The Revised Report

on the

Syntactic Theories of Sequential Control and State

Matthias Felleisen
Department of Computer Science
Rice University
Houston, TX 77251-1892

Robert Hieb
Computer Science Department
Indiana University
Bloomington, IN 47405

Rice University Technical Report # 100-89
to appear in: Theoretical Computer Science 1991

The Revised Report on the Syntactic Theories of Sequential Control and State

Matthias Felleisen*
Department of Computer Science, Rice University, Houston, TX 77251-1892

Robert Hieb†
Computer Science Department, Indiana University, Bloomington, IN 47405

Abstract

The syntactic theories of control and state are conservative extensions of the λ_v-calculus for equational reasoning about imperative programming facilities in higher-order languages. Unlike the simple λ_v-calculus, the extended theories are mixtures of equivalence relations and compatible congruence relations on the term language, which significantly complicates the reasoning process. In this paper we develop fully compatible equational theories of the same imperative higher-order programming languages. The new theories subsume the original calculi of control and state and satisfy the usual Church-Rosser and Standardization Theorems. With the new calculi, equational reasoning about imperative programs becomes as simple as reasoning about functional programs.

1 The syntactic theories of control and state

Most λ-calculus-based programming languages provide imperative programming facilities such as assignment statements, exceptions, and continuations. Typical examples are ML [16], Scheme [19], and Common Lisp [20]. While these additions add expressive power and increase the efficiency of programs, they also appear to invalidate the simple reduction rules and equational reasoning of the λ-calculus that make functional programming so appealing. In two previous papers [8, 9], we have shown that there are conservative extensions of Plotkin's λ_v-calculus [18] for such programming languages, and that it is possible to reason about programs in extended functional languages in an equational style.

The main difference between the simple lambda-calculi and its extended versions is a distinction between two classes of equations: equations for ordinary *expressions* and equations for *whole programs*. The reason for this distinction is the need to arrange the effects of assignments and jumps in the appropriate order. For example, a program like

*Supported in part by NSF grants CCR 88-07520 and 89-17022, and DARPA/NSF grant CCR 87-20277.
†Supported in part by NSF grant CCR 88-03432.

$x := 3;\ y := x + 1$ is equivalent to the program $y := 4$, yet the subexpression $x := 3;\ y := x + 1$ cannot be replaced by $y := 4$ because the context in which the former expression occurs may contain further references to x and these references must be able to perceive the assignment to x. Still, the calculi satisfy relatively simple variants of the Church-Rosser and Standardization properties. Most importantly, they satisfy most of Plotkin's [18] criteria for a correspondence between a programming language and a reduction-based calculus:

(1) the standard derivations of the calculi yield the same value for a program as the operational semantics, and

(2) a *subset* of the calculi equations for ordinary expressions are operationally sound.

The first property is important because a programmer can use the relatively simple reduction system to determine the value of an imperative higher-order program by rewriting the program until it becomes a value. The second property is a basis for program transformations and program correctness proofs. However, as the restriction in (2) indicates, the calculi are *complicated* equational theories because some equivalences are not equations in the usual sense. This distinction is unnatural and leads to problems in reasoning about equational properties of programs.

One way to simplify the equational theories for an imperative programming language is to modify the programming language. For example, we recently showed that by adding a *control delimiter* facility to the λ-calculus extended with control operators, we can simplify the calculus and get a more elegant relationship between the language and its calculus [6]. But, although this proposal provides a good example of how calculus design can influence and improve language design, it does not alleviate the need for better techniques for reasoning about *existing* languages. Languages such as Scheme, ML, and Common Lisp have grown through practical experiences and support practical applications, and they need calculi that are tuned towards their specific needs.

The solution to the problem is to relax Plotkin's first correspondence criterion. More precisely, we no longer require that the standard derivation of the programming language calculi terminate in a *value* when the machine produces a value for a program. Instead, we allow the standard derivation to produce some other kind of term that is recognizable as a *final answer*. For both kinds of imperative extensions, i.e., control operators and assignments, the result is a simple equational calculus for imperative, higher-order programming languages that can prove the same set of observational equivalences as the old calculus but with an elegant axiomatic basis. Indeed, reasoning with the new calculi is as simple as reasoning with the traditional λ-calculus.

In the next section, we briefly summarize Plotkin's work on the λ_v-calculus since it constitutes the basis of our research. Sections 3 and 4 present our new theories of control and state, respectively. These sections begin by briefly introducing our old calculi, which provide machine-independent semantics for the languages and standards against which to measure the new theories. Next, these sections introduce the new calculi and analyze the relationships between the old and new calculi. The fifth section describes the merger of the two theories. Finally we discuss related work and some implications of our work for an alternative denotational semantics for extended functional languages.

2 The λ-value-calculus

The expression language Λ of the λ-calculus and the λ_v-calculus [2, 3, 18] is the union of a set of values and expression juxtapositions:

$$e ::= v \mid (e\ e).$$

The set of values is the collection of basic constants ($b \in \mathit{BConsts}$) and functional constants ($f \in \mathit{FConsts}$), variables ($x \in \mathit{Vars}$) and λ-abstractions:

$$v ::= b \mid f \mid x \mid \lambda x.e.$$

Constants correspond to built-in algebraic language primitives like numbers and booleans and (mathematical) functions on them; identifiers are placeholders for values; and λ-abstractions are call-by-value procedures. Expression juxtaposition denotes function application.

The only binding construct in the programming language is λ-abstraction. The set of *closed expressions*, Λ^0, is the set of all expressions with no free variables; Values^0 is the set of closed values. We adopt Barendregt's [2] conventions on bound variables and abstractions:

- Bound variables are always distinct from free variables in the various expressions of mathematical definitions and claims.
- Abstractions that only differ by a renaming of bound variables are identified, e.g., $\lambda x.x \equiv \lambda y.y$.

The expression $e[x \leftarrow e_1]$ is the result of substituting the expression e_1 for a free variable x in the expression e.

An important parameter of the language definition is the set of constants and its interpretation. Following Plotkin [18], we assume that the behavior of constants is specified by a partial function from functional and basic constants to closed values:

$$\delta : \mathit{FConsts} \times \mathit{BConsts} \longrightarrow \mathit{Values}^0.$$

In the mid-60's, Landin [11, 12] illustrated in a series of papers that Λ is an interesting and powerful programming language. Most importantly, he showed how a simple stack-based calculator for algebraic expressions could be extended to the abstract SECD-machine for evaluating complete Λ-programs. From a programmer's perspective, the SECD-machine is an interpreter that implements a partial function from programs to answers, where the former are closed expressions and the latter are closed values:

$$\mathit{eval}_{\mathrm{SECD}} : \Lambda^0 \longrightarrow \mathit{Values}^0.$$

The use of Λ as a programming language with an operational semantics and as the term language for Church's λ-calculus [3] raises the natural question of how the two concepts correspond to each other. Plotkin [18] provided the answer by defining the λ_v-calculus, which matches the evaluation function $\mathit{eval}_{\mathrm{SCED}}$, and by providing a modified SECD-machine, which implements the λ-calculus correctly in the above sense of a Landin-style interpreter. The original SECD-semantics and the λ_v-calculus precisely model the call-by-value parameter-passing technique that is now predominant in the functional subsets of programming languages. Besides being easy to implement, call-by-value provides an obvious order of evaluation, which facilitates the addition of imperative features. There is, however, no *theoretical* reason for choosing one over the other, even in the presence of control operators and assignments.

The λ_v-calculus is an equational theory about Λ. More precisely, it is a set of equations that is based on a set of term relations on Λ. The two basic relations, *notions of reduction*, are:

$$fa \longrightarrow \delta(f,a) \tag{δ}$$

$$(\lambda x.e)v \longrightarrow e[x \leftarrow v]. \tag{β_v}$$

The equational theory $\boldsymbol{\lambda}_v$ is the smallest congruence relation generated from the above relations. For the formal definition, we rely on the concept of a term context, which are expressions with a hole ($[\ ]$) at the place of a subexpression:

$$C ::= [\ ] \mid (e\ C) \mid (C\ e) \mid (\lambda x.C).$$

The expression $C[e]$ stands for the result of putting the expression e into the hole of the context C, which may bind free variables in e.

Given the notions of reduction and the definition of contexts, the definition of $\boldsymbol{\lambda}_v$ is straightforward.

Definition 2.1. **($\boldsymbol{\lambda}_v$)** The basic notion of reduction is

$$\mathbf{v} = \delta \cup \beta_v.$$

The *one-step* **v**-*reduction* $\longrightarrow_v$ is the compatible closure of **v**: $e \longrightarrow_v e'$ if $(p,q) \in \mathbf{v}, e \equiv C[p]$, and $e' \equiv C[q]$ for some expressions p and q and context C. The **v**-*reduction* is denoted by $\longrightarrow\!\!\!\!\!\rightarrow_v$ and is the reflexive, transitive closure of $\longrightarrow_v$; $=_v$ is the smallest equivalence relation generated by $\longrightarrow_v$. If $e_1 =_v e_2$, we write $\boldsymbol{\lambda}_v \vdash e_1 = e_2$.

The λ_v-calculus has the same characteristic properties as Church's original λ-calculus. First, the defining notion of reduction, **v**, is Church-Rosser, i.e., the **v**-reduction satisfies the diamond property.

Theorem 2.2 (Plotkin) *If $e \longrightarrow\!\!\!\!\!\rightarrow_v e_1$ and $e \longrightarrow\!\!\!\!\!\rightarrow_v e_2$, then there exists an expression e' such that $e_1 \longrightarrow\!\!\!\!\!\rightarrow_v e'$ and $e_2 \longrightarrow\!\!\!\!\!\rightarrow_v e'$.*

Second, for every sequence of (single) reduction steps from one term to another, there is a canonical sequence of steps between the same terms that can be found algorithmically. This idea is important for an analysis of the correspondence between a calculus and an abstract machine. While it is easy to see that $e \longrightarrow\!\!\!\!\!\rightarrow_v v$ if $eval_{\mathrm{SECD}}(e) = v$, the inverse is not correct. If $e \longrightarrow\!\!\!\!\!\rightarrow_v v$ and v is a λ-abstraction, then there are possibly many different values to which e reduces, yet $eval_{\mathrm{SECD}}$, the interpreter, can only yield one value for e. To determine this value via a sequence of reductions, we need canonical reductions and an algorithm to compute them.

To describe the basis of the algorithm and to state the corresponding theorem, we need some definitions. An evaluation context is a special kind of context. The hole of an evaluation context is in such a position that a δ- or β_v-redex inserted in the hole is the leftmost-outermost redex that is not inside of a λ-abstraction. We let E range over the set of evaluation contexts and define it with the following grammar:

$$E ::= [\ \ ] \mid (v\ E) \mid (E\ e).$$

Given the definition of an evaluation context, we say that e *standard reduces* to e' if the reduction occurs in an evaluation context. In other words, a standard reduction function always picks the leftmost-outermost **v**-redex outside the scope of a λ-expression. It is undefined on values.

Definition 2.3. (*Standard Reduction Function*) The standard reduction function maps e to e', $e \longmapsto_v e'$, if for some evaluation context E, $e \equiv E[p]$, $e' \equiv E[q]$ and $(p, q) \in \mathbf{v}$. We use $\longmapsto_v^*$ to denote the transitive closure of the standard reduction function.

The concept of standard reduction *sequences* generalizes the idea of a standard reduction function such that standard reductions become applicableto arbitrary term positions. A standard reduction sequence also permits incomplete reduction sequences that may choose *not* to reduce a leftmost-outermost redex for the rest of the sequence.

Definition 2.4. (*Standard Reduction Sequences*) The set of standard reduction sequences is defined as follows:

1. Every constant and variable is a standard reduction sequence.
2. If $e_1, \ldots, e_n$ is a standard reduction sequence, then so is $\lambda x.e_1, \ldots, \lambda x.e_n$.
3. If $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$ are standard reduction sequences, then so is
$$p_1q_1, p_2q_1, \ldots, p_nq_1, p_nq_2, \ldots, p_nq_m.$$
4. If $e_1, \ldots, e_n$ is a standard reduction sequence and $e \longmapsto_v e_1$, then $e, e_1, \ldots, e_n$ is a standard reduction sequence.

We can now formalize a Curry-Feys-style Standardization Theorem.

Theorem 2.5 (Plotkin) *$e \longrightarrow\!\!\!\!\!\rightarrow_v e'$ if and only if there is a standard reduction sequence $e, \ldots, e'$.*

Together, the Church-Rosser and Standard Reduction Theorems show that there is a perfect correspondence between the SECD-evaluation function and the standard reduction function.

Theorem 2.6 (Plotkin) *Let e, v be closed terms in Λ. Then, $e \longmapsto_v^* v$ if and only if $eval_{\mathrm{SECD}}(e) = v$.*

In other words, the SECD-machine terminates and returns a value for a program if and only if the program standard reduces to the same value. It is therefore possible to define the evaluation function via the standard reduction function, ignoring the details of the actual machine:

$$eval_v(e) \stackrel{df}{\equiv} v \text{ iff } e \longmapsto_v^* v.$$

After determining that *reductions* in the calculus correspond to evaluations on a machine, the question remains what *equations* on the calculus mean for a programmer. To understand this relationship, we recall that a programmer can only observe the effects of *entire programs* via the evaluator. Thus, to compare expressions as black boxes, a programmer must rely on those equivalences that the evaluation function can validate for all programs in which the expression can occur. This argument naturally leads to the definition of the operational equivalence relation.

Definition 2.7. (*Operational Equivalence*) Two terms, e and e', are *operationally equivalent*, $e \simeq_v e'$, if and only if they are indistinguishable in all program contexts C:

$$eval_v(C[e]) \text{ terminates iff } eval_v(C[e']) \text{ terminates}$$

and

$$eval_v(C[e]) = b \text{ iff } eval(C[e']) = b$$

for some basic constant b.

