v$ and the image of $\mathcal{C}$ in the $\lambda$-calculus. Criteria (1) holds by mutual induction on terms, values, *etc*. Criteria (2) holds by the fact that the transformations are compositional and hygienic and the axioms are interderivable. Criteria (3) for $\mathcal{C}$ follows from criteria (1) and (2). □

Additionally, we can show how the $\lambda\mu\tilde{\mu}_v$-calculus relates to the original $\lambda\mu_v$-calculus by the same methodology. Specifically, the $\tilde{\mu}$-abstractions can be considered syntactic sugar on top of the $\lambda\mu_v$-calculus, by using one free co-variable $\delta$, which the $\lambda\mu\tilde{\mu}_v$ theory is able to erase

$$\lambda\mu\tilde{\mu}_v \vdash [\tilde{\mu}x.c]M =_{\varsigma_v,\mu_q,\beta_v} [\delta](\lambda x.\mu\_.c)\ M$$

so that we have not added anything essential that wasn't already in the $\lambda\mu_v$-calculus.[5] Therefore, we are justified in considering $\lambda\mu_v$ as the canonical calculus, and treating $\tilde{\mu}$-abstractions as syntactic sugar and the axioms of $\lambda\mu\tilde{\mu}_v$ in Figure 7 as derived equalities in the $\lambda\mu_v$ theory. By checking the axioms of $\lambda\mu\tilde{\mu}_v$ and $\lambda\mu_v$, we find that the two are in equational correspondence with one another.

**Theorem 3.** *$\lambda\mu_v$ is in equational correspondence with $\lambda\mu\tilde{\mu}_v$.*

*Proof.* The equational correspondence is formed by the compositional transformations that (a) inject $\lambda\mu_v$ terms into $\lambda\mu\tilde{\mu}_v$ unchanged, and (b) desugaring $\tilde{\mu}$-abstractions everywhere in a $\lambda\mu\tilde{\mu}_v$ term, by using a fresh co-variable $\delta$. Criteria (1) and (2) follow from the same proof methodology as in Theorem 1. □

By putting everything together, as a corollary we get an alternate proof [28, 29] of the soundness and completeness of $\lambda_c$ callcc with respect to its CPS transformation by factoring through $\lambda\mu_v$. Of note, the CPS transformation for callcc that we get out, after simplification, is the expected one:

$$\mathcal{C}[\![\mathcal{K}^{-1}[\![\mathsf{callcc}]\!]]\!]^V = \lambda(h,\alpha).h\ ((\lambda(x,\_).\alpha\ x),\alpha)$$

By comparing this CPS transformation of callcc with the encoding of callcc in the $\lambda\mu_v$-calculus, it shows how the $\lambda\mu_v$-calculus may be seen as a representation of control that is "closer" to the underlying CPS — and $\lambda\mu\tilde{\mu}_v$ is even closer still.

## 5. Relaxing the syntax

There is a variant of Parigot's $\lambda\mu$-calculus, originating from de Groote [10], which takes a more relaxed approach to the syntax of the calculus. In essence, the distinction between commands and terms is blurred, allowing them to mingle in new ways. Like the original $\lambda\mu$-calculus, this variant was originally studied as a typed calculus with a call-by-name interpretation. The command $[\alpha]M$, when considered as a term, can be given the type $\bot$ denoting falsehood. However, Saurin [31] showed that when considered in the untyped setting, this variant of the $\lambda\mu$-calculus, named the $\Lambda\mu$-calculus, enjoys certain properties that do not hold for Parigot's

[5] As a corollary, Theorem 2 implies that $\mathcal{C}$ in Figure 8 is the same as first desugaring away all $\tilde{\mu}$-abstractions and then taking the CPS transformation.

$$\begin{array}{rc}
 & V \in Value ::= x \mid \lambda x.M \\
 & M,N \in Term ::= V \mid M\ N \mid \mu\alpha.M \mid [\alpha]M \\
 & E \in EvalCxt ::= \Box \mid E\ M \mid V\ E \\
\mu_v & [\alpha](E[\mu\beta.M]) = M\,\{[\alpha](E[N])/[\beta]N\} \\
\eta_\mu & \mu\alpha.[\alpha]M = M \\
\beta_\mu & (\lambda x.\mu\alpha.[\beta]M)\ N = \mu\alpha.[\beta](\lambda x.M)\ N
\end{array}$$

**Figure 11.** The $\Lambda\mu_v$-calculus.

$$\begin{array}{c}
\mathcal{C}[\![[q]M]\!]^M \triangleq \mathcal{C}[\![M]\!]^M\ \mathcal{C}[\![q]\!]^q \\
\mathcal{C}[\![\mu\alpha.M]\!]^M \triangleq \lambda\alpha.\mathcal{C}[\![M]\!]^M \\
\mathcal{C}[\![\tilde{\mu}x.M]\!]^q \triangleq \lambda x.\mathcal{C}[\![M]\!]^M
\end{array}$$

**Figure 12.** CPS transformation $\mathcal{C}$ for extra terms of $\Lambda\mu\tilde{\mu}_v$.

original presentation. In contrast to David and Py's [9] proof that the original $\lambda\mu$-calculus *does not* satisfy Böhm's theorem of separability, Saurin showed that the $\Lambda\mu$-calculus *does*. In other words, there are terms in the $\lambda\mu$-calculus which are equationally distinct but cannot be observably distinguished, whereas in the $\Lambda\mu$-calculus there is a reason for every inequality between terms — some context can tell them apart.[6] This suggests that in the untyped call-by-name setting, there is something missing from the $\lambda\mu$-calculus that is restored by relaxing the syntax.

Here, we would like to ask the same question in the untyped call-by-value setting: is there something missing from the $\lambda\mu_v$-calculus that is restored by a more relaxed syntax? We begin by investigating a call-by-value variant of Saurin's $\Lambda\mu$-calculus. The call-by-value $\Lambda\mu_v$-calculus is shown in Figure 11, where the only change from $\lambda\mu_v$ is that in $\Lambda\mu_v$ terms and commands have been collapsed into the same syntactic type. This opens up the possibility to write new programs, like $\mu\alpha.\mu\beta.M$ and $\lambda x.[\alpha]x$, that were not legal in the $\lambda\mu_v$-calculus. However, it is important to stress that we are not adding anything *new* to the language — the only difference is that we allow the same constructs to be used in new contexts.

Because we are not adding anything new to the language, we don't need to add any new rules. Rather, we only lift the call-by-value equational theory $\lambda\mu_v$ into the relaxed setting by allowing the metavariables for terms and commands to stand in for more expressions, exactly the way we lifted the axioms of $\lambda_c$ into the $\lambda\mu_v$-calculus. More specifically, the axioms of the $\Lambda\mu_v$ equational theory are the $\lambda_c$ axioms plus the additional axioms of control in Figure 11. The $\Lambda\mu_v$ theory is a conservative extension of $\lambda_c$ and $\lambda\mu_v$, where the only change is that the axioms of $\lambda\mu_v$ apply to more terms and in more contexts according to the conflation of terms and commands. For example, we now have the $\beta_v$ equality $(\lambda x.[\alpha]x)\ V = [\alpha]V$ which was invalid in $\lambda\mu_v$, not because it wasn't a legal instance of $\beta_v$, but because $\lambda x.[\alpha]x$ was not a legal term. We also have the $\mu_v$ equality $[\alpha_1][\alpha_2]\mu\beta.M = [\alpha_1]M\,\{\alpha_2/\beta\}$ that occurs inside of a nested command, which could not be expressed in the syntax of $\lambda\mu_v$-calculus.

Likewise, we can apply the call-by-value CPS transformation $\mathcal{C}$ to the $\Lambda\mu_v$-calculus by lifting the transformation unchanged into the relaxed syntax. As before, we relate the CPS transformation to the intermediate language $\Lambda\mu\tilde{\mu}_v$, which extends co-terms with

[6] More formally, for any two closed normal terms $M$ and $N$ that are not equated, there is a context $C$ such that $C[M] = x$ and $C[N] = y$.

$$M ::= \lambda\alpha.M \mid M\ V^q \mid V^q\ V^V \mid V^V\ (V^V, V^q)$$
$$V^q ::= \alpha \mid \lambda x.M \quad V^V ::= x \mid \lambda(x,\alpha).M$$

**Figure 13.** The image of CPS transformation of $\Lambda\mu\tilde{\mu}_v$ by $\mathcal{C}$.

$$\mathcal{C}^{-1}[\![M\ V^q]\!]^M \triangleq [\mathcal{C}^{-1}[\![V^q]\!]^q]\mathcal{C}^{-1}[\![M]\!]^M$$
$$\mathcal{C}^{-1}[\![\lambda\alpha.M]\!]^M \triangleq \mu\alpha.\mathcal{C}^{-1}[\![M]\!]^M$$
$$\mathcal{C}^{-1}[\![\lambda x.M]\!]^q \triangleq \tilde{\mu}x.\mathcal{C}^{-1}[\![M]\!]^M$$
$$\mathcal{C}^{-1}[\![\lambda(x,\alpha).M]\!]^V \triangleq \lambda x.\mu\alpha.\mathcal{C}^{-1}[\![M]\!]^M$$

