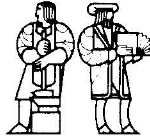
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-459

SHOULD A FUNCTION CONTINUE?

Jon Gary Riecke

September 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Should a Function Continue?

by

Jon Gary Riecke

B.A., Computer Science
Williams College
(1986)

submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment
of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
January 1989

© Massachusetts Institute of Technology 1989

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
January 25, 1989

Certified by ______________________________
Albert R. Meyer
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ______________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Should a Function Continue?

by

Jon Gary Riecke

Submitted to the Department of Electrical Engineering and Computer Science on January 25, 1989 in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science

Abstract

We show that two λ-calculus terms can be *observationally congruent* (*i.e.*, agree in all contexts) but their continuation-passing transforms may not be. We also show that two terms may be congruent in all untyped contexts but fail to be congruent in a language with `call/cc` operators, and that two terms may have the same meaning in a direct semantics but not in a continuation semantics. Hence, familiar reasoning about terms may be unsound in a setting with continuations, demonstrating the need for a theory of continuations.

This document contains corrections to the original thesis submitted in January of 1989. Portions of this report previously appeared in [13].

Thesis Supervisor: Albert R. Meyer

Title: Professor of Computer Science and Engineering

Keywords: Continuations, λ-calculus, operational semantics, denotational semantics.

Acknowledgements

I would first like to thank my advisor, Albert R. Meyer, for suggesting the project and for his numerous hours spent discussing the research. Most noteworthy is his ability to focus attention on the "right" questions. I would also like to thank Bard Bloom, Irene Greif, Lalita Jategaonkar, Trevor Jim, and Mark Reinhold for their comments on drafts of the original paper and this thesis, and for their technical assistance. My officemates, Sally Bemus, Be Hubbard, Robert Schapire, and especially my wife, Michelle Traina Riecke, provided much moral support. I finally (and gratefully) acknowledge the financial support provided by an NSF Graduate Fellowship, and by NSF Grant No. 8511190-DCR and ONR Grant No. N00014-83-K-0125.

Contents

1 Introduction **1**
1.1 Reasoning about Code 2
1.2 Outline of Thesis 3

2 The Language and its Continuation Transform **4**
2.1 Syntax 4
2.2 Operational Semantics 5
2.3 Continuation Transform 6
2.3.1 Definition 6
2.3.2 Fundamental Properties of the Transform 7

3 Continuations May Be Unreasonable **15**

4 Conclusion **19**

A Standard Theorems for the Language **22**
A.1 Church-Rosser Theorem 22
A.2 Applicative Congruence 26

Chapter 1

Introduction

Continuations control program flow using purely functional means. Informally, a **continuation** is a function representing the rest of the program: when passed an intermediate result (a value in a functional language, a store in an imperative language), the function "continues" the computation to the final result. In LISP programs, for example, the control stack can be thought of as representing the continuation of a program: the stack tells the interpreter how to continue the computation to the final answer. At a lower level, a program counter also represents a continuation, although the "function" may not be very clear.

The explicit use of continuations pervades the theory and practice of programming languages. Continuations first appeared in continuation-style semantics for imperative languages [11, 30, 31]. In this style, continuations are explicitly passed to the meanings of all program statements. The meaning of imperative statements can be modeled as functions that change the continuation. For example, in an ALGOL-like language with `goto <label>` statements, each label marks a particular continuation. The meaning of the statement `goto <label>` is one that, upon receiving a store and a continuation, discards that continuation and passes the store to the continuation associated with `<label>`. Highly imperative constructs like `goto` are difficult or impossible to represent in "direct" semantics in which statements are modeled as functions from answers to answers [11, 30].

Continuations appear in at least two other settings. In languages such as LISP and Scheme, the continuation of a program may be accessed through the control operator `call-with-current-continuation` (`call/cc`) [23]. The programmer may then use the continuation to repeat certain calculations, perform error traps, backtrack through a computation, or simulate forks and joins [10]. Continuations have also been used in compilers for languages such as Scheme and ML. These compilers apply a *continuation-passing style* (cps) transform as a fundamental step in compilation [1, 9, 28].

Each of the three settings involves "programming" with continuations, and it is almost self-evident that this requires a different style of thinking. What is not obvious, however, is whether working in a continuation setting requires new reasoning tools. Indeed, certain principles *should* remain valid in the context of continuations. For example, the substitution of actual parameters for formal parameters in procedure calls should not become invalid—otherwise, the addition of continuations would change the programming language in drastic ways!

On the other hand, the mere addition of continuation-based control operators to languages suggests that continuations change programming in a fundamental way. In the presence of control operators, a programmer may be able to distinguish pieces of code that were indistinguishable without control operators, making the language more powerful. One can make similar arguments for the other two settings. For instance, programs not expressible when programming directly in the language become expressible when using cps

converted code.

This thesis attempts to make precise the intuition that continuations "change things" in the three settings of continuations. Using specific counterexamples, we shall prove that certain familiar reasoning principles are *unsound* in the three settings of continuations. In essence, reasoning about code in the usual way may lead one to draw faulty conclusions about the behavior of that code. By understanding the failure of reasoning principles in each of the three settings of continuations, we move closer to understanding continuations themselves; insights generated by the examples will help in building a suitable theory of continuations.

1.1 Reasoning about Code

By "reasoning principles" we mean principles for proving equivalences of code. Such principles capture the notion of "behavior of code." For example, a λ-abstraction applied to an integer argument in LISP behaves the same (ignoring efficiency issues) as the body of the abstraction with the integer in place of the abstracted variable. These two pieces of code are equivalent, and the definition of a LISP interpreter may be used to verify this equivalence.

Two pieces of code are "equivalent" if they produce the same "outcomes" under the interpreter. To make this more precise, we must define the **observations**, the net outcomes of the interpreter considered important. Typically, we choose to observe terms at which the interpreter stops. In the language $\boldsymbol{\lambda}_v$ defined in Chapter 2, we will observe evaluation to numerals.[1] Let $Eval(M)$ be a partial function from terms to terms, representing the output of the interpreter on terms; we then say

Definition 1.1 (Informal) *Two terms M and N are* **observationally equivalent** *if $Eval(M)$ and $Eval(N)$ agree on all observations.*

Two programs are observationally equivalent if they produce the same observable results.