Plotkin [18] showed that the λ_v-calculus is sound with respect to operational equivalence.

Theorem 2.8 (Plotkin) *If* $\boldsymbol{\lambda}_v \vdash e = e'$ *then* $e \simeq_v e'$. *The inverse direction does not hold.*

Theorems 2.5 and 2.8 are the basis of a formal correspondence relation between programming languages and calculi. They stipulate that

1. a calculus can evaluate a program in the same way as an independently given operational semantics; and

2. the equations of a calculus imply the interchangeability of expressions in arbitrary contexts.

These two criteria are the basis for any further development of programming language calculi.

3 Theories of control

The language Λ_c for programming with procedural and *control* abstractions is an extension of Λ with a set of $\mathcal{C}$-applications of the form $(\mathcal{C}e)$:

$$e ::= v \mid (ee) \mid (\mathcal{C}e).$$

A $\mathcal{C}$-application applies its subexpression to an abstraction of the current control context, the *continuation*. The application takes place in the empty control context, the *halt* continuation. A continuation has the same first-class status as a λ-abstraction; upon invocation, it discards the control context of the application and resumes the abstracted control context with its argument.

This notion of control abstraction is derived from the treatment of continuations in the programming language Scheme [19]. However, although the continuation created by a $\mathcal{C}$-application acts just like a continuation created by the Scheme continuation constructor *call/cc*, a $\mathcal{C}$-application differs from a *call/cc* application in that the former aborts the current control context, whereas the latter leaves the current control context intact. This abortive affect allows us to define an abort abstraction as an abbreviation of a $\mathcal{C}$-application whose subexpression is a procedure that ignores its argument:

$$\mathcal{A}\ e \overset{df}{\equiv} \mathcal{C}(\lambda d.e) \text{ where } d \notin FV(e).$$

The effect of $(\mathcal{A}\ e)$ is an abort of the program evaluation. It discards the current control context and returns the value of its subexpression as the final value of the program. This abbreviation is used to simplify the reduction rules for $\mathcal{C}$-applications.

Other than the introduction of $\mathcal{C}$-applications, the syntax of Λ is adopted *mutatis mutandis*. The definition of the set of values retains its shape, even though subexpressions are in the extended language Λ_c:

$$v ::= b \mid f \mid x \mid \lambda x.e.$$

Similarly, the specification of the set of evaluation contexts stays the same:

$$E ::= [\ \ ] \mid (v\ E) \mid (E\ e),$$

but it now denotes the set of evaluation contexts whose subexpressions are in the extended language Λ_c.

The following subsection briefly presents our original theory of control abstractions with an emphasis on the set of *safe* equations; for a more complete description, we refer the reader to the earlier report [9]. The second subsection contains the development of a finite axiomatization of the theory of safe equations. The Plotkin-style correspondence theorem relies on a proof of equivalence between the two calculi and on the idea that the old calculus is an acceptable specification of the semantics of Λ_c. The final subsection presents two interesting extensions of the equational theory.

3.1 A syntactic theory of control abstractions

Originally we derived the syntactic theory of control from an abstract operational semantics based on Landin's SECD-machine [7]. Eliminating all non-program text components from the machine shows that the concept of "current continuation" is equivalent to the notion of evaluation context. The machine transition rules for abstracting a control state naturally lead to two term relations that gradually lift a $\mathcal{C}$-application to the top of an evaluation context while encoding the context as an abstraction.

When the $\mathcal{C}$-expression, $\mathcal{C}e$, occurs as the function part of an application, $(\mathcal{C}e)e'$, its immediate continuation is the application of a yet-unknown function f to the expression e'. The rest of the continuation, k, is the continuation of the entire application. Composing the two pieces, $k(fe')$, yields the functional part of the continuation of $\mathcal{C}e$, which in turn is the argument for e. Since this continuation must abort its context upon invocation, we wrap this expression in an $\mathcal{A}$-application. To obtain the outer part of the continuation, we use another $\mathcal{C}$-application:

$$(\mathcal{C}\ e)e' \longrightarrow \mathcal{C}(\lambda k.e\,(\lambda x.\mathcal{A}(k\,(xe')))).$$

Similarly, when the control expression occurs as the argument part of an application, the abstraction of the control context applies the known function to an unknown argument, passing the result to the continuation of the entire application:

$$v(\mathcal{C}\ e) \longrightarrow \mathcal{C}(\lambda k.e\,(\lambda x.\mathcal{A}(k\,(vx)))).$$

The assumption that the left part of the application is a value reflects the left-to-right evaluation order of the underlying language.

To facilitate the formal definition and future reference to the above rules, we introduce the notion of a *singular* evaluation context:

$$E^s ::= (v\ [\ \ ])\mid([\ \ ]\ e).$$

Using singular evaluation contexts, one definition schema suffices for specifying both of the above reduction relations for Λ_c:

$$E^s[\mathcal{C}\ e] \longrightarrow \mathcal{C}(\lambda k.e\,(\lambda x.\mathcal{A}(k\ E^s[x]))). \qquad (C_{\mathit{lift}})$$

The relation for lifting $\mathcal{C}$-applications gives rise to an extended notion of reduction:

$$\mathbf{c} \;=\; \delta \cup \beta_v \cup C_{\mathit{lift}}. \qquad (\mathbf{c})$$

This notion of reduction defines a full reduction relation and a congruence relation in the usual way. It is Church-Rosser and has Curry-Feys-style standard reduction sequences. The symbol $\longmapsto_c$ denotes the standard reduction function for $\mathbf{c}$. The respective theorems and proofs are straightforward adaptations of the proofs of Plotkin's corresponding theorems. We use $\boldsymbol{\lambda}_v\text{-}\mathbf{C}(\mathbf{c}) \vdash e = e'$ if $e =_c e'$.

For a complete simulation of an abstract machine for Λ_c, the reduction based on $\mathbf{c}$ is insufficient because $\mathcal{C}$-applications get stuck at the top of the program. We therefore introduce a *computation rule* that maps a $\mathcal{C}$-application at the top of the program into an application of its subterm to the *halt* continuation $(\lambda x.\mathcal{A}x)$:

$$\mathcal{C}\ e \;\triangleright\; e(\lambda x.\mathcal{A}x). \qquad (C_T)$$

Together with the extended reduction $\longrightarrow\!\!\!\!\rightarrow_c$, the computation rule forms a *computation relation*

$$\triangleright_c = \longrightarrow\!\!\!\!\rightarrow_c \cup\ C_T.$$

The computation relation satisfies the diamond property, i.e., if $e \triangleright_c e_1$ and $e \triangleright_c e_2$, then for some e', $e_1 \triangleright_c e'$ and $e_2 \triangleright_c e'$. But, since C_T only applies to entire programs, the computation relation *cannot* satisfy the full Church-Rosser property. Similarly, there are standard *computation* sequences, which are weak forms of standard reduction sequences. The computation relation generates an equivalence relation on programs, which we refer to as $\overset{\triangleright}{=}_c$. We also write $\boldsymbol{\lambda}_v\text{-}\mathbf{C}^{\triangleright} \vdash e = e'$ if $e \overset{\triangleright}{=}_c e'$. Based on the diamond property of the computation relation and the Church-Rosser property of the reduction **c**, it is easy to show that the theory $\boldsymbol{\lambda}_v$-$\mathbf{C}^{\triangleright}$ is a conservative extension of $\boldsymbol{\lambda}_v$.

The *standard computation function* is a generalization of the notion of a standard reduction function and always performs the leftmost-outermost computation step. Like the standard reduction function, it is undefined on values.

Definition 3.1. (**c**-*Standard Computation Function*) The *standard computation function* maps a program e to a program e', $e \overset{\triangleright}{\longmapsto}_c e'$, if e standard reduces to e' or if e computes to e': $\overset{\triangleright}{\longmapsto}_c \, = \, \longmapsto_c \cup C_T$.

The standard computation function faithfully simulates evaluation on a machine for Λ_c, i.e., we can use it to define a semantics instead of a machine with complex states:

$$eval_c(e) = v \qquad \text{if} \qquad e \overset{\triangleright}{\longmapsto}{}_c^{*} \, v.$$

This, in turn, gives rise to an operational equivalence relation in the usual manner. A Λ_c-expression e is operationally equivalent to e', $e \simeq_c e'$, if and only if the two are indistinguishable in the sense of Definition 2.7 relative to all Λ_c-program contexts.

From the design of the control calculus, it follows that congruences generated by **c**-reduction are operationally sound, but, due to their context-sensitivity, equations based on the computation relation are not.

Theorem 3.2 ([9]) *Let e and e' be in Λ_C.*

(i) *If* $\boldsymbol{\lambda}_v$-$\mathbf{C}(\mathbf{c}) \vdash e = e'$ *then* $e \simeq_c e'$.

(ii) $\boldsymbol{\lambda}_v$-$\mathbf{C}^{\triangleright} \vdash e = e'$ *does* **not** *imply* $e \simeq_c e'$.

Fortunately, it is possible to factor out a large subset of equations in $\boldsymbol{\lambda}_v$-$\mathbf{C}^{\triangleright}$ that are operationally sound: the *safe* equations.

Definition 3.3. (*C-Safe Equations*) An equation $e \overset{\triangleright}{=}_c e'$ is *safe* if and only if it holds in all evaluation contexts: $\boldsymbol{\lambda}_v$-$\mathbf{C}^{\triangleright} \vdash E[e] = E[e']$ for all E.

Operationally, the two terms of a safe equation have the same control effects. In order to enrich the set of safe equations we also permit the use of safe equations in the safeness proof of an equation. We use $\boldsymbol{\lambda}_v$-**C**-**safe** to refer to the equational theory generated by safe equations. The safe theory is again a conservative extension of $\boldsymbol{\lambda}_v$ and, more importantly, reduces reasoning about operational equivalence from the set of all contexts to the set of evaluation contexts.

Theorem 3.4 ([9]) *If* $\boldsymbol{\lambda}_v$-**C**-**safe** $\vdash e = e'$ *then* $e \simeq_c e'$.

In summary, the calculus of control abstractions is like an ordinary λ-calculus with Church-Rosser and Standardization Theorems. Moreover, it closely corresponds to the programming language definition for Λ_c. If we need to evaluate a program, the standard computation will produce the correct value; for proving operationally sound equations, we often must work in the theory of safe equations. In general, we will be more interested in the latter than the former because most interesting properties of programs are characterized by safe equations. Unfortunately, working with the theory of safe equations is not as easy as working with the λ-calculus since it is not a simple axiomatic theory with a finite set of axioms or axiom schemas but a theory based on a filtered subset of another theory, $\boldsymbol{\lambda}_v$-$\mathbf{C}^{\triangleright}$. We introduce a simple axiomatic characterization of safeness in the next subsection.

3.2 An axiomatic basis for safe equations

The disturbing element in the calculus of control abstractions is the rule C_T. The purpose of the rule is to replace a $\mathcal{C}$-application $(\mathcal{C}e)$ at the root of a program with an application of e to the *halt* continuation. For an axiomatic characterization of safe equations, we must find a way of replacing this special relation with simple notions of reduction that approximate its effect.[1]

A partial solution is to leave $\mathcal{C}$-applications at the root of the program alone and to continue with the evaluation of the subexpression. More precisely, when a $\mathcal{C}$-application reaches the root of the program after a number of C_{lift} reductions, it has the shape $(\mathcal{C}\lambda k.e)$, and an evaluation may continue with e. But this clearly leads to an accumulation of $\mathcal{C}$-applications at the root of a program. By observing that the outermost $\mathcal{C}$-application removes the current continuation and that therefore the next $\mathcal{C}$-application's continuation is the *halt* continuation, we are led to a rule that captures the idempotency of the abort action of $\mathcal{C}$-applications:

$$\mathcal{C}(\lambda k.\mathcal{C}\ e) \longrightarrow \mathcal{C}(\lambda k.e(\lambda x.\mathcal{A}x)). \qquad (C_{idem})$$

The only exception to this reasoning is the case where the program is already a $\mathcal{C}$-application and the subexpression is not an abstraction. We therefore need a rule for transforming an arbitrary subexpression of a $\mathcal{C}$-application into a λ-abstraction. The task of this abstraction is to receive a continuation and to apply the subexpression to it. A first attempt at the rule could be

$$\mathcal{C}\ e \longrightarrow \mathcal{C}(\lambda k.e\ k).$$

Unfortunately, this version is not strong enough. If, for example, e is a λ-abstraction that eventually causes an application of k to some value, the reduction would be stuck and no further evaluation would be possible. The solution is to replace k by $(\lambda x.\mathcal{A}(kx))$ so that an application of the continuation can initiate a program abort. Putting things together, the additional rule becomes

$$\mathcal{C}\ e \longrightarrow \mathcal{C}(\lambda k.e\ (\lambda x.\mathcal{A}(k\ x))). \qquad (C_{top})$$

Although a C_{idem} redex is also a C_{top} redex, this ambiguity causes no problem: by imposing an appropriate condition on the standard reduction function (see below), it is still possible to emulate a deterministic machine.

Together, the two new relations, C_{idem} and C_{top}, can closely simulate the top-level rule of $\boldsymbol{\lambda}_v$-$\mathbf{C}^{\triangleright}$. Indeed, the entire system of $\mathbf{c}$-reduction, C_{idem}, and C_{top} suffices for simulating

[1] Tim Griffin independently and simultaneously discovered another solution while studying the connection between a typed variant of the control calculus and classical logic [10]. He proposes to restrict the set of programs to expressions of the form $\mathcal{C}(\lambda k.ke)$ and to use C_{idem} as a replacement for C_T.

a complete evaluation. To begin, we introduce the notion of reduction

$$\mathbf{d} = \mathbf{c} \cup C_{idem} \cup C_{top}.$$

As usual, we let $\longrightarrow_d$ and $\longrightarrow\!\!\!\!\!\rightarrow_d$ stand for the respective one-step reduction and its transitive closure. Furthermore, we write $\boldsymbol{\lambda}_v\text{-}\mathbf{C(d)} \vdash e = e'$ if $e =_d e'$. Next we characterize the relationship between standard computation and the new reduction system with three lemmas. Clearly, the new system subsumes the standard computations that are entirely based on **c**-reductions.