**Figure 14.** The inverse of $\mathcal{C}$ on the image of $\Lambda\mu\tilde{\mu}_v$.

$\tilde{\mu}$-abstractions, $\tilde{\mu}x.M$ and has the relaxed axioms

$$\mu_q \quad [q]\mu\alpha.M = M\{q/\alpha\} \quad \tilde{\mu}_v \quad [\tilde{\mu}x.M]V = M\{V/x\}$$

in addition to those in Figure 7. The affected clauses of the transformation are shown in Figure 12, where a command may appear in a context previously reserved for terms, and vice versa for terms in contexts expecting commands. Again, the pattern of the CPS transformation is essentially the same. More specifically, $\mathcal{C}$ relates contexts in the $\Lambda\mu_v$-calculus to exactly the same contexts in the CPS $\lambda$-calculus: the transformation on terms still sends the context $\lambda x.\Box$ to $\lambda\alpha.\alpha\ (\lambda(x,\beta).\Box\ \beta)$, and still sends the context $\mu\alpha.\Box$ to $\lambda\alpha.\Box$. This is further evidence we have not changed our interpretation of the $\lambda\mu_v$ constructs, but are only using them in new ways.

It is interesting to observe the impact that the relaxed syntax of $\Lambda\mu_v$ has on the image of the CPS transformation. As seen in Figure 13, the previous distinction between the image of terms and commands is collapsed, as we may expect. But this collapse has opened up the possibilities of new terms in the CPS setting. More specifically, we may now have a sequence of applications, like $(\lambda\alpha.M)\ V_1^q\ V_2^q\ V_3^q$, which did not occur before. This shows us that the relaxed syntax of the $\Lambda\mu_v$-calculus has opened up the availability of more evaluation contexts in the resulting CPS terms, letting us supply a term with as many continuations as we want.

Extending our previous proof of the CPS soundness and completeness of $\lambda\mu\tilde{\mu}_v$ to cover $\Lambda\mu\tilde{\mu}_v$ is straightforward. The extension of the inverse CPS transformation of $\Lambda\mu\tilde{\mu}_v$ is shown in Figure 14. The $\mathcal{C}$ and $\mathcal{C}^{-1}$ transformations are still compositional and remain inverses, and the axioms of $\Lambda\mu\tilde{\mu}_v$ and the image of $\mathcal{C}$ are still mutually inter-derivable.

**Theorem 4.** *The $\Lambda\mu\tilde{\mu}_v$ equational theory is sound and complete with respect to the $\beta\eta$ theory of the $\lambda$-calculus with pairs.*

*Proof.* The $\mathcal{C}$ and $\mathcal{C}^{-1}$ transformations form an equational correspondence between $\Lambda\mu\tilde{\mu}_v$ and the image of $\mathcal{C}$ in the $\lambda$-calculus, as in the proof of Theorem 2. □

Additionally, $\Lambda\mu_v$ is in equational correspondence with $\Lambda\mu\tilde{\mu}_v$ in the same manner as $\lambda\mu_v$ and $\lambda\mu\tilde{\mu}_v$: we may desugar all $\tilde{\mu}$-abstractions using the same process as given in Section 4 for $\lambda\mu\tilde{\mu}_v$. As a corollary, we have soundness and completeness of $\Lambda\mu_v$ with respect to the $\lambda$-calculus with pairs by the $\mathcal{C}$ transformation.

**Theorem 5.** *$\Lambda\mu_v$ is in equational correspondence with $\Lambda\mu\tilde{\mu}_v$.*

*Proof.* The same as for Theorem 3, lifted into $\Lambda\mu\tilde{\mu}_v$. □

$$\eta_{\widehat{\mathsf{tp}}} \qquad \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]V = V$$
$$\mu_{\widehat{\mathsf{tp}}} \qquad [\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.c = c$$
$$\beta_{\widehat{\mathsf{tp}}} \qquad (\lambda x.\mu\widehat{\mathsf{tp}}.[q]M)\ (\mu\widehat{\mathsf{tp}}.c) = \mu\widehat{\mathsf{tp}}.[q](\lambda x.M)\ (\mu\widehat{\mathsf{tp}}.c)$$

**Figure 15.** The axioms of $\widehat{\mathsf{tp}}$ in the $\lambda\mu\widehat{\mathsf{tp}}$ equational theory.

## 6. Shift and delimited control

So far, we have investigated the foundations of classical control — manipulations of control flow that are expressible by operators like callcc or Felleisen's [14] $\mathcal{C}$ operator. There is also another form of control effect referred to as *delimited* control. Intuitively, delimited control operators are different from callcc in two primary ways: (1) the evaluation context that is seen by the control operator is scoped by a *delimiter*, so that only a partial snapshot is taken of the program's control state, and (2) the continuation that is produced by the control operator does not jump somewhere else when called, but instead *returns* a result to its calling context. Because the continuations that come out of delimited control actually return something useful, so that the output of one may be fed to another, they are sometimes referred to as *composable continuations*. This additional compositional power — that we may isolate and run an effectful computation to see how it behaves, and that we may connect multiple continuations together — is essential for the expressive capabilities of delimited control.

Upon Saurin's [31] discovery of the observational completeness of $\Lambda\mu$ in the untyped call-by-name setting, Herbelin and Ghilezan [17] observed that the seemingly innocuous move from $\lambda\mu$ to $\Lambda\mu$ has a profound impact on the expressive power of the language. Whereas $\lambda\mu$ is a calculus for call-by-name classical control, $\Lambda\mu$ is a calculus for call-by-name *delimited* control. Indeed, even de Groote [10] and Ong and Stewart's [24] original analysis in call-by-name and call-by-value, respectively, relates typed $\Lambda\mu$-calculi to variants of Felleisen's [14] theory of control with a composable form of continuations that do not abort like in Felleisen's theory. Furthermore, Fujita [13] analyzes an alternative call-by-value theory for $\Lambda\mu$ with a CPS transformation that actually composes continuations much like the *continuation-composing style* transformation of the shift operator [8]. In essence, the type systems prevent programs from taking advantage of the extra expressibility that is latently present in the untyped calculi.

In his seminal work, Filinski [15] showed a different notion of completeness for delimited control in call-by-value functional languages: every computational effect that can be encoded as a monad can be directly expressed by delimited control operators. This property does not hold for classical control — for example, neither callcc nor $\mathcal{C}$ are able to express mutable references. So we see similar situations in call-by-name and call-by-value, where a delimited form of control is more complete than classical control. We have already seen some indications of delimited control hiding in callcc and $\lambda\mu_v$: a jump, $\alpha\ M$, marks the end of a usable context and a command, $[\alpha]M$, encapsulates a control effect. We will now see how the additional programs available in $\Lambda\mu_v$ let us express various forms of delimited control, and look for which one corresponds to the full $\Lambda\mu_v$-calculus.

### 6.1 The $\lambda\mu\widehat{\mathsf{tp}}$-calculus

An obvious place to begin the search, since it is also based on Parigot's $\lambda\mu$-calculus, is with the $\lambda\mu\widehat{\mathsf{tp}}$-calculus of Ariola *et al.* [4]. The $\lambda\mu\widehat{\mathsf{tp}}$-calculus is a call-by-value language that extends $\lambda\mu_v$ with a single *dynamically bound* co-variable named $\widehat{\mathsf{tp}}$, so that we have the new term $\mu\widehat{\mathsf{tp}}.c$ and new command $[\widehat{\mathsf{tp}}]M$. The intuition of the dynamic nature of $\widehat{\mathsf{tp}}$ is that it corresponds to the dynamic

$$
\begin{aligned}
V \in Value &::= x \mid \lambda x.M \\
M, N \in Term &::= V \mid M\ N \mid \mu\alpha.c \\
c \in Command &::= [q]M \mid V \mid (\lambda x.c')\ c \\
q \in CoTerm &::= \alpha \mid \tilde{\mu}x.x
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{TP}^{-1}[\![\widehat{\mathsf{tp}}]\!] &\triangleq \tilde{\mu}x.x \\
\mathcal{TP}^{-1}[\![\mu\widehat{\mathsf{tp}}.c]\!] &\triangleq \mu\gamma.(\lambda x.[\gamma]x)\ \mathcal{TP}^{-1}[\![c]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{TP}[\![\tilde{\mu}x.x]\!]^q &\triangleq \widehat{\mathsf{tp}} \\
\mathcal{TP}[\![V]\!]^c &\triangleq [\widehat{\mathsf{tp}}]\mathcal{TP}[\![V]\!]^V \\
\mathcal{TP}[\![(\lambda x.c')\ c]\!]^c &\triangleq [\widehat{\mathsf{tp}}](\lambda x.\mu\widehat{\mathsf{tp}}.\mathcal{TP}[\![c']\!])\ (\mu\widehat{\mathsf{tp}}.\mathcal{TP}[\![c]\!])
\end{aligned}
$$