Observational equivalence states that two terms *as given* cannot be told apart by the interpreter. For languages with functional terms, observational equivalence is too coarse; one may still be able to distinguish two observationally equivalent terms. For instance, if we choose to observe "termination of the interpreter" in LISP, any two λ-abstractions would agree on all observations and hence would be considered observationally equivalent. Yet a programmer may be able to distinguish two λ-abstractions by writing a *context* (a term with a hole) that makes the terms evaluate to different observations. One may formalize this ability to distinguish terms:

Definition 1.2 (Informal) *Two terms M and N are* **observationally distinguishable** *iff for some context $C[\cdot]$, $C[M]$ and $C[N]$ differ on some observation (in other words, are not observationally equivalent.)*

The complementary notion is, in fact, more important:

Definition 1.3 (Informal) *Two terms M and N are* **observationally congruent** *(written $M \equiv_{obs} N$) iff they are not observationally distinguishable.*

[1] More complex observations may result in finer distinctions between terms; see [4, 17] for an example of another reasonable notion of observation.

Observational congruence is the congruence closure of observational equivalence.

From a software engineering perspective, observational congruence captures the notion of "modularity" of code. For example, two routines that "sort" should be observationally congruent: the "sort" routines should be interchangeable in any program, and the program should produce the same answers using either routine. Observational congruence also provides one definition of a "correct" compiler optimization: if one piece of code is replaced by a faster yet observationally congruent piece, the optimization is "safe," *i.e.*, the optimized code will still produce the expected answer.

When we say "reasoning about code," we mean reasoning used to prove observational congruences. In fact, almost any reasoning principle may be viewed as a way to verify observational congruences. For instance, fixpoint induction in denotational semantics and pure λ-calculus-like equational reasoning are reasoning tools for proving congruences. These formal reasoning principles help justify the informal observational congruence reasoning used by programmers, clarifying common assumptions about the behavior of code.

1.2 Outline of Thesis

We concentrate on the setting of cps conversion, since the cps transform seems fundamental to understanding the other two settings of continuations. a continuation transform forms the basis of many continuation semantics (*cf.* [24, 26, 30]) and is often used to describe the semantics of `call/cc`-like operators (*cf.* [7, 8].) Chapter 2 describes a call-by-value functional language $\boldsymbol{\lambda}_v$ and its continuation transform, both of which are the focus of study.

In Chapter 3, we describe specific examples that show the failure of reasoning principles based on observational congruence. These examples will have the form "M and N are observationally congruent but not congruent in one of the continuation settings." In particular, we show that two terms may be observationally congruent but their cps-transforms may not be. Similar observations are also made for the other two settings of continuations.

The unsoundness of familiar reasoning principles indicates that a theory of continuations remains to be found. Chapter 4 discusses possible directions for such a theory. One method (currently being pursued) involves extending the retraction-based method of Meyer and Wand [15]. One might also seek results tying the three settings of continuations together. Finally, an Appendix is included which contains proofs of "standard" theorems for $\boldsymbol{\lambda}_v$.

Chapter 2

The Language and its Continuation Transform

This chapter defines $\boldsymbol{\lambda}_v$, a call-by-value version of the language PCF [20, 25], including an interpreter for $\boldsymbol{\lambda}_v$. A call-by-value continuation transform for the language is then given, along with theorems that show the correctness of the transform.

2.1 Syntax

The familiar syntax of the simply-typed λ-calculus forms the basis of $\boldsymbol{\lambda}_v$. Each term in $\boldsymbol{\lambda}_v$ has a type of the form o or $(\sigma \to \tau)$, where o is the sole base type, the type of natural numbers, and $\sigma \to \tau$ is the type of functions from σ to τ.[1] The set of terms with their corresponding types is defined in Figure 2.1. In this definition and throughout the text, Greek letters (with the exception of κ, λ, and μ) denote types, uppercase Roman letters

$x^\sigma : \sigma$	— λ-variables, where $x \in \mathcal{L}$
$f^\sigma : \sigma$	— μ-variables, where $f \in \mathcal{M}$
$c_l : o$	— numerals ($l \geq 0$)
$\mathsf{succ}, \mathsf{pred} : o \to o$	— functional constants
$(\mathsf{cond}\ B\ M\ N) : o$	— conditionals, where $B, M, N : o$
$(M\ N) : \tau$	— applications, where $M : \sigma \to \tau$ and $N : \sigma$
$(\lambda x^\sigma.M) : \sigma \to \tau$	— λ-abstractions, where $M : \tau$
$(\mu f^\sigma.M) : \sigma$	— recursive definitions, where $M : \sigma$

Figure 2.1: The syntax for $\boldsymbol{\lambda}_v$; here, $\mathcal{L}$ and $\mathcal{M}$ are two disjoint, infinite sets of variables. Each variable in $\boldsymbol{\lambda}_v$ is tagged with a type (*cf.* [20]), but types will often be dropped when the context is clear.

denote terms, the lowercase letters f, g, and h are μ-variables, and all other letters (*e.g.*, κ, a, b, c) are λ-variables except when otherwise stated.

The λ- and μ-variables occurring in a term may be **bound** or **free** [2]. If two terms M and N differ only in the names of bound variables, we consider them to be syntactically equivalent and write $M = N$ [2]. A term is **closed** if it contains no free variables; otherwise, a term is **open**.

Contexts are special terms containing holes. A context $C[\cdot]$ is derived from a term M by replacing all free occurrences of some variable in M, say f^σ, by a hole $[\cdot]$. $C[N]$ is the result of replacing every hole in $C[\cdot]$ with N, where $N : \sigma$ and the type of the hole is σ.