Lemma 3.5 *If* $e \overset{\triangleright}{\longmapsto}{}^*_c e'$ *without use of* C_T *then* $e \longrightarrow\!\!\!\!\!\rightarrow_d e'$.

Proof. Suppose $e \overset{\triangleright}{\longmapsto}{}^*_c e'$ without use of C_T. Then $e \longmapsto^*_c e'$ and therefore $e \longrightarrow\!\!\!\!\!\rightarrow_d e'$. □

Given the operational motivation behind the introduction of C_{idem}, it should also be obvious that once a $\mathcal{C}$-application is at the root of the program the evaluation proceeds as before.

Lemma 3.6 *If* $e \overset{\triangleright}{\longmapsto}{}^*_c e'$ *then* $\mathcal{C}(\lambda k.e) \longrightarrow\!\!\!\!\!\rightarrow_d \mathcal{C}(\lambda k.e')$.

Proof. Since C_{idem} is essentially an instance of C_T inside of the context $\mathcal{C}(\lambda k.[\ \ ])$, every computation step in the old derivation is a reduction step inside of $\mathcal{C}(\lambda k.[\ \ ])$ in the revised calculus. The rest follows by transitivity. □

Finally, if an evaluation in $\boldsymbol{\lambda}_v\text{-}\mathbf{C}^{\triangleright}$ uses a top-level step, there is no equivalent step in the new reduction system. However, based on the above lemmas, we can show that the rest of the evaluation in $\boldsymbol{\lambda}_v\text{-}\mathbf{C}^{\triangleright}$ can be simulated, and that there is always a close relationship between the respective terms in the two sequences.

Lemma 3.7 *If* $e \overset{\triangleright}{\longmapsto}{}^*_c e'$ *with at least one* C_T*-step, then* $e \longrightarrow\!\!\!\!\!\rightarrow_d \mathcal{C}(\lambda k.e'_k)$, *where* e'_k *may be converted to* e' *by replacing all occurrences of* (ku) *with* u *for arbitrary values* u.

Proof. By assumption, the derivation for $e \overset{\triangleright}{\longmapsto}{}^*_c e'$ must contain a first step using C_T:

$$e \longmapsto^*_c \mathcal{C}e_1 \triangleright \underline{(e_1(\lambda x.\mathcal{A}^{\dagger}x))} \overset{\triangleright}{\longmapsto}{}^*_c e'.$$

We tag this first, newly-created halt continuation with a dagger † so that we can track it through the rest of the computation and distinguish its occurrence in the final answer.

By Lemma 3.5, the first part of the above derivation is easily simulated in the new system:

$$e \longrightarrow\!\!\!\!\!\rightarrow_d \mathcal{C}e_1 \longrightarrow_d \mathcal{C}(\lambda k.\underline{e_1(\lambda x.\mathcal{A}^{\dagger}(k\ x))}).$$

Since replacing (kx) with x in the underlined term yields the underlined term in the previous derivation, the underlined terms satisfy the desired relationship.

To complete the proof, it suffices to show that the invariant is preserved by all steps following the first top-level step. For this, we consider two cases.

1. Assume that the tagged continuation is not applied to a value during the rest of the computation. It is easy to see that in $\boldsymbol{\lambda}_v\text{-}\mathbf{C}^{\triangleright}$ the second half of the derivation,

$$e_1(\lambda x.\mathcal{A}^{\dagger}x) \overset{\triangleright}{\longmapsto}{}^*_c e',$$

can be transformed into the derivation

$$e_1 y \overset{\triangleright}{\longmapsto}{}_c^* e'_y \text{ where } e' \equiv e'_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger x)].$$

By Lemma 3.6, it follows that

$$\mathcal{C}(\lambda k.e_1 y) \longrightarrow\!\!\!\!\!\rightarrow_d \mathcal{C}(\lambda k.e'_y),$$

and, by replacing free y with $(\lambda x.\mathcal{A}^\dagger(k\ x))$,

$$\mathcal{C}(\lambda k.e_1(\lambda x.\mathcal{A}^\dagger(k\ x))) \longrightarrow\!\!\!\!\!\rightarrow_d \mathcal{C}(\lambda k.e'_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger(k\ x))]).$$

Again, a replacement of (kx) with x throughout $e'_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger(k\ x))]$ yields $e'_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger x)]$, which is e'.

2. Assume that the tagged continuation is applied to a value for a first, and last, time:

$$e_1(\lambda x.\mathcal{A}^\dagger x) \overset{\triangleright}{\longmapsto}{}_c^* E[(\lambda x.\mathcal{A}^\dagger x)v] \overset{\triangleright}{\longmapsto}{}_c \underline{E[\mathcal{A}^\dagger v]} \overset{\triangleright}{\longmapsto}{}_c^* e'.$$

By the same reasoning as in the first case, there must be an evaluation context E_y and a value v_y such that

$$\begin{aligned} \mathcal{C}(\lambda k.e_1(\lambda x.\mathcal{A}^\dagger(k\ x))) &\longrightarrow\!\!\!\!\!\rightarrow_d \mathcal{C}(\lambda k.E_y[yv_y][y \leftarrow (\lambda x.\mathcal{A}^\dagger(k\ x))]) \\ &\longrightarrow_d \mathcal{C}(\lambda k.\underline{E_y[\mathcal{A}^\dagger(kv_y)][y \leftarrow (\lambda x.\mathcal{A}^\dagger(k\ x))]}), \end{aligned}$$

where $E \equiv E_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger x)]$ and $v \equiv v_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger x)]$. Substituting v_y for (kv_y) and x for (kx) in the underlined term yields $E_y[(A^\dagger v_y)][y \leftarrow (\lambda x.\mathcal{A}^\dagger x)]$, which is the underlined term $E[A^\dagger v]$ above. The corresponding terms in the two derivation sequences still satisfy the desired invariant.

The rest of the standard computation sequence in $\boldsymbol{\lambda}_v$-$\mathbf{C}^\triangleright$ can only eliminate some or all of the evaluation context E. These steps are easily mimicked in the new calculus without violating the desired relationship. $\square$

$$\begin{aligned} fa &\longrightarrow \delta(f,a) && (\delta) \\ (\lambda x.e)v &\longrightarrow e[x \leftarrow v] && (\beta_v) \\ E^s[\mathcal{C}\ e] &\longrightarrow \mathcal{C}(\lambda k.e\ (\lambda x.\mathcal{A}(k\ E^s[x]))) && (C_{lift}) \\ \mathcal{C}\ e &\longrightarrow \mathcal{C}(\lambda k.e\ (\lambda x.\mathcal{A}(k\ x))) && (C_{top}) \\ \mathcal{C}(\lambda k.(\mathcal{C}\ e)) &\longrightarrow \mathcal{C}(\lambda k.e(\lambda x.\mathcal{A}x)) && (C_{idem}) \end{aligned}$$

Figure 1: The revised syntactic theory of control

In summary, Lemmas 3.5 through 3.7 show that if a program has a value according to $eval_c$, then the new calculus can reduce the program to a recognizably equivalent expression. The essential difference is that the reductions in $\boldsymbol{\lambda}_v$-$\mathbf{C(d)}$ "remember" whether or not the computation used any control operations. Thus, the answer in the new theory may be a simple value, v, a $\mathcal{C}$-application that abstracts over a value, $\mathcal{C}(\lambda k.v)$, or a $\mathcal{C}$-application that abstracts over the application of a continuation variable to a value, $\mathcal{C}(\lambda k.k\ v)$. In the first case, v is the same answer the evaluation function $eval_c$ would produce; in the latter two cases, the body of the $\mathcal{C}$-application may be converted to the expected answer by replacing all occurrences of (kv) with v.

More importantly, the proofs of the above lemmas also show that the new calculus basically reduces programs to answers with standard reduction steps. More precisely, an evaluation with the new rules begins with standard reduction steps based on the relation **c**. If this yields a value, the evaluation is finished. If not, it reaches a $\mathcal{C}$-application, in which case it employs a *single* C_{top}-step, followed by a number of standard reduction steps based on **c** possibly intermingled with C_{idem} reductions on the *complete program*. If this yields a $\mathcal{C}$-application of the above form, the evaluation stops and produces an answer. We abstract this process in an evaluation function.

Definition 3.8. (**d**-*Evaluation*) Let $\mathcal{C}(\lambda k.e) \longmapsto_{idem} \mathcal{C}(\lambda k.e')$ if $(\mathcal{C}(\lambda k.e), \mathcal{C}(\lambda k.e')) \in C_{idem}$ or $e \longmapsto_c e'$. Let v and v_y be values such that $v \equiv v_y[y \leftarrow (\lambda x.\mathcal{A}x)]$.

A program e in Λ_c *evaluates* to the value v, $eval_d(e) \equiv v$, if and only if

- $e \longmapsto_c^* v$, or
- $e \longmapsto_c^* \mathcal{C}e' \longrightarrow_d \mathcal{C}(\lambda k.e'(\lambda x.\mathcal{A}^\dagger(k\ x))) \longmapsto_{idem}^* \mathcal{C}(\lambda k.kv_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger(k\ x))])$, or
- $e \longmapsto_c^* \mathcal{C}e' \longrightarrow_d \mathcal{C}(\lambda k.e'(\lambda x.\mathcal{A}^\dagger(k\ x))) \longmapsto_{idem}^* \mathcal{C}(\lambda k.v_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger(k\ x))])$.

Note: Once again we tag the first halt continuation in the above definition to distinguish its occurrences in the last term of the reduction.

Based on the above lemmas, it is easy to prove that the two evaluation functions, $eval_c$ and $eval_d$, are equivalent.

Theorem 3.9 *For $e \in \Lambda_c$, $eval_c(e) \equiv eval_d(e)$.*

Proof. By Lemmas 3.5 and 3.7, $\boldsymbol{\lambda}_v$-$\mathbf{C(d)}$ can simulate standard computations. A simple check of their proofs shows that the reduction steps in the new system indeed conform to Definition 3.8. — For the other direction, assume that $eval_d(e) = v$. Then, either $e \longmapsto_c^* e'$, in which case the conclusion is obviously true. Or,

$$e \longmapsto_c^* \mathcal{C}e' \longrightarrow_d \mathcal{C}(\lambda k.e'(\lambda x.\mathcal{A}^\dagger(k\ x))) \longmapsto_{idem}^* \mathcal{C}(\lambda k.kv_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger(k\ x))])$$

or

$$e \longmapsto_c^* \mathcal{C}e' \longrightarrow_d \mathcal{C}(\lambda k.e'(\lambda x.\mathcal{A}^\dagger(k\ x))) \longmapsto_{idem}^* \mathcal{C}(\lambda k.v_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger(k\ x))])$$

for an appropriate v_y. It is easy to see that in both cases,

$$e \longmapsto_c^* \mathcal{C}e' \overset{\triangleright}{\longmapsto}_c e'(\lambda x.\mathcal{A}^\dagger x) \overset{\triangleright}{\longmapsto}{}_c^* v_y[y \leftarrow (\lambda x.\mathcal{A}^\dagger x)].$$

Hence, $e \overset{\triangleright}{\longmapsto}{}_c^* v$ and $eval_c(e) = v$ as desired. □

More importantly, we can show that the theory $\boldsymbol{\lambda}_v$-$\mathbf{C(d)}$ can also prove all safe equations in the old theory of control. To establish the claim, we need a lemma for each direction. The safeness of the new proof rules can be established by straightforward calculations.

Lemma 3.10 $\boldsymbol{\lambda}_v$**-C-safe** $\vdash C_{idem}, C_{top}$.

Proof. The safeness of C_{idem} follows from a simple calculation. Let E be an arbitrary evaluation context. Then,

$$\begin{aligned}\boldsymbol{\lambda}_v\text{-}\mathbf{C}^{\triangleright} \vdash E[\mathcal{C}(\lambda k.\mathcal{C}e)] &= (\lambda k.\mathcal{C}e)K \qquad \text{for some term } K \text{ determined by } E\\ &= \mathcal{C}e[k \leftarrow K]\\ &= (e[k \leftarrow K])(\lambda x.\mathcal{A}x)\\ &= (\lambda k.e(\lambda x.\mathcal{A}x))K\\ &= E[\mathcal{C}(\lambda k.e(\lambda x.\mathcal{A}x))].\end{aligned}$$

Verification of the safeness of C_{top} is slightly more complicated:

$$\begin{aligned}\boldsymbol{\lambda}_v\text{-}\mathbf{C}^{\triangleright} \vdash E[\mathcal{C}e] &= eK \qquad \text{for some term } K \text{ determined by } E\\ &= e(\lambda x.\mathcal{A}(Kx)) \qquad (\dagger)\\ &= (\lambda k.e(\lambda x.\mathcal{A}(kx)))K\\ &= E[\mathcal{C}(\lambda k.e(\lambda x.\mathcal{A}(kx)))].\end{aligned}$$

Since the continuation K is an abstraction of the form $\lambda x.\mathcal{A}e$, the step $(\dagger)$ is a consequence of the following safe equality:

$$\boldsymbol{\lambda}_v\text{-}\mathbf{C}^{\triangleright} \vdash (\lambda x.\mathcal{A}(Kx)) = (\lambda x.\mathcal{A}(\mathcal{A}e)) = (\lambda x.\mathcal{A}e) = K.$$

The equation $\mathcal{A}(\mathcal{A}e) = \mathcal{A}e$ follows from the safeness of C_{idem}. □

Every safe equation is also an equation in the new theory $\boldsymbol{\lambda}_v$-**C**(**d**).