**Figure 16.** The embedding $\mathcal{TP}^{-1}$ of $\lambda\mu\widehat{\mathsf{tp}}$ into $\Lambda\mu_v$, the image of $\mathcal{TP}^{-1}$ in $\Lambda\mu_v$, and the translation back into $\lambda\mu\widehat{\mathsf{tp}}$.

scope of exception handling — the chosen handler is based on the call stack at run-time rather than lexical scope — and it is used to delimit the scope of $\mu$-abstractions. Herbelin and Ghilezan [17] introduce an equational theory that is sound and complete with respect to the CPS transformation of $\lambda\mu\widehat{\mathsf{tp}}$, which includes all the axioms of $\lambda\mu_v$ along with the axioms of $\widehat{\mathsf{tp}}$ shown in Figure 15. Additionally, any of the $\lambda\mu_v$ axioms that mention a command of the form $[\alpha]M$ is extended to $[q]M$, where $q$ is either $\alpha$ or $\widehat{\mathsf{tp}}$. Note that since $\widehat{\mathsf{tp}}$ is a dynamic variable, the $\eta_{\widehat{\mathsf{tp}}}$, $\mu_{\widehat{\mathsf{tp}}}$, and extended $\mu_v$ axioms are allowed to "capture" the $\widehat{\mathsf{tp}}$ variable which can happen when $\widehat{\mathsf{tp}}$ occurs "free" in $V$ or $c$. Also notice that, unlike the usual situation, the $\eta_{\widehat{\mathsf{tp}}}$ rule is considered operational, allowing us to simplify terms like $\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]5 = 5$. The $\mu_{\widehat{\mathsf{tp}}}$ rule, on the other hand, is purely an observational property that is never needed to evaluate a term in the $\lambda\mu\widehat{\mathsf{tp}}$-calculus.

The $\Lambda\mu_v$-calculus is large enough to express all of $\lambda\mu\widehat{\mathsf{tp}}$ without the use of dynamically scoped co-variables. For convenience, we will rely on $\tilde{\mu}$-abstractions as useful syntactic sugar (recall from Section 5 that $\Lambda\mu\tilde{\mu}_v$ and $\Lambda\mu_v$ are equivalent). One way to develop the embedding from $\lambda\mu\widehat{\mathsf{tp}}$ to $\Lambda\mu_v$ is to factor through the CPS transformations by first writing $\lambda\mu\widehat{\mathsf{tp}}$ in continuation-passing style, and then translating that term into the $\Lambda\mu_v$-calculus. The use of the dynamic co-variable $\widehat{\mathsf{tp}}$ in a command like $[\widehat{\mathsf{tp}}]M$, after some simplifications, is embedded as:

$$\mathcal{C}^{-1}[\![\mathcal{C}[\![\widehat{\mathsf{tp}}]\!]^q]\!]^q \triangleq \mathcal{C}^{-1}[\![\lambda x.\lambda\gamma.\gamma\ x]\!]^q =_{\eta_\mu} \tilde{\mu}x.x$$

That is to say, $\widehat{\mathsf{tp}}$ can be thought of as a (closed) co-term that, when given a value, just returns that value up to whoever is listening for the output of the command. Binding $\widehat{\mathsf{tp}}$ in a term like $\mu\widehat{\mathsf{tp}}.c$ is embedded as:

$$
\begin{aligned}
\mathcal{C}^{-1}[\![\mathcal{C}[\![\mu\widehat{\mathsf{tp}}.c]\!]^M]\!]^M &\triangleq \mathcal{C}^{-1}[\![\lambda\alpha.\lambda\gamma.\mathcal{C}[\![c]\!]^c\ \lambda x.\alpha\ x\ \gamma]\!]^M \\
&\triangleq \mu\alpha.\mu\gamma.[\tilde{\mu}x.[\gamma][\alpha]x]\mathcal{C}^{-1}[\![\mathcal{C}[\![c]\!]^c]\!]^M \\
&=_{\beta_v,\varsigma_v} \mu\alpha.(\lambda x.[\alpha]x)\ (\mathcal{C}^{-1}[\![\mathcal{C}[\![c]\!]^c]\!]^M)
\end{aligned}
$$

Thus, we can look at $\mu\widehat{\mathsf{tp}}.c$ in one of two ways. On the one hand, binding $\widehat{\mathsf{tp}}$ grabs the nearest two continuations (the second one, $\gamma$, is generally referred to as the *meta-continuation*) and then runs the command $c$ in a meta-continuation made by composing the two. On the other hand, binding $\widehat{\mathsf{tp}}$ grabs the nearest continuation $\alpha$, wraps it in a function, calls the function with $c$ as an argument, and passes the value returned by $c$ along to $\alpha$. The embedding, $\mathcal{TP}^{-1}$, that translates the dynamic $\widehat{\mathsf{tp}}$ into $\Lambda\mu_v$ (using $\tilde{\mu}$-abstractions as syntactic sugar) is shown in Figure 16.

*Example* 4. Using the translation $\mathcal{TP}^{-1}$ into $\Lambda\mu_v$, we have an alternate interpretation of the dynamic $\widehat{\mathsf{tp}}$ co-variable that steps outside of the syntactic restrictions of $\lambda\mu\widehat{\mathsf{tp}}$. For example, the $\eta_{\widehat{\mathsf{tp}}}$ reduction of $\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]z$ to $z$ can be done in two separate steps. First, we concentrate on the command $[\widehat{\mathsf{tp}}]z$, and find out that it simplifies to the value $z$:

$$\mathcal{TP}^{-1}[\![[\widehat{\mathsf{tp}}]z]\!] \triangleq [\tilde{\mu}x.x]z =_{\tilde{\mu}_v} z$$

Therefore, the command $[\widehat{\mathsf{tp}}]z$ propagates $z$ upward to give us $\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]z = \mu\widehat{\mathsf{tp}}.z$. Next, the binding of $\widehat{\mathsf{tp}}$, when given the value $z$ as its body, also propagates $z$ upward:

$$\mathcal{TP}^{-1}[\![\mu\widehat{\mathsf{tp}}.z]\!] \triangleq \mu\gamma.(\lambda x.[\gamma]x)\ z =_{\beta_v} \mu\gamma.[\gamma]z =_{\eta_\mu} z$$

This lets us break down the reduction of $\widehat{\mathsf{tp}}$ into the two steps that propagate values upward: $\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]z = \mu\widehat{\mathsf{tp}}.z = z$.

We can also explain the dynamic nature of $\widehat{\mathsf{tp}}$ by the fact that it signifies a *closed* co-term in the $\Lambda\mu_v$-calculus. For example, consider the following command which "captures" $\widehat{\mathsf{tp}}$ in terms of its embedding into $\Lambda\mu_v$:

$$
\begin{aligned}
&\mathcal{TP}^{-1}[\![[\widehat{\mathsf{tp}}]\mu\alpha.[\widehat{\mathsf{tp}}](\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x)]\!] \\
&\triangleq [\tilde{\mu}y.y]\mu\alpha.[\tilde{\mu}z.z](\lambda x.\mu\gamma.(\lambda w.[\gamma]w)\ ([\alpha]x)) \\
&=_{\mu_v} [\tilde{\mu}z.z](\lambda x.\mu\gamma.(\lambda w.[\gamma]w)\ ([\tilde{\mu}y.y]x)) \\
&\triangleq \mathcal{TP}^{-1}[\![[\widehat{\mathsf{tp}}](\lambda x.\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]x)]\!]
\end{aligned}
$$

The $\Lambda\mu_v$-calculus has no concept of dynamic variables, and yet the reduction gives the appearance of dynamic capture when viewed in the $\lambda\mu\widehat{\mathsf{tp}}$-calculus.

Finally, instead of interpreting $\mu\widehat{\mathsf{tp}}.c$ as a passive form that forces us to evaluate the underlying command $c$, we can give it a more active interpretation that re-arranges its evaluation context. For example, when we see the term $\mu\widehat{\mathsf{tp}}.[\beta]z$ in an evaluation context, we can use the call-by-value order of function calls to specify that $[\beta]z$ is to be evaluated first:

$$
\begin{aligned}
\mathcal{TP}^{-1}[\![[\alpha]((\mu\widehat{\mathsf{tp}}.[\beta]z) + 10)]\!] &\triangleq [\alpha]((\mu\gamma.(\lambda x.[\gamma]x)\ ([\beta]z)) + 10) \\
&=_{\mu_v} (\lambda x.[\alpha](x + 10))\ ([\beta]z)
\end{aligned}
$$

In general, the embedding of $\lambda\mu\widehat{\mathsf{tp}}$ into $\Lambda\mu_v$ gives us some derived equalities that are allowed by the more relaxed syntax:

$$
\begin{aligned}
[\widehat{\mathsf{tp}}]V &= V & \mu\widehat{\mathsf{tp}}.V &= V \\
[q]E[\mu\widehat{\mathsf{tp}}.c] &= (\lambda x.[q]E[x])\ c & &\textit{End example 4.}
\end{aligned}
$$

As it turns out, the embedding of $\lambda\mu\widehat{\mathsf{tp}}$ does not cover the entire $\Lambda\mu_v$-calculus — there are some unreachable terms. However, we can still consider the image of $\lambda\mu\widehat{\mathsf{tp}}$ inside of $\Lambda\mu_v$, as shown in Figure 16, and give the correspondence between these two. Intuitively, the essential addition that $\widehat{\mathsf{tp}}$ gives us is the closed identity co-term, $\tilde{\mu}x.x$, and the ability to run a command and observe its result, $(\lambda x.c')\ c$.