[1] As is customary, parentheses will frequently be dropped from types with the understanding that $\to$ associates to the right. For example, $o \to o \to o$ is short for $(o \to (o \to o))$.

$$\begin{array}{rcl|rcl}
(\lambda x.M)\ V & \to_v & M[x := V],\ V \text{ a value} & \mu f.M & \to_v & M[f := \mu f.M] \\
\mathsf{succ}\ c_l & \to_v & c_{l+1} & \mathsf{pred}\ c_0 & \to_v & c_0 \\
\mathsf{cond}\ c_0\ M_0\ M_1 & \to_v & M_0 & \mathsf{pred}\ c_{l+1} & \to_v & c_l \\
\mathsf{cond}\ c_{l+1}\ M_0\ M_1 & \to_v & M_1 & & &
\end{array}$$

$$\frac{B \to B'}{\mathsf{cond}\ B\ M_0\ M_1 \to_v \mathsf{cond}\ B'\ M_0\ M_1} \qquad \frac{M \to_v M',\ c \in \{\mathsf{succ}, \mathsf{pred}\}}{c\ M \to_v c\ M'}$$

$$\frac{M \to_v M'}{M\ N \to_v M'\ N} \qquad \frac{N \to_v N'}{(\lambda x.M)\ N \to_v (\lambda x.M)\ N'}$$

Figure 2.2: Structured rewrite rules for $\boldsymbol{\lambda}_v$. Substitution of the term N for the variable x in M, with the necessary renaming of bound variables, is written $M[x := N]$ (see [2] for a formal definition.)

2.2 Operational Semantics

The relation $\to_v$, the one-step reduction relation on terms of $\boldsymbol{\lambda}_v$, is defined in Figure 2.2 using a structured operational semantics [19, 21]. In reducing applications, operands are substituted in for λ-bound variables only when the operand is a **value.** A **value** (usually denoted by V) is a λ-abstraction, a constant, or a λ-variable. None of these terms can be rewritten using $\to_v$, so a value is a term in evaluated form.[2]

It is relatively easy to see from the fact that values are stopped that $\to_v$ is deterministic. This allows us to define an interpreter for $\boldsymbol{\lambda}_v$ from $\to_v$. Since $\boldsymbol{\lambda}_v$ is a language for arithmetic, we choose the final answers of the interpreter to be numerals. The input to an interpreter for $\boldsymbol{\lambda}_v$ should therefore be closed terms of base type which we call **complete programs.** (A complete program is a program coupled with a particular set of inputs.) The reflexive, transitive closure of the relation $\to_v$, $\twoheadrightarrow_v$, can be used to define a partial recursive function

$$Eval_v\text{: } \textbf{Complete programs} \rightharpoonup \textbf{Numerals}$$

$$Eval_v(M) = \begin{cases} c_l & \text{if } M \twoheadrightarrow_v c_l \\ \textit{undefined} & \text{otherwise} \end{cases}$$

which is an interpreter for the language.

In our investigation of the cps transform we will be most interested in reasoning about the behavior of code under $Eval_v$. We say that

Definition 2.1 M **observationally approximates** N, *written* $M \preceq_v N$, *if, for any context* $C[\cdot]$ *such that* $C[M]$ *and* $C[N]$ *are complete programs,* $C[M] \twoheadrightarrow_v c_l$ *implies* $C[N] \twoheadrightarrow_v c_l$.

Two terms M and N are **observationally congruent**, written $M \equiv^v_{obs} N$, if $M \preceq_v N$ and $N \preceq_v M$.

Observational congruences can be difficult to prove using only the definition [12]. For example, consider the terms $N_1 = \lambda x.(\lambda y.y)\ c_3$ and $N_2 = \lambda x.c_3$. If N_1 is applied to an

[2] Using this rationale, μ-variables might also be considered values, if it were not for the fact that μ-variables may be replaced by terms that require further evaluation. For example, f gets replaced by a non-value in the reduction $\mu f.f \to_v f[f := \mu f.f]$. In contrast, λ-variables remain values when reduced and hence are considered values. This distinction explains the need for two disjoint sets of variables. Plotkin also uses two sets of variables in one version of his metalanguage [22].

argument during the evaluation of a program, the "active" subterm at the next stage will be $(\lambda y.y)\ c_3$ which will reduce to c_3. If N_2 appeared as the subterm instead, c_3 will again be the result. The terms should thus be congruent. This argument, however, is difficult to formalize and is of little use in proving other observational congruences.

Equational reasoning based on $\rightarrow_v$ can be used to prove $N_1 \equiv^v_{obs} N_2$. Define the relation $=_v$ by replacing all $\rightarrow_v$'s in the definition of $\rightarrow_v$ by $=_v$'s, adding the axioms reflexivity, symmetry, and transitivity, and condensing the operational rules with antecedents into the congruence rule

$$\frac{M =_v M'}{C[M] =_v C[M']}$$

where $C[\cdot]$ is *any* context (not necessarily making $C[M]$ a complete program.) The rules of $=_v$ are sound for proving observational congruences.

Theorem 2.2 *If $M =_v N$, then $M \equiv^v_{obs} N$.*

Proof: Delayed to the Appendix. ■

$N_1 \equiv^v_{obs} N_2$ now follows from the fact that $N_1 =_v N_2$.

The converse to Theorem 2.2 is false: there are terms that are observationally congruent but cannot be proven equivalent.[3] The following theorem will be useful in verifying congruences:

Theorem 2.3 *Let M and N be closed terms of the same type. Then $M \preceq_v N$ iff, for all vectors $\vec{V}$ of closed values, $M\ \vec{V} \twoheadrightarrow_v V'_0$ implies $N\ \vec{V} \twoheadrightarrow_v V'_1$ and $V'_0 = V'_1$ if either is a numeral.*

Proof: Delayed to the Appendix. ■

Theorem 2.3 states that *applicative* contexts determine observational congruence (*cf.* [3].)

2.3 Continuation Transform

2.3.1 Definition

The continuation transform for $\boldsymbol{\lambda}_v$ is based on a cps transform appropriate for call-by-value [9, 15, 19]. The transform of a term M, written $\overline{M}$, is another term of $\boldsymbol{\lambda}_v$. Figure 2.3 defines the transform of a term by structural induction on the term.

The behavior of the interpreter for $\boldsymbol{\lambda}_v$ provides clues to understanding the continuized version of a term. Basically, the flow of control is made explicit by the continuations of a cps-converted term. For example, since values are not evaluated, the cps transform of a value simply passes the value to a continuation (the rest of the program.) For applications as well, the continuations in the transform of an application mimic the flow of control in the interpreter: the continuation passed to the operator first evaluates the operand and passes control to the operand's continuation, which, in turn, applies the operator to the operand.