Lemma 3.11 *If* $\boldsymbol{\lambda}_v$**-C-safe** $\vdash e = e'$, *then* $\boldsymbol{\lambda}_v$-**C**(**d**) $\vdash e = e'$.

Proof. The proof requires several lemmas about the shape of proofs for safe equations. Since it only contributes insight into the old theory, the proof is explained in the appendix. □

The two preceding lemmas show that adding the two axioms C_{idem} and C_{top} to the theory $\boldsymbol{\lambda}_v$-**C**(**c**) provides an axiomatic characterization of the theory of safe equations.

Theorem 3.12 (Safeness) $\boldsymbol{\lambda}_v$-**C**(**d**) $\vdash e_1 = e_2$ *iff* $\boldsymbol{\lambda}_v$**-C-safe** $\vdash e_1 = e_2$

Proof. The theorem follows from Lemmas 3.10 and 3.11. □

An immediate consequence of this theorem is that all equations in the revised theory of control are operationally sound. In other words, two equal expressions are indistinguishable via $eval_d$ with respect to all Λ_C-contexts (in the sense of Definition 2.7).

Corollary 3.13 *If* $\boldsymbol{\lambda}_v$-**C**(**d**) $\vdash e_1 = e_2$ *then* $e_1 \simeq_c e_2$.

As to the classical properties of the new reduction **d**, we can show that it is Church-Rosser, which provides an alternative proof of $\boldsymbol{\lambda}_v$-**C**(**d**)'s soundness. The Church-Rosser property moreover shows that $\boldsymbol{\lambda}_v$-**C**(**d**) (and, by the proceeding theorem, $\boldsymbol{\lambda}_v$-**C-safe**) is a conservative extension of $\boldsymbol{\lambda}_v$.

Theorem 3.14 (Consistency) *The notion of reduction* **d** *is Church-Rosser. If* $e \twoheadrightarrow_d e_1$ *and* $e \twoheadrightarrow_d e_2$, *then there is* e' *such that* $e_1 \twoheadrightarrow_d e'$ *and* $e_2 \twoheadrightarrow_d e'$.

Proof. [2] The proof requires some generalizations of standard techniques. First, we define

[2] We gratefully acknowledge Erik Crank's help with this proof.

an alternative set of reduction rules:

$$
\begin{array}{rcll}
f a & \longrightarrow & \delta(f, a) & (\delta) \\
(\lambda x.e)v & \longrightarrow & e[x \leftarrow v] & (\beta_v) \\
E^s[\mathcal{C}(\lambda m.e)] & \longrightarrow & \mathcal{C}(\lambda k.e[m \leftarrow (\lambda x.\mathcal{A}(k\ E^s[x]))]) & (C'_{lift}) \\
\mathcal{C}(\lambda k.(\mathcal{C}(\lambda m.e))) & \longrightarrow & \mathcal{C}(\lambda k.e[m \leftarrow (\lambda x.\mathcal{A}x)]) & (C'_{idem}) \\
\mathcal{C}(\lambda m.e) & \longrightarrow & \mathcal{C}(\lambda k.e[m \leftarrow (\lambda x.\mathcal{A}(kx))]) & (C'_{top}) \\
\mathcal{C}\ e & \longrightarrow & \mathcal{C}(\lambda k.e\ (\lambda x.\mathcal{A}(k\ x))) & (C_{top})
\end{array}
$$

We refer to the new set of reductions as $\mathbf{d}'$. It is easy to show that $\mathbf{d}$ and $\mathbf{d}'$ are equivalent reduction relations, i.e., $\longrightarrow\!\!\!\!\!\rightarrow_d\ \vdash \mathbf{d}'$ and $\longrightarrow\!\!\!\!\!\rightarrow_{d'}\ \vdash \mathbf{d}$. Second, we show in several steps that the new system is Church-Rosser. The proof for

$$\mathbf{c}' = \delta \cup \beta \cup C'_{lift}$$

is a simple adaptation of the Church-Rosser proof for $\mathbf{c}$ [9]. It is also straightforward to prove that C'_{idem}, C_{top} and C'_{top} each directly satisfy the diamond property, and that they are therefore Church-Rosser. Next we combine the relations and use the Hindley-Rossen method [2:64–66] for proving the Church-Rosser property of the larger relations. This is straightforward for the union of $\mathbf{c}'$ and C'_{idem}, of β_v and C'_{top}, and of $\beta_v \cup C'_{top}$ and C_{top}. The final step requires us to show that the reductions based on $\mathbf{c}' \cup C'_{idem}$ and $C_{top} \cup C'_{top}$ commute. For this we use Barendregt's commutation lemma [2:65] for the transitive closure of relations and apply it to $\mathbf{c}' \cup C'_{idem}$ and a parallel one-step reduction relation based on $C_{top} \cup C'_{top}$. Based on this, it is easy to show that the reductions based on $\mathbf{c}' \cup C'_{idem}$ and $\beta_v \cup C_{top} \cup C'_{top}$ commute. Hence, the union, which is the reduction based on $\mathbf{d}'$ is Church-Rosser because both sub-relations are Church-Rosser. Since $\mathbf{d}$ and $\mathbf{d}'$ are equivalent as reductions, $\mathbf{d}$ is also Church-Rosser. □

The new theory of control also has standard reduction sequences, albeit non-traditional ones. To allow both C_{top} and other $\mathbf{d}$-reductions in standard reduction sequences, we must extend the set of evaluation contexts to a set of $\mathbf{d}$-evaluation contexts such that $\mathbf{c}$ standard reduction steps can take place *after* a $\mathcal{C}$-application reaches the top of the entire term. The rest of the definition is conventional.

Definition 3.15. (**d**-*Standard Reduction Relation;* **d**-*Standard Reduction Sequences*) Let the set of $\mathbf{d}$-evaluation contexts (E^d) be defined as follows:

$$E^d ::= E \mid \mathcal{C}(\lambda k.E).$$

The standard reduction relation maps e to e', $e \longmapsto_d e'$, if there is a $\mathbf{d}$-standard evaluation context E^d such that $e \equiv E^d[p]$, $e' \equiv E^d[q]$ for some $(p, q) \in \mathbf{d}$.

By adding the following clause to the definition of standard reduction sequences of the λ_v-calculus (Definition 2.4), we get the set of standard reduction sequences for $\boldsymbol{\lambda}_v$-$\mathbf{C}(\mathbf{d})$:

- If $e_1, \ldots, e_n$ is a standard reduction sequence, then so is $\mathcal{C}\ e_1, \ldots, \mathcal{C}\ e_n$.

Clearly, the standard reduction for $\mathbf{d}$ generalizes the standard reduction for $\mathbf{v}$ but is a *relation* instead of a function. The reduction theory based on $\mathbf{d}$ satisfies the same standardization theorem as conventional λ-calculi.

Theorem 3.16 (Standardization) *$e \longrightarrow\!\!\!\!\rightarrow_d e'$ if and only if there is a standard reduction sequence $e, \ldots, e'$.*

Proof. The proof is an adaptation of Plotkin's corresponding proof. □

Finally, we can show that the evaluation function is again determined by the transitive closure of the standard reduction *relation* based on **d**. Since the latter determines a relation but not a function, the statement of the theorem takes on a slightly peculiar form.

Theorem 3.17 (Evaluation) *$eval_d \subseteq \{(e, v) \mid e \longmapsto_d^* v,$ or $e \longmapsto_d^* \mathcal{C}(\lambda k.kv'),$ or $e \longmapsto_d^* \mathcal{C}(\lambda k.v'),$ where $v \equiv v'[y \leftarrow (\lambda x.\mathcal{A}x)]\}$*

Proof. The standard reduction relation obviously extends the relations $\longmapsto_c$ and $\longmapsto_{idem}$ from Definition 3.8. □

With this last theorem, we have explored all the conventional aspects of the connection between programming languages and calculi.

3.3 Extensions of the equational theory

It is not immediately obvious from the preceding discussion why the reduction C_{lift} must be restricted to capturing only *singular* evaluation contexts. Combined with C_{top}, which effectively captures *empty* evaluation contexts, the two relations serve to capture *arbitrary* evaluation contexts. Consequently, the following generalization of C_{lift} would seem to be a natural unification of C_{lift} and C_{top}:

$$E[\mathcal{C}\ e] \ \longrightarrow\ \mathcal{C}(\lambda k.e\ (\lambda x.\mathcal{A}\ (k\ E[x]))). \qquad (C_E)$$

This rule captures an arbitrary evaluation context in a single step and applies the subterm of the $\mathcal{C}$-application to an appropriate continuation.

The notion of reduction C_E subsumes C_{lift} and C_{top} as sub-relations, but the inverse is not true. Consider the term $u(v\ (\mathcal{C}\ e))$. Two uses of C_{lift} yield the term

$$\mathcal{C}(\lambda k.e\ (\lambda x.\mathcal{A}\ ((\lambda y.\mathcal{A}\ (k\ (u\ y)))(v\ x)))),$$

but a single application of C_E with $E \equiv u\ (v\ [\ \ ])$ produces

$$\mathcal{C}(\lambda k.e\ (\lambda x.\mathcal{A}\ (k\ (u\ (v\ x))))).$$

Both terms are in normal form and it is thus impossible to *prove* their equivalence in either $\boldsymbol{\lambda}_v$-$\mathbf{C}(\mathbf{d})$ or $\boldsymbol{\lambda}_v$-$\mathbf{C}(\mathbf{d})$ modified with C_E. In short, although C_E adds equational power to the calculus, it destroys the Church-Rosser property.

A second extension of the theory $\boldsymbol{\lambda}_v$-$\mathbf{C}(\mathbf{d})$ is based on the observation that the safe theory cannot simulate the evaluation in a perfect manner. In $\boldsymbol{\lambda}_v$-$\mathbf{C}(\mathbf{d})$, there are three different types of answers. First, an evaluation may simply yield a value. Second, an evaluation may abort some part of a computation and produce the answer $(\mathcal{C}\lambda k.v)$ (with v possibly containing k), which basically is an *exceptional* answer. Finally, the answer may have the shape $(\mathcal{C}\lambda k.kv)$. In this case, the program discovered the answer at some point in the evaluation and used a continuation to escape from the rest of the evaluation. If the answer does not contain any references to the captured continuation, it is uninteresting from an observational perspective that the program used a continuation for escaping from the evaluation process.

We could avoid the third kind of answer for an evaluation in $\boldsymbol{\lambda}_v$-$\mathbf{C(d)}$ by introducing an additional reduction that eliminates $\mathcal{C}$-applications when they have become superfluous:

$$\mathcal{C}(\lambda k.k\ e) \longrightarrow e \text{ if } k \notin FV(e) \qquad (C_{elim})$$

Unfortunately $\mathbf{d} \cup C_{elim}$ is *not* Church-Rosser. A counterexample is the C_{elim}-redex itself, which is also a C_{top}-redex. Whereas a C_{elim}-step yields e, a reduction with C_{top} followed by a β_v-step leads to $\mathcal{C}\lambda k.((\lambda x.\mathcal{A}\ (k\ x))\ e)$. Since e does not necessarily have a value, we cannot continue the reduction as necessary.

We leave unsolved the problem of finding an extended theory that includes C_E or C_{elim} and still satisfies the classical properties of reduction theories.

4 Theories of state

The extension of the λ_v-calculus to a theory of procedural abstraction and assignment requires two new syntactic constructs for the underlying term language. First, there is a need for *assignable* variables—also called *state* variables—that denote different values at different times. To distinguish the set of *assignable* variables from the set of *binding* variables of the simple λ_v-calculus, we rename the latter set $Vars_\lambda$ and refer to the former as $Vars_\sigma$, annotating elements according to their set-membership: $x_\lambda \in Vars_\lambda$ and $x_\sigma \in Vars_\sigma$. Since assignable variables do not denote fixed values, we do *not* use them as *values*.

Second, the extended language needs a construct for altering the value, or state, of an assignable variable. For this purpose we use the σ-capability, which is a new form of value, $(\sigma x_\sigma.e)$. A σ-capability is similar to a λ-abstraction, but instead of binding a variable in some expression, it represents the right to assign the variable a new value. Upon invocation, it globally alters the value of its variable and then continues with the evaluation of its subexpression or *body*. We refer to the extended language as Λ_σ.

The notions of substitution, contexts and evaluation contexts are adapted appropriately. The latter definition has the same shape as in the λ_v- and the λ_v-C-calculus framework, but denotes a subset of contexts over Λ_σ.

In the following subsection, we introduce the calculus of procedural abstraction and state [8]. Like the original calculus of control, the state calculus requires two kinds of term relations and, moreover, relies on further extensions of the language Λ_σ. We show in the second subsection that both program-level term relations as well as additional language extensions are superfluous. In addition, our new theory of state is a proper extension of the existing one.

4.1 A syntactic theory of state

According to β_v, the application of a procedural abstraction to an argument value is equivalent to the evaluation of the procedure body with all occurrences of the procedure parameter replaced by the argument. Given this, it is reasonable to expect that a reduction relation for procedures with *assignable* parameters replaces the assignable parameter with something that corresponds to the argument value. The traditional solution is to maintain an additional function that maps a parameter name to a value: a *store*. In our earlier report [8], we demonstrated that the store and its management can be incorporated into the term structure of the program. The key is to keep track of the substituted values via a *unique label* that is attached to the value before substitution. Based on this labeling scheme, an assignment can be simulated by replacing all values that are tagged with the same label by a different labeled value. The use of the value of an assignable variable requires stripping off the label of the labeled value. The deallocation (or garbage collection) of unusable storage happens automatically.