**Theorem 6.** *The $\lambda\mu\widehat{\mathsf{tp}}$ equational theory is sound and complete with respect to the $\Lambda\mu_v$ equational theory.*

*Proof.* The compositional transformations $\mathcal{TP}$ and $\mathcal{TP}^{-1}$ form an equational correspondence between $\lambda\mu\widehat{\mathsf{tp}}$ and its image in $\Lambda\mu_v$. □

### 6.2 Shift and reset

One of the most well-studied presentations of delimited control is given by Danvy and Filinski's [7] shift and reset control operators. These operators were used by Filinski [15] to encode a direct representation of monadic effects, and have been given a sound and complete axiomatization by Kameyama and Hasegawa [19] with respect to their CPS transformations. Furthermore, the shift and

$$
\begin{array}{rc}
\mathsf{reset}_{value} & \langle V \rangle = V \\
\mathsf{reset}_{lift} & (\lambda x.\langle M \rangle)\ \langle N \rangle = \langle (\lambda x.M)\ \langle N \rangle \rangle \\
\mathcal{S}_{elim} & \mathcal{S}(\lambda\alpha.\alpha\ M) = M \\
\mathsf{reset}\,\mathcal{S} & \langle E[\mathcal{S}M] \rangle = \langle M\ (\lambda x.\langle E[x] \rangle) \rangle \\
\mathcal{S}\,\mathsf{reset} & \mathcal{S}(\lambda\alpha.\langle M \rangle) = \mathcal{S}(\lambda\alpha.M) \\
\mathcal{S}_{pure} & \langle \alpha\ V \rangle = \alpha\ V
\end{array}
$$

**Figure 17.** The axioms of shift ($\mathcal{S}$) and reset ($\langle M \rangle$) in $\lambda_{\mathcal{S}\alpha}$.

$$
\begin{array}{rl}
\mathcal{SR}[\![\mu\alpha.c]\!] & \triangleq \mathcal{S}(\lambda\alpha.\mathcal{SR}[\![c]\!]) \\
\mathcal{SR}[\![\mu\widehat{\mathsf{tp}}.c]\!] & \triangleq \mathcal{SR}[\![c]\!] \\
\mathcal{SR}[\![[q]M]\!] & \triangleq \langle \mathcal{SR}[\![q]\!]\ \mathcal{SR}[\![M]\!] \rangle \\
\mathcal{SR}[\![\alpha]\!] & \triangleq \alpha \\
\mathcal{SR}[\![\widehat{\mathsf{tp}}]\!] & \triangleq \lambda x.x \\
\\
\mathcal{SR}^{-1}[\![\mathcal{S}]\!] & \triangleq \lambda h.\mu\alpha.[\widehat{\mathsf{tp}}]h\ (\lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x) \\
\mathcal{SR}^{-1}[\![\mathcal{S}(\lambda\alpha.M)]\!] & \triangleq \mu\alpha.[\widehat{\mathsf{tp}}]\mathcal{SR}^{-1}[\![M]\!] \\
\mathcal{SR}^{-1}[\![\langle M \rangle]\!] & \triangleq \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mathcal{SR}^{-1}[\![M]\!] \\
\mathcal{SR}^{-1}[\![\alpha]\!] & \triangleq \lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x
\end{array}
$$

**Figure 18.** The isomorphism between $\lambda_{\mathcal{S}\alpha}$ and $\lambda\mu\widehat{\mathsf{tp}}$.

reset operators, with Kameyama and Hasegawa's axiomatization, have been shown to correspond to the $\lambda\mu\widehat{\mathsf{tp}}$-calculus [4, 17]. Since the $\lambda\mu\widehat{\mathsf{tp}}$-calculus expresses only a subset of $\Lambda\mu_v$, it means that shift and reset is not the form of delimited control that corresponds to the full $\Lambda\mu_v$-calculus. However, for the sake of completeness, we still include shift and reset in our analysis.

Recall that for $\lambda_c$ callcc, we gave a special status to continuation variables $\alpha$ that are introduced by a call to callcc, as in $\mathsf{callcc}(\lambda\alpha.M)$, in the style of Sabry [28]. These special continuation variables correspond to a particular use of co-variables in the $\lambda\mu$-calculus, which allowed us to "abort" the current evaluation context when calling $\alpha$. Although Kameyama and Hasegawa [19] do not consider continuation variables with special properties, they greatly ease local reasoning about open terms without explicit mention of their closing context. If we do, we may add a special rule for continuation variables which must be introduced by a shift, as in $\mathcal{S}(\lambda\alpha.M)$. In contrast to callcc, which creates functions that never return, the functions created by shift *always* return because they introduce a reset when called. Therefore, we may give the additional axiom that asserts that $\alpha$ is always a *pure* function that returns to its calling context, so a surrounding reset is unnecessary

$$\mathcal{S}_{pure} \qquad \langle \alpha\ V \rangle = \alpha\ V$$

which we add to Kameyama and Hasegawa's $\lambda_{\mathcal{S}}$-calculus to get the $\lambda_{\mathcal{S}\alpha}$-calculus shown in Figure 17. This axiom is supported by the embedding of continuation variables into $\lambda\mu\widehat{\mathsf{tp}}$ that reflects exactly the way that shift wraps up a continuation.

We strengthen the result of Herbelin and Ghilezan [17] and relate shift and reset to $\Lambda\mu_v$ through the $\lambda\mu\widehat{\mathsf{tp}}$-calculus. In particular, Kameyama and Hasegawa's [19] $\lambda_{\mathcal{S}}$ theory of shift and reset is sound and complete with respect to $\lambda_{\mathcal{S}\alpha}$, which is in equational correspondence with $\lambda\mu\widehat{\mathsf{tp}}$ by the compositional transformations in Figure 18. As a corollary, $\lambda_{\mathcal{S}}$ is sound and complete with respect to the $\Lambda\mu_v$-calculus.

**Theorem 7.** *The $\lambda_{\mathcal{S}}$ equational theory is sound and complete with respect to the $\lambda_{\mathcal{S}\alpha}$ equational theory, and $\lambda_{\mathcal{S}\alpha}$ is in equational correspondence with $\lambda\mu\widehat{\mathsf{tp}}$.*

*Proof.* There is a direct injection of $\lambda_{\mathcal{S}}$ into $\lambda_{\mathcal{S}\alpha}$ as well as a reverse embedding that forgets the special status of continuation variables. These compositional transformations form an equational correspondence between $\lambda_{\mathcal{S}}$ and the subset of $\lambda_{\mathcal{S}\alpha}$ terms that do not contain free occurrences of continuation variables. Furthermore, the compositional $\mathcal{SR}$ and $\mathcal{SR}^{-1}$ transformations form an equational correspondence between $\lambda_{\mathcal{S}\alpha}$ and $\lambda\mu\widehat{\mathsf{tp}}$. □

*Example* 5. Let us consider how continuations compose using shift and reset, $\lambda\mu\widehat{\mathsf{tp}}$, and $\Lambda\mu_v$, and how the reduction is simulated in each of the three calculi. For example, the term $\langle \mathcal{S}\ (\lambda k.k\ (k\ 2)) + 10 \rangle$ captures the evaluation context $\square + 10$ and applies it twice to the number 2, giving us 22 as the result. In terms of the $\lambda_{\mathcal{S}}$-calculus, we have the following reduction:

$$
\begin{array}{rl}
\langle \mathcal{S}\ (\lambda k.k\ (k\ 2)) + 10 \rangle & =_{\mathsf{reset}\,\mathcal{S}}\ \langle (\lambda k.k\ (k\ 2))\ (\lambda x.\langle x + 10 \rangle) \rangle \\
& =_{\beta_v}\ \langle (\lambda x.\langle x + 10 \rangle)\ ((\lambda x.\langle x + 10 \rangle)\ 2) \rangle \\
& =_{\beta_v}\ \langle (\lambda x.\langle x + 10 \rangle)\ \langle 2 + 10 \rangle \rangle \\
& =_{\mathsf{reset}_{value}}\ \langle (\lambda x.\langle x + 10 \rangle)\ 12 \rangle \\
& =_{\beta_v}\ \langle \langle 12 + 10 \rangle \rangle \\
& =_{\mathsf{reset}_{value}}\ 22
\end{array}
$$