The explicit incorporation of continuations requires that the transform change the type of a term. A continuized term accepts a continuation as an argument (a function from some type to a final answer), and produces a final answer given that continuation. The type of final answers for $\boldsymbol{\lambda}_v$ is o, so a term of type o is transformed into a term of type $(o \rightarrow o) \rightarrow o$.

[3] In fact, observational congruence is *not* axiomatizable [2, 32], so one cannot hope for an equational proof system that captures observational congruence.

$$
\begin{array}{rcl}
\overline{x^\sigma} & = & \lambda\kappa.\kappa\ x^{\sigma'} \\
\overline{f^\sigma} & = & \lambda\kappa.f^{(\sigma'\to o)\to o}\ \kappa \\
\overline{c_l} & = & \lambda\kappa.\kappa\ c_l \\
\overline{\mathsf{succ}} & = & \lambda\kappa.\kappa\ (\lambda x^o.\lambda\kappa_1.\kappa_1\ (\mathsf{succ}\ x)) \\
\overline{\mathsf{pred}} & = & \lambda\kappa.\kappa\ (\lambda x^o.\lambda\kappa_1.\kappa_1\ (\mathsf{pred}\ x)) \\
\overline{\mathsf{cond}\ B\ M_0\ M_1} & = & \lambda\kappa.\overline{B}(\lambda m^o.\mathsf{cond}\ m\ (\overline{M_0}\ \kappa)\ (\overline{M_1}\ \kappa)) \\
\overline{(M\ N)} & = & \lambda\kappa.\overline{M}\ (\lambda m^{(\sigma\to\tau)'}.\overline{N}\ (\lambda n^{\sigma'}.m\ n\ \kappa)) \\
 & & (\text{where } M:\sigma\to\tau \text{ and } N:\sigma) \\
\overline{\lambda x^\sigma.M} & = & \lambda\kappa.\kappa\ (\lambda x^{\sigma'}.\overline{M}) \\
\overline{\mu f^\sigma.M} & = & \lambda\kappa.(\mu f^{(\sigma'\to o)\to o}.\overline{M})\ \kappa
\end{array}
$$

Figure 2.3: The continuation transform for $\boldsymbol{\lambda}_v$. The types of continuations κ (which have the form $\alpha' \to o$) have been omitted for clarity. Note that variables change types when transformed.

The situation for higher-typed terms is more complicated. The continuation of a higher-order term needs to accept functions which, given a value and another continuation, produce final answers. The transform of a term of type α is thus a term of type $(\alpha' \to o) \to o$, where α' is defined recursively by (*cf.* [15])

$$
\begin{array}{rcl}
o' & = & o \\
(\sigma \to \tau)' & = & \sigma' \to (\tau' \to o) \to o.
\end{array}
$$

2.3.2 Fundamental Properties of the Transform

By inspecting the definition of the transform, one may observe that every operand in a transformed term is a value and hence need not be evaluated. In other words, transformed terms may be evaluated **tail-recursively**. Tail-recursiveness can lead to increased efficiency. A traditional call-by-value interpreter (or code generated by compilers) uses a stack to remember the position of the subterm currently being evaluated. In transformed terms, all operands in applications are in evaluated form, so an interpreter designed specifically for transformed terms does not require a stack.[4]

A corollary to the fact that all operands are values is **unambiguous reducibility**: call-by-name and call-by-value reduction strategies coincide on transformed terms. Unambiguous reducibility allows one to use the transform to simulate call-by-value in a call-by-name interpreter, as is done in [19].

Of course, the transform must satisfy correctness properties as well. If one expects to use the transform as a first step in compilation, for example, transformed terms must not produce different answers than the original terms! The continuation transform for the language satisfies two properties that guarantee its correctness: provable equality (*i.e.*, $=_v$) is preserved by the transform and complete programs produce the same output as their transformed versions [9, 19].

2.3.2.1 Preservation of equational reasoning

We follow Plotkin's proof in [19] to show that $M =_v N$ implies $\overline{M} =_v \overline{N}$.

[4]One may regard the cps version of a term as incorporating an explicit representation of the interpreter's control stack.

Substitutions performed by $=_v$ pose problems to a direct proof. Suppose, for example, that $=_v$ performs the substitution $M[x := V]$. We want $\overline{(\lambda x.M)\,V} =_v \overline{M[x := V]}$. In point of fact, it is easy to show that $\overline{(\lambda x.M)\,V} =_v \overline{M}[x := \Psi(V)]$, where

Definition 2.4 *If V is a value, then $\Psi(V)$ is defined*

- $\Psi(x^\sigma) = x^{\sigma'}$;
- $\Psi(c_l) = c_l$;
- $\Psi(\mathsf{succ}) = \lambda x.\lambda\kappa_1.\kappa_1\ (\mathsf{succ}\, x)$;
- $\Psi(\mathsf{pred}) = \lambda x.\lambda\kappa_1.\kappa_1\ (\mathsf{pred}\, x)$;
- $\Psi(\lambda x^\sigma.M) = \lambda x^{\sigma'}.\overline{M}$.

(Essentially, $\Psi(V)$ is $\overline{V}$ without the leading continuation.) The following lemma allows us to complete the argument that $\overline{(\lambda x.M)\,V} =_v \overline{M[x := V]}$:

Lemma 2.5 *If V is a value and x is a λ-variable, then $\overline{M}[x^{\sigma'} := \Psi(V)] = \overline{M[x^\sigma := V]}$.*

Proof: By structural induction on M. For the base case, M must be a constant or variable:

Case 1: $M = x$. Then $\overline{M}[x := \Psi(V)] = \lambda\kappa.\kappa\ \Psi(V) = \overline{V} = \overline{M[x := V]}$.

Case 2: $M = t$ for some variable $t \neq x$. Then $\overline{M}[x := \Psi(V)] = \overline{t} = \overline{M[x := V]}$.

Case 3: $M = a$ for some constant a. Similar to Case 2.