A complicating fact for the definition of the extended term language is the potential for circular (self-referential) values. For example, the expression $(\lambda x.(\sigma x.x)(\lambda y.x))0$ evaluates to a recursive function that returns itself upon application. To achieve canonicity in the representation of such values, we add labeled bullets of the shape $\bullet^x$ for all labels x. For convenience, we add σ-capabilities with labeled bullets in the variable position $(\sigma \bullet^x .e)$, which represent the result of substituting labeled values for free variables.

Following these preliminary remarks, we define the extension of Λ_σ to Λ_S with the following abstract syntax:

$$\begin{array}{rcl} e & ::= & v \mid (ee) \mid x_\sigma \mid v^x \mid \bullet^x \\ v & ::= & c \mid x_\lambda \mid (\lambda x_\lambda.e) \mid (\lambda x_\sigma.e) \mid (\sigma x_\sigma.e) \mid (\sigma \bullet^x .e) \end{array}$$

The set of labels is the set of assignable variables (used without subscript). When the distinction between assignable and binding variables is irrelevant or deducible from context, we omit the subscripts λ and σ from variables. As indicated above, the substitution of free variables in terms is adapted *mutatis mutandis* with one exception: $(\sigma x.e)[x \leftarrow v^l] = (\sigma \bullet^l .e[x \leftarrow v^l])$.

Since the labeling strategy is a textual representation of a store, we need to ensure that programs describe consistent *stores*. For example, every label should be attached to only one value and labeled bullets should be used only to indicate self-references; typical terms that violate these conditions are $(\lambda x.1)^y(\lambda x.0)^y$ and $\bullet^l$. To eliminate such terms without a corresponding store configuration, we impose three context-sensitive conditions on the term language and use the resulting language as the basis of the calculus:

(C1) an x-labeled bullet ($\bullet^x$) can only occur as a sub-term of an x-labeled value or in the variable-position of a σ-capability, and an x-labeled value must not contain x-labeled values, only x-labeled bullets;

(C2) the bound variable of a λ-abstraction must not occur as a sub-term in a labeled value;

(C3) the labeling of the two subexpressions in an application must be consistent: if v^x is a subterm of e and u^x is a subterm of e' where $(e\ e')$ is an application, then v and u must be identical after replacing labeled values in them by the labels.

Equipped with the notion of labeled terms, we introduce *labeled-value substitution*, $e[\bullet^x \leftarrow v^x]$, which replaces all x-labeled values in an expression e by v^x such that the resulting expression respects the above conditions. This may involve replacing labeled subvalues by labeled bullets in v.

Next, we can turn to the question of how to simulate the execution of a Λ_σ program through reductions of Λ_S-terms. For an example, we consider the application of an abstraction with an assignable parameter to a value. We would like to model this effect with a substitution of the parameter by a labeled value. Since the label must be unique for every reduction of such an application, it is impossible to perform several reductions in parallel in different parts of the term.

The coordination of the effects of labeled-value substitutions becomes possible by ensuring that only one such contraction is applicable. Since the only unique point in a term is the root, the calculus again coordinates imperative effects of transition steps by splitting the set of term relations into a set of simple notions of reduction and computation rules. The reductions lift a redex to the top of the program where the computation rules perform the appropriate action. There are three kinds of redexes that require *unique* actions:

1. the application of a procedural abstraction with an assignable variable to a value, $(\lambda x_\sigma.e)v$;
2. the application of a σ-capability to a value, $(\sigma \bullet^x .e)v$, which must proceed with the evaluation of the body after replacing *all* occurrences of u^x with v^x in the *entire* program; and
3. the use of a labeled value, v^x, which produces the value $v[\bullet^x \leftarrow v^x]$.

According to the above reasoning, such redexes must be lifted to the top of the program just before they are evaluated. Consequently, the reductions must lift the redexes out of *evaluation* contexts, and, after applying the appropriate computation rule, the evaluation must continue with the expression in the hole of the original evaluation context. Putting all of this together, we introduce the following notions of reductions where the meta-variables X ranges over assignable variables, labeled values and labeled bullets (depending on context):

$$\begin{aligned} E[((\lambda x_\sigma.e)v)] &\longrightarrow (\lambda x_\sigma.E[e])v && (\beta_E) \\ E[((\sigma X.e)v)] &\longrightarrow (\sigma X.E[e])v && (\sigma_E) \\ E[X] &\longrightarrow (\lambda v.E[v])X && (D_E) \end{aligned}$$

In accord with the variable assumptions in Section 2, we assume in these equations that variables are renamed as necessary to avoid conflicts.

Once redexes reach the top of the program, the appropriate action must take place. For the simulation of these in a term rewriting system, we define the following computation rules:

$$\begin{aligned} (\lambda x_\sigma.e)v &\rhd e[x_\sigma \leftarrow v^y] \text{ where } y \notin FV(e,v) && (\beta_T) \\ (\sigma \bullet^x .e)v &\rhd e[\bullet^x \leftarrow v^x] && (\sigma_T) \\ (uv^x) &\rhd u(v[\bullet^x \leftarrow v^x]) && (D_T) \end{aligned}$$

Notice that $(\sigma x.e)v$ is a redex for the reduction relations but not for the computation relations: in Λ_σ assignments can only be made to bound variables (which are replaced by labeled variables in time).

We define the calculus of state in the same way as the calculus of control. The basic notion of reduction is

$$\mathbf{s} = \mathbf{v} \cup \beta_E \cup \sigma_E \cup D_E.$$

When terms are equal according to $\mathbf{s}$, $e =_s e'$, we write $\boldsymbol{\lambda}_v\text{-}\mathbf{S}(\mathbf{s}) \vdash e = e'$. As usual, $\longrightarrow_s$ and $\longrightarrow\!\!\!\!\!\rightarrow_s$ denote the one-step reduction and its transitive closure. The computation relation is defined by:

$$\triangleright_s = \longrightarrow\!\!\!\!\!\rightarrow_s \cup \beta_T \cup \sigma_T \cup D_T.$$

The relation $\stackrel{\triangleright}{=}_s$ is the smallest equivalence relation generated by the computation relation $\triangleright_s$. We denote equivalences in this theory by $\boldsymbol{\lambda}_v\text{-}\mathbf{S}^{\triangleright} \vdash e = e'$.

The syntactic theory of state satisfies the same variants of the classical properties as the syntactic theory of control. Its sub-theory based on the relation $\mathbf{s}$ is Church-Rosser and the computation relation satisfies the diamond property. There are standard reduction sequences for the reduction relation and standard computation sequences for the computation relation. As above, we denote the standard reduction function with $\stackrel{\triangleright}{\longmapsto}_s$. Most importantly, a subset of the standard computation mapping defines an evaluation function.

Definition 4.1. ($\mathbf{s}$-*Standard Computation Function*) The *standard computation function* maps a program e to a program e', $e \stackrel{\triangleright}{\longmapsto}_s e'$, if e standard reduces to e' or if e computes to e': $\stackrel{\triangleright}{\longmapsto}_s = \longmapsto_s \cup \beta_T \cup \sigma_T \cup D_T$.

Now the evaluation function on Λ_σ (and Λ_S), $eval_s$, can again be defined as the transitive closure of the standard computation relation:

$$eval_s(e) = v \quad \text{if} \quad e \stackrel{\triangleright}{\longmapsto}_s^* v.$$

Mutatis mutandis, this definition induces an operational equivalence relation for Λ_σ ($\simeq_\sigma$) (along the lines of Definition 2.7). Most importantly, we can prove that equations between Λ_σ terms in the calculus are safe and imply operational equivalence.

Theorem 4.2 ([8]) *Let e and e' be in Λ_σ. If $\boldsymbol{\lambda}_v\text{-}\mathbf{S}^{\triangleright} \vdash e = e'$ then $e \simeq_\sigma e'$.*

Unfortunately, the theory $\boldsymbol{\lambda}_v\text{-}\mathbf{S}^{\triangleright}$ is not compatible with respect to equations over Λ_σ terms. For example,

$$\boldsymbol{\lambda}_v\text{-}\mathbf{S}^{\triangleright} \vdash (\lambda x.(\sigma x.2)1)0 = 2,$$

yet,

$$\boldsymbol{\lambda}_v\text{-}\mathbf{S}^{\triangleright} \not\vdash \lambda y.(\lambda x.(\sigma x.2)1)0 = \lambda y.2.$$

In the second equation, the top-level steps that are crucial for evaluating assignments can no longer be performed because the expressions are embedded inside of λ-abstractions. We could solve this problem by introducing an extended theory of safe equations as in the old theory of control, but fortunately, there is a better solution for this problem.

4.2 The revised syntactic theory of state

The crucial insight that leads to an improved theory of state originates from a simple observation about the context-sensitive restrictions of the language Λ_S. The motivation for the restrictions is the existence of terms in the unconstrained language that do not represent an intermediate consistent store in the evaluation of a Λ_σ-term. The context-sensitive restrictions eliminate such terms.

Lemma 4.3 *For every term $e \in \Lambda_S$ there is a term $e' \in \Lambda_\sigma$ such that $e' \triangleright_s^* e$.*

Proof. By the context-sensitive restrictions on Λ_S, for every term $e \in \Lambda_S$ with labels $x_1, \ldots, x_n$, there is a term $e'' \in \Lambda_\sigma$ with free assignable variables $x_1, \ldots, x_n$ and values $u_1, \ldots, u_n$ in Λ_σ such that

$$e \equiv e'' \ldots [x_i \leftarrow (\lambda x.x)^{x_i}] \ldots [\bullet^{x_i} \leftarrow (u_i \ldots [x_i \leftarrow (\lambda x.x)^{x_i}] \ldots)^{x_i}] \ldots$$

First, for every label x_i in e, there is a unique value u_i that corresponds to the collection of x_i-labeled values. By condition (C3), we can construct this value by replacing all labeled values with their labels in an *arbitrary* x-labeled value (not a bullet!). This algorithm produces the values u_1 through u_n. Second, we can also obtain e'' by replacing all occurrences of a labeled value with its label. By construction, the terms $e'', u_1, \ldots, u_n$ satisfy the above condition. We can now take

$$e' \equiv (\lambda x_1 \ldots x_n.(\sigma x_1 \ldots x_n.e'')u_1 \ldots u_n)(\lambda x.x) \ldots (\lambda x.x),$$

which proves the proposition. □

In order to simplify the presentation of terms like e' in the preceding lemma, we introduce a simplified version of Landin's [11] **letrec**-abbreviation: the ρ-application.[3] A ρ-application is a combination of a finite function from assignable variables to values, represented as a set $\theta = \{(x_1, v_1), \ldots, (x_n, v_n)\}$, and an expression e; it expands according to the construction in the lemma:

$$\rho\{(x_1, u_1), \ldots, (x_n, u_n)\}.e \stackrel{df}{\equiv} (\lambda x_1 \ldots x_n.(\sigma x_1 \ldots x_n.e)u_1 \ldots u_n)(\lambda x.x) \ldots (\lambda x.x).$$

The set notation is justified since all expansions corresponding to some linear arrangement of the set clearly reduce to the same term in $\boldsymbol{\lambda}_v$-$\mathbf{S}^{\triangleright}$. When we write $\rho\theta \cup \theta'.e$, we assume that $\theta \cup \theta'$ is a finite function. Finally, we define $\rho\emptyset.e \equiv e$. We use $Dom(\theta)$ to denote the set of defined variables, $\{x_1, \ldots, x_n\}$, in the function θ.

It also follows from the above Lemma that every theorem $e_1 = e_2$ in $\boldsymbol{\lambda}_v$-$\mathbf{S}^{\triangleright}$ for $e_1, e_2 \in \Lambda_S$ implies the existence of a theorem

$$\rho\theta_1.e_1' = \rho\theta_2.e_2'$$

[3] Recently, Abadi *et al.* [1] proposed and studied a variant of the λ-calculus that incorporates explicit substitutions. Our ρ-applications correspond to their closures: in the notation of Abadi *et al.* $\rho\theta.e$ would be the term $e[\theta]$ for a non-recursive θ. In other words, our ρ-applications generalize their notion of closure to the more common notion of Scheme- and ML-like closures whose lexical variables may be bound to recursive values.

for some $\rho\theta_1.e'_1, \rho\theta_2.e'_2 \in \Lambda_\sigma$. This holds, in particular, for the computation rules, which we would like to eliminate. Assuming that no labeled value gets lost during a transition, the reformulation of the top-level relations yields the following set of term relations:

$$\begin{aligned} \rho\theta.((\lambda x_\sigma.e)v) &\longrightarrow \rho\theta \cup \{(x,v)\}.e \\ \rho\theta \cup \{(x,v)\}.(ux) &\longrightarrow \rho\theta \cup \{(x,v)\}.(uv) \\ \rho\theta \cup \{(x,u)\}.((\sigma x.e)v) &\longrightarrow \rho\theta \cup \{(x,v)\}.e \end{aligned}$$

The first rule says that a β_T-transition creates a new entry in the ρ-application. The second rule specifies that the use of an assignable variable corresponds to a lookup of the variable in the ρ-application-set. And finally, the assignment is a modification of one pair in the set.

In short, the set of the (global) ρ-application acts like a *store*, and the translation of the computation rules have the appropriate effects on the finite store. More importantly, these rules are completely independent of the context in which they occur. They do not rely on the uniqueness of new variables, have no effect on the context, and the lookup is relative to the closest (part of the) store in the term. Hence, there is no further need for coordinating these rules, and we may as well take these relations as notions of reduction.