By translating the shift and reset operators into $\lambda\mu\widehat{\mathsf{tp}}$ using the $\mathcal{SR}^{-1}$ embedding, we get the simplified term

$$
\begin{array}{l}
\mathcal{SR}^{-1}[\![\langle \mathcal{S}\ (\lambda\alpha.\alpha\ (\alpha\ 2)) + 10 \rangle]\!] \\
\quad = \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]((\mu\alpha.[\alpha](\mu\widehat{\mathsf{tp}}.[\alpha]2)) + 10)
\end{array}
$$

which also produces the answer 22:

$$
\begin{array}{l}
\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]((\mu\alpha.[\alpha](\mu\widehat{\mathsf{tp}}.[\alpha]2)) + 10) \\
=_{\mu_v}\ \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]((\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}](2 + 10)) + 10) \\
=_{\eta_{\widehat{\mathsf{tp}}}}\ \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}](12 + 10) \\
=_{\eta_{\widehat{\mathsf{tp}}}}\ 22
\end{array}
$$

Again, we can translate the term into $\Lambda\mu_v$ using the $\mathcal{TP}^{-1}$ embedding to get the simplified term

$$
\begin{array}{l}
\mathcal{TP}^{-1}[\![\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]((\mu\alpha.[\alpha](\mu\widehat{\mathsf{tp}}.[\alpha]2)) + 10)]\!] \\
\quad =_{\mu_v}\ \mu\gamma.(\lambda x.[\gamma]x)\ ([\tilde{\mu}y.y]((\mu\alpha.(\lambda z.[\alpha]z)\ ([\alpha]2)) + 10))
\end{array}
$$

which produces the same result:

$$
\begin{array}{l}
\mu\gamma.(\lambda x.[\gamma]x)\ ([\tilde{\mu}y.y]((\mu\alpha.(\lambda z.[\alpha]z)\ ([\alpha]2)) + 10)) \\
=_{\mu_v}\ \mu\gamma.(\lambda x.[\gamma]x)\ ((\lambda z.[\tilde{\mu}y.y](z + 10))\ ([\tilde{\mu}y.y](2 + 10))) \\
=_{\tilde{\mu}_v,\beta_v}\ \mu\gamma.(\lambda x.[\gamma]x)\ ([\tilde{\mu}y.y](12 + 10)) \\
=_{\tilde{\mu}_v,\beta_v}\ \mu\gamma.[\gamma]22 \\
=_{\eta_\mu}\ 22 \qquad \textit{End example 5.}
\end{array}
$$

Observe that, like with callcc, the embeddings into $\lambda\mu\widehat{\mathsf{tp}}$ and $\Lambda\mu_v$ give the expected CPS transformation of shift and reset, up to currying. In particular, after the embeddings we attain a two-pass CPS transformation [8, 19] for shift and reset:

$$
\begin{array}{l}
\mathcal{C}[\![\mathcal{TP}^{-1}[\![\mathcal{SR}^{-1}[\![\mathcal{S}]\!]]\!]]\!]^V \\
\quad = \lambda(h, \alpha).h\ ((\lambda(x, \beta).\lambda\gamma.\alpha\ x\ (\lambda y.\beta\ y\ \gamma)), (\lambda z.\lambda\gamma'.\gamma'\ z)) \\
\mathcal{C}[\![\mathcal{TP}^{-1}[\![\mathcal{SR}^{-1}[\![\langle M \rangle]\!]]\!]]\!] \\
\quad = \lambda\alpha.\lambda\gamma.[\![M]\!]\ (\lambda x.\lambda\gamma'.\gamma'\ x)\ (\lambda y.\alpha\ y\ \gamma)
\end{array}
$$

*Remark* 1. Recall that in Section 5, we noted that the original typed interpretation of the de Groote [10]-style $\Lambda\mu$-calculus effectively

$$
\begin{array}{ll}
\beta_{\$} & V\$\mathcal{S}_0(\lambda\alpha.M) = M\,\{V/\alpha\} \\
\eta_{\$} & \mathcal{S}_0(\lambda\alpha.\alpha\$M) = M \\
\$_v & V\$V' = V\ V' \\
\$_E & V\$E[M] = (\lambda x.V\$E[x])\$M
\end{array}
$$

**Figure 19.** The axioms of $\mathsf{shift}_0$ ($\mathcal{S}_0$) and \$ in $\lambda_{\$}$.

$$
\begin{array}{rcl}
\mathcal{Z}[\![\mu\alpha.M]\!] & \triangleq & \mathcal{S}_0(\lambda\alpha.\mathcal{Z}[\![M]\!]) \\
\mathcal{Z}[\![[q]M]\!] & \triangleq & \mathcal{Z}[\![q]\!]\$\mathcal{Z}[\![M]\!] \\
\mathcal{Z}[\![\alpha]\!] & \triangleq & \alpha \\
\mathcal{Z}[\![\tilde{\mu}x.M]\!] & \triangleq & \lambda x.\mathcal{Z}[\![M]\!] \\
\\
\mathcal{Z}^{-1}[\![\mathcal{S}_0]\!] & \triangleq & \lambda h.\mu\alpha.h\ (\lambda x.[\alpha]x) \\
\mathcal{Z}^{-1}[\![\mathcal{S}_0(\lambda\alpha.M)]\!] & \triangleq & \mu\alpha.\mathcal{Z}^{-1}[\![M]\!] \\
\mathcal{Z}^{-1}[\![N\$M]\!] & \triangleq & (\lambda k.[\tilde{\mu}x.k\ x]\mathcal{Z}^{-1}[\![M]\!])\ \mathcal{Z}^{-1}[\![N]\!] \\
\mathcal{Z}^{-1}[\![\alpha]\!] & \triangleq & \lambda x.[\alpha]x
\end{array}
$$

**Figure 20.** The isomorphism between $\lambda_{\$}$ and $\Lambda\mu\tilde{\mu}_v$.

considers commands to have the type $\bot$. This typing would force $x$ in both $[\tilde{\mu}x.x]M$ and $(\lambda x.c')\ c$ to always have the type $\bot$. Applying this typing regime to the $\lambda\mu\widehat{\mathsf{tp}}$-calculus, we would only allow $[\widehat{\mathsf{tp}}]M$ when $M$ has type $\bot$, and likewise $\mu\widehat{\mathsf{tp}}.c$ would have type $\bot$. In other words, this type system severely restricts the new $\widehat{\mathsf{tp}}$ variable as there is no way to use it with interesting types like integers or booleans. This restriction gives an intuitive reason why de Groote's typing forces the $\Lambda\mu$-calculus to be equivalent in expressive power to Parigot's classical $\lambda\mu$-calculus, by enforcing the restriction in the types rather than in the syntax.

From another perspective, by embedding shift and reset into the $\Lambda\mu_v$-calculus, de Groote's typing would only allow $\langle M\rangle$ when $M$ has the type $\bot$. Therefore, the classical type system restricts the use of the delimiter to terms that cannot return a result because they belong to a type with no values. However, there are extensions of the basic, classical type system for $\lambda\mu\widehat{\mathsf{tp}}$ [4] and $\lambda_{\mathcal{S}}$ [7] that allow for delimiters to return values with various types. In the approach taken here, the untyped semantics of the $\Lambda\mu_v$-calculus is already expressive enough to represent the dynamic behavior of delimited control. The difference between classical and delimited control is then a matter of choosing a less expressive, classical type system — like de Groote [10], Ong and Stewart [24], or Fujita [13] — or a more expressive, "delimited" one that allows programs like $([\tilde{\mu}x.x]1) + 2$ to be well-typed. *End remark* 1.

## 7. Correspondence of $\Lambda\mu_v$ and shift0

Having considered languages that only touch on a subset of the $\Lambda\mu_v$-calculus, we may wonder what language constructs express the full calculus. Next, we consider two other operators, now known as $\mathsf{shift}_0$ and $\mathsf{reset}_0$, that were introduced as variants of shift and reset by Danvy and Filinski [7], but have garnered relatively less attention. More recently, Materzok and Biernacki [20, 21] have investigated the static and dynamic semantics of $\mathsf{shift}_0$ and $\mathsf{reset}_0$. Intuitively, the difference between shift and $\mathsf{shift}_0$ is that when shift captures its calling context, the surrounding reset delimiter is left in place, whereas when $\mathsf{shift}_0$ captures its calling context the surrounding $\mathsf{reset}_0$ delimiter is removed. For example, we have different equations for the two different operators:

$$
\begin{array}{c}
\langle f\ (\mathcal{S}\ (\lambda k.M))\rangle = \langle(\lambda k.M)\ (\lambda x.\langle f\ x\rangle)\rangle \\
\langle f\ (\mathcal{S}_0\ (\lambda k.M))\rangle_0 = (\lambda k.M)\ (\lambda x.\langle f\ x\rangle_0)
\end{array}
$$

This seemingly minor alteration on the surrounding context makes all the difference. Indeed, Materzok [22] shows that $\mathsf{shift}_0$ and $\mathsf{reset}_0$ are not only powerful enough to encode shift and reset, they are able to represent an arbitrary hierarchy of nested shift and reset operators [8] with a dynamically growing and shrinking stack of continuations.