For the induction case, we also divide into cases depending on the form of M:

Case 1: $M = \mathsf{cond}\ B\ M_0\ M_1$. Then

$$\begin{aligned}\overline{M}[x := \Psi(V)] &= (\lambda\kappa.\overline{B}\ (\lambda m.\mathsf{cond}\ m\ (\overline{M_0}\ \kappa)\ (\overline{M_1}\ \kappa)))[x := \Psi(V)]\\ &= \lambda\kappa.\overline{B[x := V]}\ (\lambda m.\mathsf{cond}\ m\ (\overline{M_0[x := V]}\ \kappa)\ (\overline{M_1[x := V]}\ \kappa))\\ &\quad \text{(by the induction hypothesis)}\\ &= \overline{M[x := V]}.\end{aligned}$$

Case 2: $M = (M_1\ M_2)$. Then

$$\begin{aligned}\overline{M}[x := \Psi(V)] &= (\lambda\kappa.\overline{M_1}\ (\lambda m.\overline{M_2}\ (\lambda n.m\ n\ \kappa)))[x := \Psi(V)]\\ &= \lambda\kappa.\overline{M_1[x := V]}\ (\lambda m.\overline{M_2[x := V]}\ (\lambda n.m\ n\ \kappa))\\ &\quad \text{(by the induction hypothesis)}\\ &= \overline{M[x := V]}.\end{aligned}$$

Case 3: $M = \lambda y.M'$. If $y = x$, then $\overline{M}[x := \Psi(V)] = \overline{M} = \overline{M[x := V]}$. If $y \neq x$,

$$\begin{aligned}\overline{M}[x := \Psi(V)] &= (\lambda\kappa.\kappa\ (\lambda y.\overline{M'}))[x := \Psi(V)]\\ &= \lambda\kappa.\kappa\ (\lambda y.\overline{M'[x := V]})\\ &\quad \text{(by the induction hypothesis)}\\ &= \overline{M[x := V]}.\end{aligned}$$

Case 4: $M = \mu f.M'$. We know that $x \neq f$; so

$$\begin{aligned}\overline{M}[x := \Psi(V)] &= (\lambda\kappa.(\mu f.\overline{M'})\ \kappa)[x := \Psi(V)] \\ &= \lambda\kappa.(\mu f.\overline{M'[x := V]})\ \kappa \\ &\quad \text{(by the induction hypothesis)} \\ &= \overline{M[x := V]}.\end{aligned}$$

We have exhausted all cases, hence the lemma holds. ■

The analog of Lemma 2.5 for recursive definitions works somewhat more easily:

Lemma 2.6 *If f is a μ-variable, then* $\overline{M}[f^{(\sigma' \to o) \to o} := \mu f^{(\sigma' \to o) \to o}.\overline{N}] = \overline{M[f^{\sigma} := \mu f^{\sigma}.N]}$.

Proof: By structural induction on M. In the base case, we divide into cases on the form of M:

Case 1: $M = f$. Then $\overline{M}[f := \mu f.\overline{N}] = \lambda\kappa.(\mu f.\overline{N})\ \kappa = \overline{\mu f.N} = \overline{M[f := \mu f.N]}$.

Case 2: $M = t$ for some variable $t \neq f$. Then $\overline{M}[f := \mu f.\overline{N}] = \overline{t} = \overline{M[f := \mu f.N]}$.

Case 3: $M = a$ for some constant a. Similar to Case 2.

For the induction case, there are four cases to consider:

Case 1: $M = \mathsf{cond}\ B\ M_0\ M_1$. Then

$$\begin{aligned}\overline{M}[f := \mu f.\overline{N}] &= (\lambda\kappa.\overline{B}\ (\lambda m.\mathsf{cond}\ m\ (\overline{M_0}\ \kappa)\ (\overline{M_1}\ \kappa)))[f := \mu f.\overline{N}] \\ &= \lambda\kappa.\overline{B[f := \mu f.N]}\ (\lambda m.\mathsf{cond}\ m\ (\overline{M_0[f := \mu f.N]}\ \kappa)\ (\overline{M_1[f := \mu f.N]}\ \kappa)) \\ &\quad \text{(by the induction hypothesis)} \\ &= \overline{M[f := \mu f.N]}.\end{aligned}$$

Case 2: $M = (M_1\ M_2)$. Thus,

$$\begin{aligned}\overline{M}[f := \mu f.\overline{N}] &= (\lambda\kappa.\overline{M_1}\ (\lambda m.\overline{M_2}\ (\lambda n.m\ n\ \kappa)))[f := \mu f.\overline{N}] \\ &= \lambda\kappa.\overline{M_1[f := \mu f.N]}\ (\lambda m.\overline{M_2[f := \mu f.N]}\ (\lambda n.m\ n\ \kappa)) \\ &\quad \text{(by the induction hypothesis)} \\ &= \overline{M[f := \mu f.N]}.\end{aligned}$$

Case 3: $M = \lambda y.M'$. Note that $f \neq y$; thus

$$\begin{aligned}\overline{M}[f := \mu f.\overline{N}] &= (\lambda\kappa.\kappa\ (\lambda y.\overline{M'}))[f := \mu f.\overline{N}] \\ &= \lambda\kappa.\kappa\ (\lambda y.\overline{M'[f := \mu f.N]}) \\ &\quad \text{(by the induction hypothesis)} \\ &= \overline{M[f := \mu f.N]}.\end{aligned}$$

Case 4: $M = \mu g.M'$. If $g = f$, $\overline{M}[f := \mu f.\overline{N}] = \overline{M} = \overline{M[f := \mu f.N]}$. On the other hand, if $g \neq f$,

$$\begin{aligned}\overline{M}[f := \mu f.\overline{N}] &= (\lambda\kappa.(\mu g.\overline{M'})\ \kappa)[f := \mu f.\overline{N}] \\ &= \lambda\kappa.(\mu g.\overline{M'[f := \mu f.N]})\ \kappa \\ &\quad \text{(by the induction hypothesis)} \\ &= \overline{M[f := \mu f.N]}.\end{aligned}$$

This concludes the proof. ■

Given these two lemmas, we may complete the proof of the theorem:

Theorem 2.7 *If $M =_v N$, then $\overline{M} =_v \overline{N}$.*

Proof: By induction on the length n of the proof that $M =_v N$. In the base case, the length of the proof is 1, so an axiom was used:

Case 1: $(\lambda x.M)\ V =_v M[x := V]$, where V is a value. Recall that $\overline{V} = \lambda\kappa.\kappa\,\Psi(V)$. Therefore,

$$\begin{aligned}
\overline{(\lambda x.M)\ V} &=_v \lambda\kappa.(\lambda\kappa_1.\kappa_1\ (\lambda x.\overline{M}))\ (\lambda m.\overline{V}\ (\lambda n.m\ n\ \kappa)) \\
&=_v \lambda\kappa.(\lambda m.\overline{V}\ (\lambda n.m\ n\ \kappa))\ (\lambda x.\overline{M}) \\
&=_v \lambda\kappa.\overline{V}\ (\lambda n.(\lambda x.\overline{M})\ n\ \kappa) \\
&=_v \lambda\kappa.(\lambda n.(\lambda x.\overline{M})\ n\ \kappa)\ \Psi(V) \\
&=_v \lambda\kappa.(\lambda x.\overline{M})\ \Psi(V)\ \kappa \\
&=_v \lambda\kappa.(\overline{M}[x := \Psi(V)])\ \kappa \\
&=_v \lambda\kappa.\overline{M[x := V]}\ \kappa
\end{aligned}$$

where the last equation follows from Lemma 2.5. Examining the continuation transform, we note that every continuized term begins with a λ-abstraction; thus,

$$\lambda\kappa.\overline{M[x := V]}\ \kappa \ =_v\ \overline{M[x := V]}$$

so $\overline{(\lambda x.M)\ V} =_v \overline{M[x := V]}$.

Case 2: $\textsf{cond}\ c_0\ M_0\ M_1 =_v M_0$. By calculation,

$$\begin{aligned}
\overline{\textsf{cond}\ c_0\ M_0\ M_1} &=_v \lambda\kappa.(\lambda\kappa_1.\kappa_1\ c_0)\ (\lambda m.\textsf{cond}\ m\ (\overline{M_0}\ \kappa)\ (\overline{M_1}\ \kappa)) \\
&=_v \lambda\kappa.\textsf{cond}\ c_0\ (\overline{M_0}\ \kappa)\ (\overline{M_1}\ \kappa) \\
&=_v \lambda\kappa.(\overline{M_0}\ \kappa)\ =_v\ \overline{M_0}.
\end{aligned}$$

Case 3: $\textsf{cond}\ c_{l+1}\ M_0\ M_1 =_v M_1$. Similar to the previous case.

Case 4: $\textsf{succ}\ c_l =_v c_{l+1}$. By calculation,

$$\begin{aligned}
\overline{\textsf{succ}\ c_l} &=_v \lambda\kappa.(\lambda\kappa_1.\kappa_1\ (\lambda x.\lambda\kappa_2.\kappa_2\ (\textsf{succ}\ x)))\ (\lambda m.(\lambda\kappa_3.\kappa_3\ c_l)\ (\lambda n.m\ n\ \kappa)) \\
&=_v \lambda\kappa.(\lambda\kappa_3.\kappa_3\ c_l)\ (\lambda n.(\lambda x.\lambda\kappa_2.\kappa_2\ (\textsf{succ}\ x))\ n\ \kappa) \\
&=_v \lambda\kappa.(\lambda x.\lambda\kappa_2.\kappa_2\ (\textsf{succ}\ x))\ c_l\ \kappa \\
&=_v \lambda\kappa.\kappa\ (\textsf{succ}\ c_l)\ =_v\ \lambda\kappa.\kappa\ c_{l+1}\ =_v\ \overline{c_{l+1}}.
\end{aligned}$$

Case 5: $\textsf{pred}\ c_0 =_v c_0$. Similar to the previous case.

Case 6: $\textsf{pred}\ c_{l+1} =_v c_l$. Similar to the previous case.

Case 7: $\mu f.M =_v M[f := \mu f.M]$. By calculation,

$$\begin{aligned}
\overline{\mu f.M} &=_v \lambda\kappa.(\mu f.\overline{M})\ \kappa \\
&=_v \lambda\kappa.(\overline{M}[f := \mu f.\overline{M}])\ \kappa \\
&=_v \lambda\kappa.(\overline{M[f := \mu f.M]})\ \kappa\ =_v\ \overline{M[f := \mu f.M]},
\end{aligned}$$

the third equation following from Lemma 2.6.

Case 8: $M =_v M$. Trivial.

In the induction case, the length of the proof is $n + 1$; again, we divide into cases, this time depending on the last rule used:

Case 1: $M =_v N$ and $N =_v P$ implies $M =_v P$. By the induction hypothesis, we know that $\overline{M} =_v \overline{N}$ and $\overline{N} =_v \overline{P}$, so we can conclude that $\overline{M} =_v \overline{P}$ by the transitivity rule.

Case 2: $M =_v N$ implies $N =_v M$. Trivial.

Case 3: $M =_v N$ implies $C[M] =_v C[N]$. Using the induction hypothesis $\overline{M} =_v \overline{N}$, an easy structural induction on the context $C[\cdot]$ shows that $\overline{C[M]} =_v \overline{C[N]}$.