Unfortunately, the above rules are not quite strong enough to replace the computation rules in the preceding subsection. The assumption that a transition does not loose labeled values is too strong. If, for example, a bound assignable variable does not occur in the procedure body, the corresponding instance of β_T would translate as

$$\rho\theta.((\lambda x_\sigma.e)v) \longrightarrow \rho\theta.e,$$

or even

$$\rho\theta.((\lambda x_\sigma.e)v) \longrightarrow \rho\theta'.e, \qquad \theta' \subset \theta$$

if v contains the last reference to some other assignable variables. In general, the right hand side of the new reductions may contain variables in the store of the ρ-application that are no longer relevant to the evaluation of the body. These variables and their associated values are *garbage* and can be discarded. Whereas garbage collection is *automatic* in $\boldsymbol{\lambda}_v$-$\mathbf{S}^{\triangleright}$, we need to introduce an explicit garbage collection rule for the new system:

$$\rho\theta_0 \cup \theta_1.e \;\longrightarrow\; \rho\theta_1.e \text{ if } \theta_0 \neq \emptyset \text{ and } Dom(\theta_0) \cap FV(\rho\theta_1.e) = \emptyset. \qquad (gc)$$

We have summarized the revised theory of state in Figure 2. The rules in the figure slightly differ from the rules developed above. In order to reduce the number of reductions, we have merged the λ_E-, σ_E- and D_E-rules with the replacements for the computation rules. This also requires a new term relation, $\rho_\cup$, for merging two ρ-applications, which would otherwise be the effect of the lifting rules. The basic reduction relation for the new calculus is

$$\mathbf{t} = \mathbf{v} \cup \beta_\sigma \cup D \cup \sigma \cup gc \cup \rho_\cup.$$

The new theory is referred to as $\boldsymbol{\lambda}_v$-$\mathbf{S}(\mathbf{t})$. With Lemma 4.3 and the garbage collection rule, we can show that the new set of rules is a complete replacement for the computation rules.

Lemma 4.4 *Let $e_1, e_2 \in \Lambda_S$ and let $\rho\theta_1.e'_1, \rho\theta_2.e'_2 \in \Lambda_\sigma$ be their counterparts according to Lemma 4.3. If $\boldsymbol{\lambda}_v$-$\mathbf{S}^{\triangleright} \vdash e_1 \triangleright_s e_2$ then $\boldsymbol{\lambda}_v$-$\mathbf{S}(\mathbf{t}), \beta_E, D_E, \sigma_E \vdash \rho\theta_1.e'_1 \longrightarrow\!\!\!\!\!\rightarrow \rho\theta_2.e'_2$.*

$$
\begin{array}{rcll}
fa & \longrightarrow & \delta(f,a) & (\delta) \\
(\lambda x_\lambda.e)v & \longrightarrow & e[x_\lambda \leftarrow v] & (\beta_v) \\
(\lambda x_\sigma.e)v & \longrightarrow & \rho\{(x_\sigma,v)\}.e & (\beta_\sigma) \\
\rho\theta\cup\{(x,v)\}.E[x] & \longrightarrow & \rho\theta\cup\{(x,v)\}.E[v] & (D) \\
\rho\theta\cup\{(x,u)\}.E[(\sigma x.e)v] & \longrightarrow & \rho\theta\cup\{(x,v)\}.E[e] & (\sigma) \\
\rho\theta_0\cup\theta_1.e & \longrightarrow & \rho\theta_1.e \text{ if } \theta_0\neq\emptyset \text{ and } Dom(\theta_0)\cap FV(\rho\theta_1.e)=\emptyset & (gc) \\
\rho\theta.E[\rho\theta'.e] & \longrightarrow & \rho\theta\cup\theta'.E[e] \text{ if } \theta'\neq\emptyset \text{ and } \rho\theta.E\neq[\ ] & (\rho_\cup)
\end{array}
$$

Figure 2: The revised syntactic theory of state

Proof. The proof relies on two facts about the construction in Lemma 4.3:

1. The algorithm for converting e to $\rho\theta.e'$ does not alter the structure of the term e except for replacing labeled values by labels. In particular, values remain values and non-values remain non-values.

2. The labeled values in e that are moved into the store of the program $\rho\theta.e'$ preserve their structure in the same way.

As a consequence, a redex in e_1 not inside of a labeled value becomes a redex at the homologous position in e_1'. More specifically, **s**-redexes becomes **s**-redexes and β_T-, σ_T-, and D_T-redexes become instances of β_σ-, σ-, and D-redexes, respectively. Similarly, **s**-redexes inside of labeled values in e_1 become **s**-redexes inside of the values in the store of $\rho\theta_1.e_1'$ that directly contain the redex (a labeled value directly contains a subexpression if there is not a labeled sub-value that contains the subexpression).

Given these preliminaries, it is easy to see that, given a reduction in $\boldsymbol{\lambda}_v$-$\mathbf{S}^{\triangleright}$, a reduction in $\boldsymbol{\lambda}_v$-$\mathbf{S(t)}$ of the corresponding redex in $\rho\theta_1.e_1'$ leads to a term $\rho\theta^*.e_2'$. Clearly, neither **s**- nor **t**-redexes create new free variables but the substitution process associated with **s**-redexes may eliminate some labels by vacuous substitutions. On the other hand, the corresponding **t**-redexes will eliminate the corresponding variables. Hence,

$$\rho\theta^*.e_2' \equiv \rho\theta_2\cup\theta.e_2'$$

such that, by the construction of $\rho\theta_2.e_2'$,

$$Dom(\theta)\cap FV(\rho\theta_2.e_2') = \emptyset.$$

This permits an application of the garbage collection rule, gc, and we get

$$\rho\theta_1.e_1' \longrightarrow \rho\theta^*.e_2' \longrightarrow \rho\theta_2.e_2'. \qquad \square$$

The lemma implies that the new theory, extended with the lifting reductions, can prove all the equations on Λ_σ that the old theory can prove.

Theorem 4.5 *Let $e, e' \in \Lambda_\sigma$.*

(i) *If* $\boldsymbol{\lambda}_v$-$\mathbf{S}^{\triangleright} \vdash e = e'$ *then* $\boldsymbol{\lambda}_v$-$\mathbf{S(t)}, \beta_E, D_E, \sigma_E \vdash e = e'$.

(ii) *The converse does not hold.*

Proof. (i) By the diamond property, $\boldsymbol{\lambda}_v$-$\mathbf{S}^{\triangleright} \vdash e = e'$ implies that there is a term e^* such that $\boldsymbol{\lambda}_v$-$\mathbf{S}^{\triangleright} \vdash e \triangleright_s^* e^*$ and $\boldsymbol{\lambda}_v$-$\mathbf{S}^{\triangleright} \vdash e' \triangleright_s^* e^*$. It follows from Lemma 4.4 that $\boldsymbol{\lambda}_v$-$\mathbf{S(t)}, \beta_E, \sigma_E, D_E \vdash e \longrightarrow\!\!\!\!\!\rightarrow e^*$ and $\boldsymbol{\lambda}_v$-$\mathbf{S(t)}, \beta_E, \sigma_E, D_E \vdash e' \longrightarrow\!\!\!\!\!\rightarrow e^*$. Therefore, $\boldsymbol{\lambda}_v$-$\mathbf{S(t)}, \beta_E, D_E, \sigma_E \vdash e = e'$.

(ii) Here is a simple proof in $\boldsymbol{\lambda}_v$-$\mathbf{S(t)}$:

$$\boldsymbol{\lambda}_v\text{-}\mathbf{S(t)} \vdash \lambda y.(\lambda x.(\sigma x.2)1)0 = \lambda y.\rho\{(x,0)\}.(\sigma x.2)1 = \lambda y.\rho\{(x,1)\}.2 = \lambda y.2$$

As explained at the end of the previous subsection, the resulting theorem is not provable in the old theory. □

A second important consequence of Lemma 4.4 is that the reduction theory based on **t** alone can simulate the evaluation of Λ_σ-programs.

Lemma 4.6 *Let* $e \in \Lambda_\sigma$. *If* $eval_s(e) = v$ *for some value* $v \in \Lambda_S$ *then* $e \longrightarrow\!\!\!\!\!\rightarrow_t \rho\theta.v'$ *where* $\rho\theta.v' \in \Lambda_\sigma$ *is the counterpart of* v *according to Lemma 4.3.*

Proof. If $eval_s(e) = v$ for some value $v \in \Lambda_S$, then $e \overset{\triangleright}{\longmapsto}{}_s^* v$. In such a series of standard computation steps, subsequences of standard reduction steps according to β_E, σ_E, and D_E are always followed by standard computation steps according to β_T, σ_T, and D_T, respectively; the latter always precedes a β_v step, which puts the de-labeled value into the original evaluation context. In other words, β_E, σ_E, and D_E in standard computations only occur in clusters that, by Church-Rosser and diamond property, are equivalent to the following three cases:

1. $E[(\lambda x.e)v] \longmapsto_s (\lambda x.E[e])v \triangleright_s E[e[x \leftarrow v^l]]$
2. $E[v^l] \longmapsto_s (\lambda x.E[x])v^l \triangleright_s (\lambda x.E[x])v[\bullet^l \leftarrow v^l] \longmapsto_s E[v[\bullet^l \leftarrow v^l]]$
3. $E[(\sigma x.e)v] \longmapsto_s (\sigma x.E[e])v \triangleright_s E[e][\bullet^l \leftarrow v^l]$

Translating these kinds of sequences into the new calculus according to Lemma 4.4, merges them as $\beta_\sigma/\rho_\cup$, D, and σ steps:

1. $\rho\theta.E[(\lambda x.e)v] \longrightarrow_t \rho\theta.E[\rho\{(x,v)\}.e] \longrightarrow_t \rho\theta \cup \{(x,v)\}.E[e]$
2. $\rho\theta.E[l] \longrightarrow_t \rho\theta.E[\theta(l)]$
3. $\rho\theta \cup \{(x,u)\}.E[(\sigma x.e)v] \longrightarrow_t \rho\theta \cup \{(x,v)\}.E[e]$

In short, the translation incorporates preliminary lifting reductions into the simulated top-level steps. But then the derivation in the extended theory no longer uses any lifting steps, i.e., $\boldsymbol{\lambda}_v$-$\mathbf{S(t)} \vdash e \longrightarrow\!\!\!\!\!\rightarrow \rho\theta.v'$. □

Based on this lemma, we can now define an evaluation function using only **t** reductions. The main idea behind the definition is that programs can maintain a textual representation of the store in the form of a ρ-application at the root of the program.

Definition 4.7. (**t**-*Evaluation*) Let $e \longmapsto_{t1} e'$ if

1. $e \equiv \rho\theta.M$, $M \longmapsto_v M'$, and $e' \equiv \rho\theta.M'$, or

2. $(e, e') \in (\sigma \cup D \cup \mathbf{gc} \cup (\rho_{\cup} \circ \beta_\sigma))$, where $(\rho_{\cup} \circ \beta_\sigma)$ is the composition of β_σ and $\rho_{\cup}$, i.e., a β_σ-step followed by a $\rho_{\cup}$-step.

A program e in Λ_σ *evaluates* to the answer $\rho\theta.v$, $eval_t(e) \equiv \rho\theta.v$, if $e \longmapsto^*_{t1} \rho\theta.v$ and there is no e' such that $\rho\theta.v \longmapsto_{t1} e'$.

The single-step evaluation relation ($\longmapsto_{t1}$) is a proper relation because of its non-deterministic use of garbage collection. On the other hand, by demanding complete garbage collection, $eval_t$ becomes a (partial) function on Λ_σ programs. Moreover, it is equivalent to the old evaluation function.

Theorem 4.8 *Let $e, \rho\theta.v \in \Lambda_\sigma$, $v' \in \Lambda_S$, and assume that $\rho\theta.v' \overset{\triangleright}{\longmapsto}{}^*_s v$. Then, $eval_s(e) = v'$ if and only if $eval_t(e) = \rho\theta.v$.*

Proof. A simple check of Lemma 4.6 shows that the left to right direction is built into Definition 4.7, and that the arguments are invertible. $\square$

Since, unlike in the case of control, the new theory extends the old theory, we cannot prove the soundness of the new theory via the old one. Instead, we must assert some classical properties first.

First, the theory is Church-Rosser.

Theorem 4.9 (Consistency) *The notion of reduction* $\mathbf{t}$ *is Church-Rosser. If $e \longrightarrow\!\!\!\!\to_t e_1$ and $e \longrightarrow\!\!\!\!\to_t e_2$, then there is e' such that $e_1 \longrightarrow\!\!\!\!\to_t e'$ and $e_2 \longrightarrow\!\!\!\!\to_t e'$.*

Proof. The classical methods for Church-Rosser proofs for untyped λ-calculi apply. $\square$

Second, we can define a standard reduction relation and a set of standard reduction sequences for $\boldsymbol{\lambda}_v$-$\mathbf{S(t)}$.

Definition 4.10. (**t**-*Standard Reduction Relation; Standard Reduction Sequences*) The definition of $\longmapsto_t$ is based on a set of **t**-*standard evaluation contexts*, E:

$$E ::= E \mid \rho\theta.E.$$

The standard reduction relation maps e to e', $e \longmapsto_t e'$, if there is a **t**-standard evaluation context E such that $e \equiv E[p]$, $e' \equiv E[q]$ for some $(p, q) \in \mathbf{t}$.

By adding the following clause to the definition of standard reduction sequences of the λ_v-calculus (Definition 2.4), we get the set of standard reduction sequences for $\boldsymbol{\lambda}_v$-$\mathbf{S(t)}$:

- If $e_1, \ldots, e_n$ is a standard reduction sequence, then so is $\sigma x.e_1, \ldots, \sigma x.e_n$.

Third, the new theory of state satisfies the usual standardization theorem.