*Example* 6. We can view the difference between the shift and $\mathsf{shift}_0$ operators by the influence they have over their delimited evaluation contexts. For example, the term $\mathcal{S}\ (\lambda k_1.\mathcal{S}\ (\lambda k_2.k_1\ (k_2\ 1)))$ appears to swap two surrounding evaluation contexts by capturing them locally as $k_1$ and $k_2$, respectively, and then applying them in the reverse order so that the nearest context $k_1$ sees the result of evaluating $k_2$ with 1. However, because shift leaves its surrounding reset delimiter intact, the second call to shift can only see the empty context. Therefore, the second shift effectively does nothing productive, and the net effect of the entire term is to yield 1. For example, if we evaluate the term using the $\lambda_{\mathcal{S}}$ theory in the context $\langle\langle\Box \times 2\rangle + 10\rangle$, where the first evaluation context doubles its input and the second adds 10, then we get the same result as the simple numeric expression $(1 \times 2) + 10$:

$$
\begin{array}{l}
\langle\langle(\mathcal{S}\ (\lambda k_1.\mathcal{S}\ (\lambda k_2.k_1\ (k_2\ 1)))) \times 2\rangle + 10\rangle \\
=_{\mathsf{reset}\,\mathcal{S},\beta_v}\ \langle\langle\mathcal{S}\ (\lambda k_2.(\lambda y.\langle y \times 2\rangle)\ (k_2\ 1))\rangle + 10\rangle \\
=_{\mathsf{reset}\,\mathcal{S},\beta_v}\ \langle\langle(\lambda y.\langle y \times 2\rangle)\ ((\lambda x.\langle x\rangle)\ 1)\rangle + 10\rangle \\
=_{\beta_v}\ \langle\langle(\lambda y.\langle y \times 2\rangle)\ \langle 1\rangle\rangle + 10\rangle \\
=_{\beta_v,\mathsf{reset}_{value}}\ \langle\langle\langle 1 \times 2\rangle\rangle + 10\rangle \\
=_{\mathsf{reset}_{value}}\ 12
\end{array}
$$

Alternatively, consider the same term implemented with $\mathsf{shift}_0$ in place of shift. The $\mathsf{shift}_0$ operator removes its surrounding $\mathsf{reset}_0$ delimiter after it is done, allowing the second call to $\mathsf{shift}_0$ to capture some surrounding, possibly non-empty, evaluation context. Therefore, the net effect of the term is to swap the two nearest evaluation contexts, separated by $\mathsf{reset}_0$, before yielding 1. Likewise, if we evaluate the term in the similar context $\langle\langle\Box \times 2\rangle_0 + 10\rangle_0$ using Materzok's [20] $\lambda_{\mathcal{S}_0}$ theory for $\mathsf{shift}_0$ and $\mathsf{reset}_0$, we see that the program returns the result of $(1 + 10) \times 2$:

$$
\begin{array}{l}
\langle\langle(\mathcal{S}_0\ (\lambda k_1.\mathcal{S}_0\ (\lambda k_2.k_1\ (k_2\ 1)))) \times 2\rangle_0 + 10\rangle_0 \\
=_{\langle\mathcal{S}_0\rangle,\beta_v}\ \langle(\mathcal{S}_0\ (\lambda k_2.(\lambda y.\langle y \times 2\rangle_0)\ (k_2\ 1))) + 10\rangle_0 \\
=_{\langle\mathcal{S}_0\rangle,\beta_v}\ (\lambda y.\langle y \times 2\rangle_0)\ ((\lambda x.\langle x + 10\rangle_0)\ 1) \\
=_{\beta_v}\ (\lambda y.\langle y \times 2\rangle_0)\ \langle 1 + 10\rangle_0 \\
=_{\langle v\rangle,\beta_v}\ \langle 11 \times 2\rangle_0 \\
=_{\langle v\rangle}\ 22
\end{array}
$$

*End example* 6.

Materzok's [20] $\lambda_{\mathcal{S}_0}$ theory of $\mathsf{shift}_0$ and $\mathsf{reset}_0$ was developed in terms of another call-by-value $\lambda$-calculus which includes $\mathsf{shift}_0$ and the delimiting form $M\$N$.[7] Intuitively, the term $V\$M$ surrounds $M$ with a delimiter represented by $V$, where $V$ is a function that describes how to resume if $M$ evaluates to a value. If $M$ is instead a call to $\mathsf{shift}_0$, then $V$ is captured along with the delimiter. The $\mathsf{reset}_0$ delimiter comes out as the special case when we have $(\lambda x.x)\$M$, so that the delimiter immediately returns the value it's given. The axioms of the $\lambda_{\$}$-calculus are $\beta_v$, $\eta_v$ and the additional axioms shown in Figure 19. The rules for $\lambda_{\$}$ already look quite similar to the rules for $\Lambda\mu\tilde{\mu}_v$, and in fact the two correspond. We give translations between $\Lambda\mu\tilde{\mu}_v$ and $\lambda_{\$}$ in Figure 20 which gives the exact correspondence between the two notions of delimited control.

[7] Additionally, $E\$M$ becomes another form of evaluation context.

$$M, N \in Term ::= x \mid \lambda x.M \mid M\ N \mid \mu\alpha.M \mid [\alpha]M$$
$$E \in EvalCxt ::= \Box \mid E\ M$$

$$\begin{array}{rc} \beta_n & (\lambda x.M)\ N = M\{N/x\} \\ \eta_n & \lambda x.M\ x = M \\ \mu_n & [\alpha]E[\mu\beta.M] = M\{[\alpha]E[N]/[\beta]N\} \\ \eta_\mu & \mu\alpha.[\alpha]M = M \end{array}$$

**Figure 21.** The call-by-name $\Lambda\mu$-calculus.

In essence, the $V\$M$ delimiter corresponds to a command in $\Lambda\mu_v$ where $V$ is converted into a co-term (*i.e.,* a continuation).

**Theorem 8.** *$\lambda_\$$ is in equational correspondence with $\Lambda\mu\tilde{\mu}_v$.*

*Proof.* The compositional $\mathcal{Z}$ and $\mathcal{Z}^{-1}$ transformations form an equational correspondence between $\lambda_\$$ and $\Lambda\mu\tilde{\mu}_v$. □

*Remark* 2. To illustrate the difference between shift/reset and $\mathsf{shift}_0$/$\mathsf{reset}_0$, we point out that $\mathsf{reset}_0$ may be defined in the following equivalent ways:

$$\langle M\rangle_0 \triangleq (\lambda x.x)\$M \triangleq [\tilde{\mu}x.x]M \triangleq [\widehat{\mathsf{tp}}]M$$

Note that in the second two definitions, a command is being used in a place where we might normally expect a term in the $\lambda\mu_v$-calculus. This definition for the delimiter is observationally different from reset. For example, according to both $\lambda_\mathcal{S}$ and $\lambda\mu\widehat{\mathsf{tp}}$, the reset delimiter is idempotent:

$$\langle\langle M\rangle\rangle \triangleq \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]M =_{\mu_{\widehat{\mathsf{tp}}}} \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]M \triangleq \langle M\rangle$$

This equation makes sense because a reset forms a hard barrier that shift can never cross, so having more than one is redundant. However, the $\mathsf{reset}_0$ delimiter is *not* idempotent:

$$\langle\langle M\rangle_0\rangle_0 \triangleq [\widehat{\mathsf{tp}}][\widehat{\mathsf{tp}}]M \neq [\widehat{\mathsf{tp}}]M \triangleq \langle M\rangle_0$$

This equation is impossible because multiple calls to $\mathsf{shift}_0$ in a row can capture the evaluation context several layers out, effectively "digging" out of a series of nested $\mathsf{reset}_0$s (see Example 6). For example, by instantiating the above $M$ with $\mu\alpha.\mu\beta.(\lambda x.[\beta]x)$, we are able to differentiate between the contexts $[\widehat{\mathsf{tp}}][\widehat{\mathsf{tp}}]\Box$ and $[\widehat{\mathsf{tp}}]\Box$.[8] Note, however, that idempotency of reset is provable in $\Lambda\mu_v$ *regardless* of $M$, even if it does not come from $\lambda_\mathcal{S}$, so it is a property of $\Lambda\mu_v$ as a whole and not just the image of $\mathcal{SR}^{-1}$. In other words, this observational property of reset holds even with a program using $\mathsf{shift}_0$ to "dig" out of nested $\mathsf{reset}_0$ delimiters. It follows that not only are shift and $\mathsf{shift}_0$ different control operators, reset and $\mathsf{reset}_0$ are *different* delimiters, as shown by their different encodings and observational properties in the $\Lambda\mu_v$-calculus. Going the other way, Shan [33] shows how to operationally simulate $\mathsf{shift}_0$ in terms of shift by directly representing [15] stacks of continuations. However, here we are not only concerned with the operational correctness of encodings, but also in preserving the other observational properties of the various control operators, like the idempotency of reset in $\lambda_\mathcal{S}$ or the $\eta_\$$ axiom from $\lambda_\$$. *End remark* 2.