This list exhausts the possibilities for last rule used, hence we are done. ■

2.3.2.2 Adequacy

Theorem 2.7 does not explain the correspondence of *evaluation* of terms and their cps-versions. For complete programs in particular, we expect the interpreter to give the same answers from both the direct and continuized versions, except that continuized versions must be passed a "default continuation," *viz.*, the identity function:

$$M \twoheadrightarrow_v c_l \text{ iff } \overline{M}\,(\lambda x.x) \twoheadrightarrow_v c_l.$$

Indeed, this fact must hold if we wish to use cps conversion in compilers.[5]

The proof proceeds using the method in [19]. The key observation is that certain reductions on transformed terms have no corresponding reduction on non-continuized versions. For example, consider the complete program c_5. The direct version cannot be reduced, but $\overline{c_5}\,(\lambda x.x)$ can be:

$$(\lambda\kappa.\kappa\; c_5)\,(\lambda x.x) \rightarrow_v (\lambda x.x)\; c_5 \rightarrow_v c_5.$$

The first reduction is called an **administrative** reduction, since only a continuation is passed. The relation $\star$ applies a continuized term to a continuation and performs all possible administrative reductions:

Definition 2.8 *For any term $M : \sigma$ and any value $K : \sigma' \to o$, we define $\overline{M} \star K$ by*

$$\begin{array}{rcl}
\overline{M} \star K &=& K\,\Psi(M), \quad \textit{if } M \textit{ is a value} \\
\overline{f^\sigma} \star K &=& f^{(\sigma' \to o) \to o}\, K, \quad \textit{if } f \textit{ is a } \mu\textit{-variable} \\
\overline{(\mathsf{cond}\; B\; M_1\; M_2)} \star K &=& \left\{ \begin{array}{ll} \mathsf{cond}\,\Psi(B)\,(\overline{M_1}\; K)\,(\overline{M_2}\; K) & \textit{if } B \textit{ is a value} \\ \overline{B} \star (\lambda m.(\mathsf{cond}\; m\; (\overline{M_1}\; K)\,(\overline{M_2}\; K))) & \textit{otherwise} \end{array} \right. \\
\overline{(M_1\; M_2)} \star K &=& \left\{ \begin{array}{ll} \overline{M_1} \star (\lambda m.\overline{M_2}\,(\lambda n.m\; n\; K)) & \textit{if } M_1 \textit{ is not a value} \\ \overline{M_2} \star (\lambda n.\Psi(M_1)\; n\; K) & \textit{if } M_1\textit{, but not } M_2\textit{, is a value} \\ \Psi(M_1)\,\Psi(M_2)\; K & \textit{otherwise} \end{array} \right. \\
\overline{\mu f.M'} \star K &=& \mu f.(\overline{M'}\; K)
\end{array}$$

The following lemma confirms that the definition actually represents a "partial reduction" of a continuized term:

[5]Note that the ⇒ direction follows from Theorems 2.2 and 2.7, but the converse does not follow directly.

Lemma 2.9 *If K is a value, then $\overline{M}\ K \twoheadrightarrow_v \overline{M} \star K$.*

Proof: By structural induction on M. For the base case, divide into cases depending on M:

> *Case* 1: $M = x$. Then $\overline{M}\ K = (\lambda\kappa.\kappa\ x)\ K \rightarrow_v K\ x = \overline{M} \star K$.
>
> *Case* 2: $M = f$. Then $\overline{M}\ K = (\lambda\kappa.f\ \kappa)\ K \rightarrow_v f\ K = \overline{M} \star K$.
>
> *Case* 3: $M = c_l$. Then $\overline{M}\ K = (\lambda\kappa.\kappa\ c_l)\ K \rightarrow_v K\ c_l = \overline{M} \star K$.
>
> *Case* 4: $M = \mathsf{succ}$. Then $\overline{M}\ K \rightarrow_v K\ (\lambda x.\lambda\kappa.\kappa\ \mathsf{succ}\ x) = \overline{M} \star K$.
>
> *Case* 5: $M = \mathsf{pred}$. Similar to the previous case.

For the induction case,

Case 1: $M = \mathsf{cond}\ B\ M_1\ M_2$. Then

$$\begin{array}{rcl} \overline{M}\ K & = & (\lambda\kappa.\overline{B}\ (\lambda m.\mathsf{cond}\ m\ (\overline{M_1}\ \kappa)\ (\overline{M_2}\ \kappa)))\ K \\ & \rightarrow_v & \overline{B}\ (\lambda m.\mathsf{cond}\ m\ (\overline{M_1}\ K)\ (\overline{M_2}\ K)). \end{array}$$

If B is a value, then $\overline{M}\ K \twoheadrightarrow_v \mathsf{cond}\ \Psi(B)\ (\overline{M_1}\ K)\ (\overline{M_2}\ K)$; otherwise,

$$\begin{array}{rcl} \overline{M}\ K & \twoheadrightarrow_v & \overline{B} \star (\lambda m.\mathsf{cond}\ m\ (\overline{M_1}\ K)\ (\overline{M_2}\ K)) = \overline{M} \star K \\ & & \text{(by the induction hypothesis.)} \end{array}$$

Case 2: $M = (M_1\ M_2)$. If M_1 is not a value,

$$\begin{array}{rcl} \overline{M}\ K & = & (\lambda\kappa.\overline{M_1}\ (\lambda m.\overline{M_2}\ (\lambda n.m\ n\ \kappa)))\ K \\ & \rightarrow_v & \overline{M_1}\ (\lambda m.\overline{M_2}\ (\lambda n.m\ n\ K)) \\ & \twoheadrightarrow_v & \overline{M_1} \star (\lambda m.\overline{M_2}\ (\lambda n.m\ n\ K)) = \overline{M} \star K \\ & & \text{(by the induction hypothesis.)} \end{array}$$

If M_1 but not M_2 is a value,

$$\begin{array}{rcl} \overline{M}\ K & \twoheadrightarrow_v & \overline{M_2}\ (\lambda n.\Psi(M_1)\ n\ K) \\ & \twoheadrightarrow_v & \overline{M_2} \star\ (\lambda n.\Psi(M_1)\ n\ K) = \overline{M} \star K \\ & & \text{(by the induction hypothesis.)} \end{array}$$

Finally, if both M_1 and M_2 are values,

$$\overline{M}\ K \twoheadrightarrow_v \overline{M_2}\ (\lambda n.\Psi(M_1)\ n\ K) \twoheadrightarrow_v \Psi(M_1)\ \Psi(M_2)\ K = \overline{M} \star K.$$

Case 3: $M = \lambda x.M'$. Then

$$\overline{M}\ K = (\lambda\kappa.\kappa\ (\lambda x.\overline{M'}))\ K \rightarrow_v K\ (\lambda x.\overline{M'}) = \overline{M} \star K.$$

Case 4: $M = \mu f.M'$. Then

$$\overline{M}\ K = (\lambda\kappa.(\mu f.\overline{M'})\ \kappa)\ K \rightarrow_v (\mu f.\overline{M'})\ K = \overline{M} \star K.$$