Theorem 4.11 (Standardization) *$e \longrightarrow\!\!\!\!\rightarrow_t e'$ if and only if there is a standard reduction sequence $e, \ldots, e'$.*

Proof. The proof is an adaptation of Plotkin's corresponding proof. □

Finally, we are ready to prove that the new theory is sound. We do this in two steps.

Theorem 4.12 (Evaluation) *Let $e, \rho\theta.v$ be in Λ_σ.*

(i) *If $eval_t(e) = \rho\theta.v$, then $e \longmapsto_t^* \rho\theta.v$.*

(ii) *If $e \longmapsto_t^* \rho\theta.v$, then there exists $\rho\theta'.v'$ such that $eval_t(e) = \rho\theta'.v'$.*

Proof. (i) The relation $\longmapsto_{t1}$ is clearly a subset of the standard reduction relation, in which all non-**v** steps are restricted to the root of the program.

(ii) The relation $\longmapsto_t$ generalizes $\longmapsto_{t1}$ such that all reductions can be performed inside of a program as well as at its root. Moreover, it disconnects the relation $\rho_\cup \circ \beta_\sigma$ such that $\rho_\cup$- and β_σ-reductions can be separated. However, it is also easy to see that a sequence of $\longmapsto_t$ steps can be rearranged so that all ρ-applications are merged with the top-level ρ-applications as soon as they occur in an evaluation context. Clearly, such rearranged sequences are still standard reduction sequences, and more importantly, they are also sequences of $\longmapsto_{t1}$-steps. The difference between the two answers is that a standard reduction sequence does not assume that all garbage is eliminated whereas the evaluation function insists on this. □

Now, recall that two Λ_σ expressions e and e' are operationally equivalent, $e \simeq_\sigma e'$, if and only if they are indistinguishable relative to all Λ_σ program contexts (in the sense of Definition 2.7). The final theorem says that the new calculus is operationally sound in the sense that two expressions are equivalent in the calculus only if they are operationally equivalent.

Theorem 4.13 *If $\boldsymbol{\lambda}_v$-$\mathbf{S(t)} \vdash e = e'$ then $e \simeq_\sigma e'$.*

Proof. Since $\boldsymbol{\lambda}_v$-$\mathbf{S(t)}$ is a conventional calculus, $\boldsymbol{\lambda}_v$-$\mathbf{S(t)} \vdash e = e'$ implies $\boldsymbol{\lambda}_v$-$\mathbf{S(t)} \vdash C[e] = C[e']$ for all contexts C. Now assume that for some context C, $eval_t(C[e])$ terminates. By the Standardization Theorem 4.11, $C[e] \longmapsto_t^* \rho\theta.v$ and therefore $\boldsymbol{\lambda}_v$-$\mathbf{S(t)} \vdash C[e'] = C[e] = v$. By the Consistency Theorem 4.9 and the Evaluation Theorem 4.12, $C[e'] \longmapsto_{t1}^* \rho\theta'.v'$ and therefore $eval_t(C[e'])$ is defined too. By symmetry, $C[e]$ terminates if and only if $C[e']$ terminates.

For the second condition, assume that $eval_t(C[e]) = c$ and $eval_t(C[e']) = d$ for constants c and d. Then, by Lemma 4.6, $\boldsymbol{\lambda}_v$-$\mathbf{S(t)} \vdash C[e] \longrightarrow\!\!\!\!\rightarrow c$ and $\boldsymbol{\lambda}_v$-$\mathbf{S(t)} \vdash C[e'] \longrightarrow\!\!\!\!\rightarrow d$. Hence, $c = C[e] = C[e'] = d$. Again by the Consistency Theorem, $c = d$, which proves that $e \simeq_\sigma e'$. □

In summary, the new theory of state based on the reduction **t** is the essential calculus of state. It can evaluate programs (4.8); it is consistent (4.9); it has standard reduction sequences whose standard reduction relation is an evaluation mechanism (4.12); and it is sound (4.13). Finally, it also extends the old theory (4.5).

Note: The nature of variables

From Scheme's [19, 21] practical point of view, the new theory only contains one disturbing element, namely, the partitioning of the variable set into *binding* and *assignable* variables. The reason for this separation is the desire to use variables as values as in $\boldsymbol{\lambda}_v$. However, in a language with assignments variables no longer stand for one value but for a series of values. Consequently, they should *not* be considered as values but as expressions that always have a value. By excluding the set of variables from values, the distinction between the two variable sets becomes superfluous and the language becomes Scheme-like:

$$\begin{array}{rcl} e & ::= & x \mid v \mid (e\ e) \\ v & ::= & \lambda x.e \mid \sigma x.e. \end{array}$$

A revised calculus only requires a single axiom for parameter-passing, namely, β_σ. The only loss of this modified theory is that it is no longer a conservative extension of the original λ_v-calculus.

On the other hand, such a revised calculus easily accommodates another reduction that simplifies work with the calculus. In the revised calculus a variable is said to be assignable if it occurs in the variable position of a σ-capability. When a variable in a ρ-set is no longer assignable, the new calculus can replace the variable with its recursive value:

$$\begin{array}{lcl} \rho\theta \cup \{(x, v)\}.e & \longrightarrow & (\rho\theta.e)[x \leftarrow v[x \leftarrow \mathbf{Y}(\lambda x.v)]] \\ & & \text{if } x \text{ is not assignable in } e,\ v, \text{ and } \theta \\ & & \text{and where } \mathbf{Y} \stackrel{df}{\equiv} (\lambda f y.(\lambda x.xx)(\lambda x.f(\lambda y.(xx)y))y). \end{array} \qquad (\rho_Y)$$

A restricted version of β_v can be derived from ρ_Y.

5 Unified theories of control and state

The original theories of control and state are completely orthogonal to each other [5]. The sum of the extended notions of reduction yields a theory for a language with facilities for both control and state manipulation; indeed, the *shape* of the reduction relations as pattern-matching rules stays the same. As a result, the larger theory contains the theories of procedural abstraction, control and state as sub-sets.

In our new framework, a simple merger is insufficient, since a $\mathcal{C}$-application may block variable references and assignments. Thus we must introduce an additional notion of reduction to move $\mathcal{C}$-applications outside of ρ-applications:

$$\rho\theta.\mathcal{C}e \longrightarrow \mathcal{C}\rho\theta.e \qquad (\rho\mathcal{C})$$

Let $\Lambda_{C\sigma}$ stand for the merged language:

$$\begin{array}{rcl} e & ::= & v \mid (e\ e) \mid (\mathcal{C}\ e) \mid x_\sigma \\ v & ::= & b \mid f \mid x_\lambda \mid \lambda x.e \mid \sigma x.e \end{array}$$

Furthermore, let $\mathbf{d}'$ and $\mathbf{t}'$ stand for the extension of the notions of reduction $\mathbf{d}$ and $\mathbf{t}$ to $\Lambda_{C\sigma}$. The new theory of control and state is based on the union of these reductions with $\rho\mathcal{C}$:

$$\mathbf{cs} = \mathbf{d}' \cup \mathbf{t}' \cup \rho\mathcal{C}.$$

Most importantly, the new notion of reduction is syntactically consistent.

Theorem 5.1 *The extended notion of reduction* **cs** *is Church-Rosser.*

Proof. All three parts of the relation satisfy the Church-Rosser property. The proof that the union does is a straightforward generalization of the Hindley-Rossen method [2:ch3]. □

As a consequence, the larger theory contains the theories $\boldsymbol{\lambda}_v$-$\mathbf{C(d)}$ and $\boldsymbol{\lambda}_v$-$\mathbf{S(t)}$ as subsets. Evaluation can be defined for the larger theory. A program p evaluates to q in the new theory if and only if p evaluates to a value v in the old theory, where q is of the form v' or $\mathcal{C}\lambda k.v'_k$, v'_k can be converted to v' by replacing all occurrences of (ku) with u as in Lemma 3.7, and v' may be constructed from v using the algorithm of Lemma 4.3.

6 Towards a better understanding of imperative languages

The most closely related research on reasoning with continuations and assignments is the work by Mason and Talcott. Over the past few years, they have developed equational theories for a first-order version of Lisp with destructive cell operations [13], for a Λ_C-like language on control [22, 23], and, most recently, for a higher-order imperative version of Lisp without control abstractions [14]. For a fragment of first-order destructive Lisp without arithmetic and recursion, they have also shown that it is possible to obtain a *complete* theory [15].

Mason and Talcott's equational proof systems are essentially *ad hoc* approximations to the operational equivalences of the respective languages. They find the axioms of these theories by extracting and generalizing frequently used laws from example correctness proofs of programs. From a high-level perspective, the axioms are related to our notions of reduction, but the two frameworks strongly differ in the details. Mason and Talcott have not yet addressed the questions of how their theories relate to the underlying theory of procedural abstraction and of how the various theories relate to each other.

An early effort in the direction of equational theories for proving the correctness of higher-order imperative programs is due to Demers and Donahue [4]. The focus of their research is Russell, an extension of the higher-order typed λ-calculus with cells and destructive cell operations; their major result is a proof system for Russell with several dozen axioms, quite unlike our reductions or the Mason-Talcott axioms. Besides equational assertions, the theory also has statements for expressing the purity and legality of expressions as well as their imperative effect. There are no formal results on the equational theory nor its relationship to the original λ-calculus.

Neither Mason and Talcott's research nor the work by Demers and Donahue provides an analysis of the equational theories from the perspective of a reduction theory. Both theories are clearly intended for practical use with a particular programming language and proof system.

The principal motivation for our work is a better understanding of the *essence* of imperative extensions of higher-order programming languages based on the λ-calculus. Our new theories rely on minimal sets of notions of reduction, which provide a simple operational semantics for the respective languages. The λ_v-calculus is the core of all theories; the various theories are conservative extensions of the respective subtheories. In this sense, our operational semantics is *modular*: the semantics of an extended language is an extension of the semantics for the simpler language. The advantage of this approach is that results on evaluation and proof systems automatically lift to richer languages; the disadvantage is a certain weakness of the proof systems. We believe that recent work by Moggi [17] on the *computational* λ-calculus—motivated by similar concerns—and our own work are the correct starting point for developing modular proof systems for large, powerful languages.

The development of a good proof system will require the development of an induction principle and other mathematical tools in order to strengthen the power of the system. One possible solution is to work with the underlying operational approximation relation and to axiomatize its use [14]. The more popular direction relies on the ideas of denotational semantics. Currently, however, denotational semantics provides different models for different languages, especially in the realm of the imperative, higher-order language family. It is consequently difficult to relate results on a language to results on its extensions. Our approach to operational semantics should lead to a collection of denotational models for imperative higher-order languages in which a model for an extended language *contains* the model for the core language as a projection. Such a denotational theory would provide an improved understanding of control and state in programming languages and their relationship to other language facilities.

Acknowledgement Both Udday Reddy and Carolyn Talcott independently suggested to look for simpler, congruent versions of our calculi. Tim Griffin read an early draft of this paper and proposed clarifications of several opaque points in our discussions. Erik Crank came up with large number of counter-examples to the Church-Rosser property of various extensions of our control theory; he also pointed out a flaw in an early draft of the proof for the Soundness Theorem of the new state theory. We also appreciate the referees's efforts, leading to the elimination of a number of mistakes and a greatly improved presentation of our results.

A Appendix: Proof of Lemma 3.11

Before we can sketch the proof of Lemma 3.11, we need to collect some facts about the general shape of proofs of safe equations. We know from the definition of safeness that if $e \overset{\triangleright}{=}_c e'$ is safe then $E[e] \overset{\triangleright}{=}_c E[e']$ is a theorem for every evaluation contexts E. Consequently, by the Church-Rosser and the Standardization Theorems, there must be standard *computation* sequences from $E[e]$ and $E[e']$ to some term p. The proof of Lemma 3.11 relies on the fact that these two standard computation sequences have certain properties.

Definition A.1. (*Standard Computation Sequences*) Standard *reduction* sequences based on the relation $\mathbf{c}$ are defined just like standard reduction sequences for the relation $\mathbf{d}$: see Definition 3.15. We extend standard reduction sequences for $\mathbf{c}$ to standard *computation* sequences for the theory $\boldsymbol{\lambda}_v$-$\mathbf{C}^{\triangleright}$ as follows:

1. All standard reduction sequences are standard computation sequences.
2. If $e \overset{\triangleright}{\longmapsto}_c e_1$ and $e_1, \ldots, e_n$ is a standard computation sequence, then $e, e_1, \ldots, e_n$ is a standard computation sequence.

For the following lemmas, we use the terminology *grabbing a continuation*, by which we mean a sequence of applications of C_{lift} followed by a top-level transition C_T, which creates a *new* abstraction of the form $(\lambda x.\mathcal{A}x)$ and provides access to an abstraction of the evaluation context E. We represent such a continuation with $(\mathcal{A}+E)$. The following lemma provides the justification for this notation by connecting the invocation of the continuation to the reduction of the encoded evaluation context. Again, we label instances of $\mathcal{A}$ in order to keep track of continuations.

Lemma A.2 ([9]) *$((\mathcal{A}+E)v) \overset{\triangleright}{\longmapsto}{}_c^* \mathcal{A}u$ if and only if $E[v] \longrightarrow\!\!\!\!\!\rightarrow_c u$.*

Furthermore, the definition of a standard computation sequence implies that all top-level transitions in a standard computation sequence are part of the series of standard computation steps at the front-end of the sequence. In particular, if a sequence grabs and invokes a continuation, then there is a standard mapping between the two points.