## 8. A parametric approach to delimited control

Having determined a second interpretation of the $\Lambda\mu$-calculus, we can now start to examine the impact of evaluation strategies on delimited control effects by seeing how the call-by-value and call-by-name theories are related. Herbelin and Ghilezan [17] use the call-by-name equational theory shown in Figure 21 for analyzing the relationship between the relaxed $\Lambda\mu$-calculus and delimited control. Now, observe how the call-by-name axioms vary from the call-by-value axioms from Figures 2 and 11. On the one hand, in the call-by-name theory, the $\beta$ rule for reducing function calls has been generalized so that the argument can be any term instead of a value, and likewise the $\eta$ rule for eliminating a trivial $\lambda$-abstraction has been generalized for any term. On the other hand, the $\mu$ rule for capturing an evaluation context has been restricted, since call-by-name has a more restricted form of evaluation contexts: including only chains of applications like $\Box\ N_1\ N_2\ N_3$. Contrastingly, the $\eta_\mu$ rule is completely unchanged between the two theories.

At a higher level, consider what happens when we move from a call-by-name theory, like the original $\lambda$-calculus, to a call-by-value theory. We must restrict rules like $\beta$ to only substitute the simpler subset of terms that we call "values" because it would be improper to substitute non-value terms — so by contrast every term in call-by-name is a "value" and subject to substitution. On the other hand, consider how to move from a call-by-value theory with control, like $\lambda\mu_v$ or $\lambda\,\mathsf{callcc}_v$, to a call-by-name theory. We must restrict rules like $\mu_v$ to avoid capturing certain contexts that are no longer "strict" and do not evaluate their inputs. For example, if we consider the $\lambda\mu$ term $(\lambda\_.M)\ (\mu\alpha.c)$, then the context $(\lambda\_.M)\ \Box$ is not strict — it never evaluates the term we plug into $\Box$ — so the correct move is to discard the argument $(\mu\alpha.c)$ instead of letting it capture its context.

Therefore, the differences between call-by-name and call-by-value can be summarized by the answers to two questions: "what terms are values?" and "what contexts are strict?" To further highlight the commonalities between the call-by-value and call-by-name theories of the $\Lambda\mu$-calculus, we present a unified *parametric* equational theory in Figure 22, similar to the parametric $\lambda$-calculus [27], but extended with control effects (both classical and delimited). In this theory, we state the basic axioms generically by assuming that there is some decision on what we mean by values $V$ and evaluation contexts $E$, and leaving their precise definitions to be filled in later. More specifically, a definition for the sets of values and evaluation contexts is a definition of a *strategy* $\mathcal{S}$. We then recover our previous call-by-value and call-by-name equational theories by instantiating the parametric $\Lambda\mu_\mathcal{S}$-calculus with the appropriate strategies shown in Figure 23. As expected, the $\Lambda\mu_\mathcal{V}$ gives

$$\begin{array}{rc} \beta_V & (\lambda x.M)\ V = M\{V/x\} \\ \eta_V & \lambda x.V\ x = V \\ \mu_E & [\alpha]E[\mu\beta.M] = M\{[\alpha]E[N]/[\beta]N\} \\ \eta_\mu & \mu\alpha.[\alpha]M = M \\ \beta_{E\Omega} & (\lambda x.E[x])\ M = E[M] \\ \beta_\mu & (\lambda x.\mu\alpha.[\beta]M)\ N = \mu\alpha.[\beta](\lambda x.M)\ N \end{array}$$

**Figure 22.** A parametric equational theory for the $\Lambda\mu_\mathcal{S}$-calculus.

$$V \in Value_\mathcal{V} ::= x \mid \lambda x.M$$
$$E \in StrictCxt_\mathcal{V} ::= \Box \mid E\ M \mid V\ E$$

$$V \in Value_\mathcal{N} ::= M \qquad E \in StrictCxt_\mathcal{N} ::= \Box \mid E\ M$$

**Figure 23.** Call-by-value ($\mathcal{V}$) and call-by-name ($\mathcal{N}$) strategies.

[8] This is a dual to the usual concept of observational equivalence: rather than having a context to look at the differences between two terms, here we are using a term to look at the differences between two contexts.

$$V \in Value ::= \lambda x.M \quad M \in Term ::= V \mid x \mid M\ N$$
$$A \in Answer ::= B[V] \quad B \in BindCxt ::= \Box \mid (\lambda x.B)\ M$$
$$E \in EvalCxt ::= \Box \mid E\ M \mid (\lambda x.E[x])\ E \mid (\lambda x.E)\ M$$

$$deref \quad (\lambda x.E[x])\ V = (\lambda x.E[V])\ V$$
$$lift \quad (\lambda x.A)\ M\ N = (\lambda x.A\ N)\ M$$
$$assoc \quad (\lambda y.E[y])\ ((\lambda x.A)\ M) = (\lambda x.(\lambda y.E[y])\ A)\ M$$

**Figure 24.** A call-by-need $\lambda$-calculus.

$$V \in Value_{\mathcal{AF}} ::= \lambda x.M$$
$$E \in EvalCxt_{\mathcal{AF}} ::= \Box \mid E\ M \mid (\lambda x.Q[x])\ E$$
$$Q \in Question ::= \Box \mid Q\ M \mid (\lambda x.Q[x])\ Q \mid (\lambda x.Q)\ M \mid [\alpha]Q \mid \mu\alpha.Q$$

**Figure 25.** A call-by-need ($\mathcal{AF}$) strategy.

us exactly the $\Lambda\mu_v$ theory, and in $\Lambda\mu_{\mathcal{N}}$ the $\beta_{E\Omega}$ and $\beta_\mu$ axioms fall out as derivable from the others, giving us the theory in Figure 21.

**Call-by-need and delimited control:** Having a parametric theory for the $\Lambda\mu$-calculus makes it easier to investigate the impact of a strategy on languages with classical or delimited control. For example, how might we fit call-by-need — a strategy for "more efficient" lazy evaluation that defers evaluating arguments to function calls until they are needed but remembers their value for future use — into the picture? To start, let's consider Ariola and Felleisen's [1] call-by-need $\lambda$-calculus, shown in Figure 24. We can see how the call-by-need $\lambda$-calculus defers and saves work by considering a possible evaluation trace for a term like $(\lambda x.M)\ N$:[9]

$$(\lambda x.\boxed{M})\ N = (\lambda x.E[x])\ \boxed{N} = \boxed{(\lambda x.E[x])\ V}$$
$$=_{deref} (\lambda x.E[V])\ V$$

However, life is not always so simple, since $N$ may not evaluate exactly to a value $V$, but instead it might be a value surrounded by context of bindings, $B[V]$. To deal with this issue, the call-by-need $\lambda$-calculus has the axiom $assoc$ that moves the evaluation context $(\lambda x.E[x])\ \Box$ through the bindings until it reaches the value. Similarly, $lift$ brings an evaluation context like $\Box\ N$ to a value.

In order to turn this call-by-need $\lambda$-calculus into an instance of the parametric theory we need to figure out what is the correct definition for values and evaluation contexts. First, let's consider just the pure $\lambda$-calculus subset of the theory. The definition for values can be taken rather directly from Figure 24 which says that only $\lambda$-abstractions are values. This decision gives us a $\beta_V$ rule that is a stronger version of $deref$ and can substitute a value into any context (recall the substitution procedure from Section 3). But what about the definition of evaluation contexts? The $\beta_{Elift}$ rule, $E[(\lambda x.M)\ N] = (\lambda x.E[M])\ N$, that is derivable from $\beta_{E\Omega}$ (see [19] for deriving $\beta_{lift}$ from $\beta_\Omega$) can simulate $lift$ and $assoc$ assuming that both $\Box\ M$ and $(\lambda x.E[x])\ \Box$ are considered evaluation contexts. Therefore, our evaluation contexts should include at least these two cases.[10]

[9] The evaluation context has been highlighted by drawing a box that separates the term currently being evaluated from its surrounding context.

[10] We can already make some interesting comparisons between the three strategies by the way axioms for the pure $\lambda$-calculus are used: in $\mathcal{N}$, $\beta_V$ is operational, $\eta_V$ is observational, and $\beta_{E\Omega}$ is trivial; in $\mathcal{V}$, $\beta_V$ is operational and $\eta_V$ and $\beta_{E\Omega}$ are observational; in $\mathcal{AF}$, $\beta_V$ and $\beta_{E\Omega}$ are operational and $\eta_V$ is trivial.

Now, let's consider what happens when we take control effects into account. In particular, does it make sense for context $(\lambda x.E)\ M$ to be capturable by the $\mu_E$ rule? For example, what happens if we have a classical control effect inside the body of a function call? Because of the $\beta_\mu$ axiom, we could either pull out the $\mu$-abstraction first or capture the binding context and copy it

$$[\alpha](\lambda x.\mu\beta.[\gamma]M)\ N =_{\beta_\mu,\mu_E} [\gamma](\lambda x.M\ \{\alpha/\beta\})\ N$$
$$=_{\mu_E} [\gamma]M\ \{[\alpha](\lambda x.N')\ N/[\beta]N'\}$$

leading to completely different sharing properties (whether there is one shared $N$ or every $\beta$ command gets a fresh copy of $N$) depending on which path you take. This same issue is raised in [5] on the interaction between bindings and control effects in call-by-need. Since the call-by-need $\lambda$-calculus is a calculus about sharing, non-deterministic sharing is undesirable, and we leave $(\lambda x.E)\ M$ out of our set of evaluation contexts. In a related question, what happens if we have an otherwise strict function whose body begins with a control effect, like $\lambda x.\mu\alpha.[\beta]x$? Should we be required to extract the $\mu$-abstraction out of the function body before realizing the function was strict all along? Let's suppose that we call this function strict, since it will evaluate $x$ in some evaluation context given by $\beta$, giving us the following step:

$$[\alpha](\lambda x.\mu\delta.[\beta]x)\ (\mu\gamma.M) =_{\mu_E} M\ \{[\alpha](\lambda x.\mu\delta.[\beta]x)\ N/[\gamma]N\}$$

These decisions give us the strategy for call-by-need shown in Figure 25, where the strict evaluation contexts are either a calling context, $E\ M$, or an application of a strict function that has some question about its argument, $(\lambda x.Q[x])\ E$. Also note that because the parametric $\Lambda\mu_{\mathcal{S}}$-calculus includes delimited control, this gives us a definition of the call-by-need evaluation strategy that works with delimited control effects. For example, in the terminology of the $\lambda\mu\widehat{\mathsf{tp}}$-calculus, we can derive the following equations by their embedding in the $\Lambda\mu_{\mathcal{AF}}$-calculus:

$$[\widehat{\mathsf{tp}}]\mu\alpha.M =_{\mu_E} M\ \{\widehat{\mathsf{tp}}/\alpha\} \qquad \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]V =_{\beta_V,\mu_E,\eta_\mu} V$$

These equations are not terribly surprising, they are quite similar to the call-by-value axioms for $\widehat{\mathsf{tp}}$, but we were able to derive them completely from our decisions about what we mean by values and evaluation contexts. Of note, one difference with call-by-value is that $\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]x$ does *not* reduce to $x$, because we said that $x$ is not a value. However, it is relatively straightforward to come up with a call-by-need strategy with variables in the set of values, and to pull out the corresponding instance of the $\Lambda\mu_{\mathcal{S}}$ equational theory.

## 9. Conclusion

By now, we have seen how the core of classical control effects, as expressed by callcc in Scheme, naturally scales to the far more powerful world of delimited control. No new programming constructs or rules are required to explain delimited control — the characterization of classical control is enough. On the one hand, this conclusion may be rather surprising. After all, we know that delimited control operators like shift and reset are vastly more powerful than callcc. On the other hand, the conclusion should be reassuring. Compared to composable continuations, we have a relatively better understanding of callcc, so the fact that there are no new ingredients means that there is nothing extra that we have to explain. The theory that we get out for delimited control is the one that (1) arises canonically from a sub-language of classical control, (2) doesn't add any new programming constructs, and (3) applies the existing rules without change to a more flexible syntax.

Although they were introduced around the same time, presented side-by-side by Danvy and Filinski [7], the shift and reset formulation of delimited control receives more attention than the relatively neglected variants $\mathsf{shift}_0$ and $\mathsf{reset}_0$. For example, there was a ten-year lag between the developments of equational theories for shift and reset [19] and for $\mathsf{shift}_0$ and $\mathsf{reset}_0$ [20]. However, this analysis of composable continuations suggests that $\mathsf{shift}_0$ is a more "primitive" control operator than shift, while still allowing for desirable and strong observational guarantees that hold for reset (like idempotency), and warrants more attention. On the one hand, Herbelin and Ghilezan [17] conjecture that the $\lambda\mu\widehat{\mathsf{tp}}$-calculus, and by extension the call-by-value $\lambda$-calculus with shift and reset, satisfy Böhm's theorem of separability. On the other hand, the close relationship with the call-by-value $\Lambda\mu_v$-calculus instead suggests beginning with separability of the $\mathsf{shift}_0$ and $\mathsf{reset}_0$ notion of composable continuations (or better yet $\mathsf{shift}_0$ and \$), from which the others may follow.

The $\Lambda\mu$-calculus may also serve as a more general framework for understanding control effects in functional programming languages. For instance, although we focus on untyped semantics here, we can still consider type and effect systems for such languages. Materzok [20, 21] provides a type system for the call-by-value $\lambda$-calculus with the $\mathsf{shift}_0$ operator, which may be translated to the call-by-value $\Lambda\mu_v$-calculus since the two are isomorphic. Additionally, Saurin [32] provides a type system for the call-by-name $\Lambda\mu$-calculus. It would be interesting to use the parametric $\Lambda\mu$-calculus as a common language to compare the two type systems, and to see the impact that evaluation strategy has on the static typing of programs using delimited control. This may suggest a generalized type and effect systems for static analysis of delimited control under both call-by-value and call-by-name evaluation strategies in the same vein as the parametric equational theory for the $\Lambda\mu$-calculus. Furthermore, in practice functional languages with delimited control allow for the use of multiple different prompts [12] for different purposes, like the ability to create exception handlers that catch only certain kinds of exceptions. The $\lambda\mu$-calculus has already [11] shed some light on this language feature, and the analysis of delimited control in the $\Lambda\mu$-calculus may provide more insight into a formulation that is practical yet easy to reason about with strong observational guarantees similar to shift and reset.


### Acknowledgements

We would like to thank Matthew Might and the anonymous reviewers for their feedback and help in improving this paper. Paul Downen and Zena M. Ariola have been supported by NSF grant CCF-0917329.


## References


[1] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programmin*, 7(3):265–301, 1997.

[2] Z. M. Ariola and H. Herbelin. Control reduction theories: the benefit of structural substitution. *Journal of Functional Programming*, 18(3): 373–419, 2008.

[3] Z. M. Ariola, H. Herbelin, and A. Sabry. A proof-theoretic foundation of abortive continuations. *Higher-Order and Symbolic Computation*, 20(4):403–429, 2007.

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

[5] Z. M. Ariola, P. Downen, H. Herbelin, K. Nakata, and A. Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In *FLOPS*, pages 32–46, 2012.

[6] P.-L. Curien and H. Herbelin. The duality of computation. In *ICFP*, pages 233–243, 2000.

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

[8] O. Danvy and A. Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

[9] R. David and W. Py. Lambda-mu-calculus and Böhm's theorem. *Journal of Symbolic Logic*, 66(1):407–413, 2001.

[10] P. de Groote. On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In *LPAR*, pages 31–43, 1994.

[11] P. Downen and Z. M. Ariola. Delimited control and computational effects. *Journal of Functional Programming*, 24(1):1–55, 2014.

[12] R. K. Dybvig, S. P. Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(06): 687–730, 2007.

[13] K. etsu Fujita. Explicitly typed $\lambda\mu$-calculus for polymorphism and call-by-value. In *TLCA*, pages 162–176, 1999.

[14] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2): 235–271, 1992.

[15] A. Filinski. Representing monads. In *POPL*, pages 446–457, 1994.

[16] T. Griffin. A formulae-as-types notion of control. In *POPL*, pages 47–58, 1990.

[17] H. Herbelin and S. Ghilezan. An approach to call-by-name delimited continuations. In *POPL*, pages 383–394, 2008.

[18] H. Herbelin and S. Zimmermann. An operational account of call-by-value minimal and classical $\lambda$-calculus in "natural deduction" form. In *TLCA*, pages 142–156, 2009.

[19] Y. Kameyama and M. Hasegawa. A sound and complete axiomatization of delimited continuations. In *ICFP*, pages 177–188, 2003.

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

[21] M. Materzok and D. Biernacki. Subtyping delimited continuations. In *ICFP*, pages 81–93, 2011.

[22] M. Materzok and D. Biernacki. A dynamic interpretation of the CPS hierarchy. In *APLAS*, pages 296–311, 2012.

[23] E. Moggi. Computational $\lambda$-calculus and monads. In *Logic in Computer Science*, 1989.

[24] C.-H. L. Ong and C. A. Stewart. A curry-howard foundation for functional computation with control. In *POPL*, pages 215–227, 1997.

[25] M. Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR*, pages 190–201, 1992.

[26] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[27] S. Ronchi Della Rocca and L. Paolini. *The Parametric Lambda Calculus: A Metamodel for Computation*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[28] A. Sabry. Note on axiomatizing the semantics of control operators. Technical Report CIS-TR-96-03, Department of Computer and Information Science, University of Oregon, 1996.

[29] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

[30] A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, 1997.

[31] A. Saurin. Separation with streams in the $\lambda\mu$-calculus. In *LICS*, pages 356–365, 2005.

[32] A. Saurin. Typing streams in the $\Lambda\mu$-calculus. *ACM Transactions on Computational Logic*, 11(4), 2010.

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