Lemma A.3 *If $E[\mathcal{C}\ e] \overset{\triangleright}{\longmapsto}{}_c^* e(\mathcal{A}^\dagger + E)$ and $e(\mathcal{A}^\dagger + E), \ldots, \mathcal{A}^\dagger v$ is a standard computation sequence (for some value v), then $E[\mathcal{C}\ e] \overset{\triangleright}{\longmapsto}{}_c^* \mathcal{A}^\dagger v$.*

Proof. Obvious: $(\mathcal{A}^\dagger v)$ can only get to the root of the program by computation rules. By Definition A.1 such transitions can only take place within the series of standard computation steps at the front-end of the term sequence. $\square$

The two preceding lemmas lead to the first crucial property of the standard computation sequences for safe equations. If both sequences grab a continuation, the continuation is invoked if and only if both sequences invoke it.

Lemma A.4 *Let $\mathcal{C}e \overset{\triangleright}{=}_c \mathcal{C}e'$ be a safe equation. Let E be an arbitrary evaluation context and let p be such that*

$$E[\mathcal{C}e] \longmapsto_c^* e(\mathcal{A}^\dagger + E), \ldots, p$$

and

$$E[\mathcal{C}e'] \longmapsto_c^* e'(\mathcal{A}^\dagger + E), \ldots, p$$

where $e(\mathcal{A} + E), \ldots, p$ and $e'(\mathcal{A} + E), \ldots, p$ are standard computation sequences.

Then, $e(\mathcal{A}^\dagger + E) \overset{\triangleright}{\longmapsto}{}_c^* \mathcal{A}^\dagger v$ if and only if $e'(\mathcal{A}^\dagger + E) \overset{\triangleright}{\longmapsto}{}_c^* \mathcal{A}^\dagger u$ for some values v and u.

Proof. By Lemma A.3, it suffices to look at the front-end of the standard computation sequence. Thus assume that e invokes the continuation but e' does not. Since the computation sequences are in standard form, the decision to invoke or not to invoke the continuation does not depend on the evaluation context E. Hence, we may consider a less arbitrary context, say, $E \equiv (\lambda x.c)[\ \ ]$ for some constant c not in e or e'. By Lemma A.3, this implies $p \equiv c$.

The second derivation sequence, on the other hand, may or may not discard the newly created continuation. If it does not, p must still contain the corresponding new $\mathcal{A}$-application. On the other hand, if e' throws away its continuation, p can no longer contain any part of the evaluation context, i.e., the unique constant c. In either case, the second derivation sequence places inconsistent requirements on the term p. This contradiction proves our claim. $\square$

A second property of standard computation sequences for safe equations is that if only one of the derivation sequences grabs the evaluation context, then the common term is an $\mathcal{A}$-application and the continuation is never invoked.

Lemma A.5 *Let* $\mathcal{C}e \stackrel{\triangleright}{=}_c e'$ *be a safe equation. Let* E *be an arbitrary evaluation context and let* p *be such that*

$$E[\mathcal{C}e] \longmapsto_c^* e(\mathcal{A}^\dagger + E), \ldots, p$$

where $e(\mathcal{A}^\dagger + E), \ldots, p$ *is a standard computation sequence. Moreover, let*

$$E[e'] \longrightarrow\!\!\!\!\!\rightarrow_c p$$

and let $E[e'], \ldots, p$ *be the corresponding standard computation sequence.*

Then, $p \equiv \mathcal{A}q$ *for some* q*. Moreover, it is impossible that* $e(\mathcal{A}^\dagger + E)$ *invokes the continuation* $(\mathcal{A}^\dagger + E)$*, i.e., it is impossible that* $p \equiv \mathcal{A}^\dagger q$*.*

Proof. It is easy to see that the evaluation context in this continuation must not occur in p because the second term, e', cannot construct $(\mathcal{A}^\dagger + E)$ for arbitrary contexts E. Consequently, the first derivation must eliminate all pieces of E including the labeled continuation $(\mathcal{A}^\dagger + E)$, and p cannot contain any pieces of the evaluation context. As a result, the second derivation sequence must abort the entire evaluation context E without performing a top-level step. Consequently, the term p is of the shape $\mathcal{A}q$ for some term q. By the above argument that p does not contain a tagged abort application, we also know that $p \not\equiv \mathcal{A}^\dagger q$. □

With Lemmas A.4 and A.5, we can now prove Lemma 3.11.

Lemma 3.11 *If* $\boldsymbol{\lambda}_v$-**C**-**safe** $\vdash e = e'$*, then* $\boldsymbol{\lambda}_v$-$\mathbf{C(d)} \vdash e = e'$*.*

Proof. The proof is an analysis of the derivations of the equations $E[e] \stackrel{\triangleright}{=}_c E[e']$. As discussed, there must be two standard computation sequences that start in the two distinct terms and end in a common term:

$$E[e], \ldots, p \text{ and } E[e'], \ldots, p.$$

There are three major cases:

1. Neither standard computation sequence uses top-level rules. Then the standard computation sequences are such that

$$E[e] \longrightarrow\!\!\!\!\!\rightarrow_c p \text{ and } E[e'] \longrightarrow\!\!\!\!\!\rightarrow_c p.$$

   Since $\longrightarrow\!\!\!\!\!\rightarrow_c \subset \longrightarrow\!\!\!\!\!\rightarrow_d$, these reductions also hold in **d**, and $\boldsymbol{\lambda}_v$-$\mathbf{C(d)} \vdash E[e] = E[e']$.

2. Both sequences grab the continuation. According to Lemma A.4, we must now distinguish two subcases:

   (a) Both sequences invoke the continuation:

$$\begin{aligned} E[e] \longmapsto_c^* q(\mathcal{A}^\dagger + E) &\stackrel{\triangleright}{\longmapsto}{}_c^* E'[(\mathcal{A}^\dagger + E)v] \\ &\stackrel{\triangleright}{\longmapsto}{}_c^* \mathcal{A}^\dagger u[y \leftarrow (\mathcal{A}^\dagger + E)] \end{aligned}$$

   and

$$\begin{aligned} E[e'] \longmapsto_c^* q'(\mathcal{A}^\dagger + E) &\stackrel{\triangleright}{\longmapsto}{}_c^* E''[(\mathcal{A}^\dagger + E)v'] \\ &\stackrel{\triangleright}{\longmapsto}{}_c^* \mathcal{A}^\dagger u'[y \leftarrow (\mathcal{A}^\dagger + E)]. \end{aligned}$$

By Lemmas 3.5 through 3.7, we know that the following holds in $\boldsymbol{\lambda}_v$-$\mathbf{C(d)}$:

$$\begin{array}{rll} E[e] & \longmapsto_d^* & \mathcal{C}(\lambda k.q(\mathcal{A}^\dagger + k + E)) \\ & \longrightarrow\!\!\!\!\!\rightarrow_d & \mathcal{C}(\lambda k.ku[y \leftarrow (\mathcal{A}^\dagger + k + E)]) \end{array}$$

and

$$\begin{array}{rll} E[e'] & \longmapsto_d^* & \mathcal{C}(\lambda k.q'(\mathcal{A}^\dagger + k + E)) \\ & \longrightarrow\!\!\!\!\!\rightarrow_d & \mathcal{C}(\lambda k.ku'[y \leftarrow (\mathcal{A}^\dagger + k + E)]). \end{array}$$

Since u and u' are values, the rest of the standard computation sequence must be provable in $\boldsymbol{\lambda}_v$-$\mathbf{C(c)}$:

$$\boldsymbol{\lambda}_v\text{-}\mathbf{C(c)} \vdash u[y \leftarrow (\mathcal{A}^\dagger + E)] = u'[y \leftarrow (\mathcal{A}^\dagger + E)]$$

By this we directly have that

$$\boldsymbol{\lambda}_v\text{-}\mathbf{C(d)} \vdash u[y \leftarrow (\mathcal{A}^\dagger + k + E)] = u'[y \leftarrow (\mathcal{A}^\dagger + k + E)].$$

But then we also get that

$$\begin{array}{rl} \boldsymbol{\lambda}_v\text{-}\mathbf{C(d)} \vdash & \mathcal{C}(\lambda k.ku[y \leftarrow (\mathcal{A}^\dagger + k + E)]) = \\ & \quad \mathcal{C}(\lambda k.ku'[y \leftarrow (\mathcal{A}^\dagger + k + E)]) \end{array}$$

and hence $\boldsymbol{\lambda}_v$-$\mathbf{C(d)} \vdash E[e] = E[e']$.

(b) Neither sequence invokes the continuation. The analysis of case 2a applies again with the exception that the intermediate terms

$$\mathcal{C}(\lambda k.ku[y \leftarrow (\mathcal{A}^\dagger + k + E)])$$

and

$$\mathcal{C}(\lambda k.ku'[y \leftarrow (\mathcal{A}^\dagger + k + E)])$$

look like

$$\mathcal{C}(\lambda k.u[y \leftarrow (\mathcal{A}^\dagger + k + E)])$$

and

$$\mathcal{C}(\lambda k.u'[y \leftarrow (\mathcal{A}^\dagger + k + E)]),$$

respectively.

3. Finally, it may be the case that one sequence grabs the continuation and the other does not:

$$E[e] \longmapsto_c^* q(\mathcal{A}^\dagger + E), \ldots, p \text{ and } E[e'] \longrightarrow\!\!\!\!\!\rightarrow_c p.$$

It follows from Lemma A.5 that p has the shape $\mathcal{A}r$, that r does not contain the tagged continuation, and that q does not invoke the continuation. Again by Lemmas 3.5 through 3.7

$$E[e] \longrightarrow\!\!\!\!\!\rightarrow_d \mathcal{C}(\lambda k.q(\mathcal{A}^\dagger + k + E))$$

and, given that $(\mathcal{A}^\dagger + k + E)$ does not occur in p,

$$E[e] \longrightarrow\!\!\!\!\rightarrow_d \mathcal{C}(\lambda k.q(\mathcal{A}^\dagger + k + E)) \longrightarrow\!\!\!\!\rightarrow_d \mathcal{C}(\lambda k.p), k \notin FV(p).$$

Since we know from Lemma A.5 that p is of the shape $\mathcal{A}r \equiv \mathcal{C}\lambda d.r$ for some r with d not in r, we can derive the rest with a simple calculation:

$$\begin{array}{rcl} \boldsymbol{\lambda}_v\text{-}\mathbf{C(d)} \vdash \mathcal{C}\lambda k.p & = & \mathcal{C}\lambda k.(\mathcal{C}\lambda d.r) \\ & = & \mathcal{C}\lambda k.(\lambda d.r)(\lambda x.\mathcal{A}(k\ x)) \\ & = & \mathcal{C}\lambda k.r \\ & \equiv & p. \end{array}$$

These are all possible cases and now we know that $\boldsymbol{\lambda}_v\text{-}\mathbf{C(d)} \vdash E[e] = E[e']$ for all E. This holds in particular for $E \equiv [\ \ ]$ and therefore $\boldsymbol{\lambda}_v\text{-}\mathbf{C(d)} \vdash e = e'$. □

References

1. ABADI, M., L. CARDELLI, P.-L. CURIEN, AND J.-J. LÉVY. Explicit substitution. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, 1990, 31–46.
2. BARENDREGT, H.P. *The Lambda Calculus: Its Syntax and Semantics.* Revised Edition. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984.
3. CHURCH, A. *Introduction to Mathematical Logic.* Princeton University Press, Princeton, New Jersey, 1956.
4. DEMERS, A. AND J. DONAHUE. Making variables abstract: an equational theory for Russell. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, 1983, 59–72.
5. FELLEISEN, M. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* Ph.D. dissertation, Indiana University, 1987.
6. FELLEISEN, M. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, 180–190.
7. FELLEISEN, M. AND D.P. FRIEDMAN. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, edited by M. Wirsing. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986, 193–217.
8. FELLEISEN, M. AND D.P. FRIEDMAN. A syntactic theory of sequential state. *Theor. Comput. Sci.* **69**(3), 1989, 243–287. Preliminary version in: *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987, 314-325.
9. FELLEISEN, M., D.P. FRIEDMAN, E. KOHLBECKER, AND B. DUBA. A syntactic theory of sequential control. *Theor. Comput. Sci.* **52**(3), 1987, 205–237. Preliminary version in: *Proc. Symposium on Logic in Computer Science*, 1986, 131–141.
10. GRIFFIN, T. A formulae-as-types notion of control. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, 1990, 47–58.

11. LANDIN, P.J. The next 700 programming languages. *Commun. ACM* **9**(3), 1966, 157–166.

12. LANDIN, P.J. The mechanical evaluation of expressions. *Comput. J.* **6**(4), 1964, 308–320.

13. MASON, I.A. Equivalences of first-order Lisp programs. In *Proc. Symposium on Logic in Computer Science*, 1986, 105–117.

14. MASON, I.A. AND C. TALCOTT. Programming, transforming, and proving with function abstractions and memories. In *Proc. International Conference on Automata, Languages and Programming*. Springer Lecture Notes in Computer Science, Berlin, 1989, 574-588.

15. MASON, I.A. AND C. TALCOTT. A sound and complete axiomatization of operational equivalence between programs with memory. In *Proc. Symposium on Logic in Computer Science*, 1989, 284–293.

16. MILNER, R., M. TOFTE, AND R. HARPER. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts and London, England, 1990.

17. MOGGI, E. Computational lambda-calculus and monads. In *Proc. Symposium on Logic in Computer Science*, 1989, 14–23.

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

19. REES, J. AND W. CLINGER (Eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices* **21**(12), 1986, 37–79.

20. STEELE, G.L., JR. *Common Lisp—The Language*. Digital Press, 1984.

21. SUSSMAN, G.J. AND G.L. STEELE JR. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.

22. TALCOTT, C. Rum: An intensional theory function and control abstractions. In *Proc. 1987 Workshop on Foundations of Logic and Functional Programming*. Springer Lecture Notes **306**, 1988.

23. TALCOTT, C. *The Essence of Rum—A Theory of the Intensional and Extensional Aspects of Lisp-type Computation*. Ph.D. dissertation, Stanford University, 1985.