This concludes the proof of the lemma. ∎

Once the administrative reductions on a continuized term have been performed, the next reductions correspond to reductions on the original version of the term:

Lemma 2.10 *If $M \to_v N$ and K is a value, then $\overline{M} \star K \twoheadrightarrow_v \overline{N} \star K$.*

Proof: By induction on the length of proof of $M \to_v N$. In the base case, the length of the reduction is 1; we divide into cases depending on the operational rule used:

Case 1: $(\lambda x.M')\ V \to_v M'[x := V]$. Then

$$\begin{array}{rcl} \overline{M} \star K & = & \Psi(\lambda x.M')\ \Psi(V)\ K \\ & \to_v & (\overline{M'}[x := \Psi(V)])\ K \\ & \twoheadrightarrow_v & \overline{M'[x := V]} \star K \\ & & \text{(by Lemmas 2.5 and 2.9)} \\ & = & \overline{N} \star K. \end{array}$$

Case 2: $\mathsf{succ}\, c_l \to_v c_{l+1}$. Then

$$\overline{M} \star K \twoheadrightarrow_v \Psi(\mathsf{succ})\ \Psi(c_l)\ K \twoheadrightarrow_v K\ (\mathsf{succ}\, c_l) \to_v K\ c_{l+1} = \overline{N} \star K.$$

Case 3: $\mathsf{pred}\, c_0 \to_v c_0$. Similar to the previous case.

Case 4: $\mathsf{pred}\, c_{l+1} \to_v c_l$. Similar to the previous case.

Case 5: $\mathsf{cond}\ c_0\ M_1\ M_2 \to_v M_1$. Then

$$\overline{M} \star K = \mathsf{cond}\ c_0\ (\overline{M_1}\ K)\ (\overline{M_2}\ K) \twoheadrightarrow_v \overline{M_1} \star K$$

by Lemma 2.9.

Case 6: $\mathsf{cond}\ c_{l+1}\ M_1\ M_2 \to_v M_2$. Similar to the previous case.

Case 7: $(\mu f.M') \to_v M'[f := \mu f.M']$. Then

$$\begin{array}{rcl} \overline{M} \star K & = & (\mu f.\overline{M'})\ K \\ & \to_v & (\overline{M'}[f := \mu f.\overline{M'}])\ K \\ & \twoheadrightarrow_v & \overline{M'[f := \mu f.M']} \star K \end{array}$$

by Lemmas 2.6 and 2.9.

In the induction case we consider proofs of length greater than 1, and divide into cases depending on the last operational rule used:

Case 1: $B \to_v B'$ implies $\mathsf{cond}\ B\ M_1\ M_2 \to_v \mathsf{cond}\ B'\ M_1\ M_2$. Note that B cannot be a value; hence if B' is not a value,

$$\begin{array}{rcl} \overline{M} \star K & = & \overline{B} \star (\lambda m.\mathsf{cond}\ m\ (\overline{M_1}\ K)\ (\overline{M_2}\ K)) \\ & \twoheadrightarrow_v & \overline{B'} \star (\lambda m.\mathsf{cond}\ m\ (\overline{M_1}\ K)\ (\overline{M_2}\ K)) = \overline{N} \star K \end{array}$$

by the induction hypothesis. If B' is a value, then

$$\overline{M} \star K \twoheadrightarrow_v \mathsf{cond}\ \Psi(B')\ (\overline{M_1}\ K)\ (\overline{M_2}\ K) = \overline{N} \star K.$$

Case 2: $P \to_v P'$ implies $\mathsf{succ}\, P \to_v \mathsf{succ}\, P'$. P cannot be a value, so if P' is not a value,

$$\begin{array}{rcl} \overline{M} \star K & = & \overline{P} \star (\lambda n.(\lambda x.\lambda \kappa.\kappa\ (\mathsf{succ}\, x))\ n\ K) \\ & \twoheadrightarrow_v & \overline{P'} \star (\lambda n.(\lambda x.\lambda \kappa.\kappa\ (\mathsf{succ}\, x))\ n\ K) = \overline{N} \star K \end{array}$$

by the induction hypothesis. If P' is a value, then

$$\overline{M} \star K \twoheadrightarrow_v (\lambda x.\lambda \kappa.\kappa\ (\mathsf{succ}\, x))\ \Psi(P')\ K = \overline{N} \star K.$$

Case 3: $P \rightarrow_v P'$ implies $\mathsf{pred}\, P \rightarrow_v \mathsf{pred}\, P'$. Similar to the previous case.

Case 4: $Q \rightarrow_v Q'$ implies $(\lambda x.P)\, Q \rightarrow_v (\lambda x.P)\, Q'$. Similar to the previous case.

Case 5: $P \rightarrow_v P'$ implies $P\, Q \rightarrow_v P'\, Q$. P cannot be a value, so if P' is not a value,

$$\begin{array}{rcl} \overline{M} \star K & = & \overline{P} \star (\lambda m.\overline{Q}\ (\lambda n.m\ n\ K)) \\ & \twoheadrightarrow_v & \overline{P'} \star (\lambda m.\overline{Q}\ (\lambda n.m\ n\ K)) = \overline{N} \star K \end{array}$$

by the induction hypothesis. If P' is a value and Q is not, then

$$\begin{array}{rcl} \overline{M} \star K & \twoheadrightarrow_v & \overline{Q}\ (\lambda n.\Psi(P')\ n\ K) \\ & \twoheadrightarrow_v & \overline{Q} \star (\lambda n.\Psi(P')\ n\ K) = \overline{N} \star K. \end{array}$$

by Lemma 2.9. If both P' and Q are values, then

$$\begin{array}{rcl} \overline{M} \star K & \twoheadrightarrow_v & \overline{Q}\ (\lambda n.\Psi(P')\ n\ K) \\ & \twoheadrightarrow_v & \Psi(P')\ \Psi(Q)\ K = \overline{N} \star K. \end{array}$$

As all operational rules have been considered, we are done. ■

These facts about administrative and non-administrative reductions on continuized terms give us the ability to prove the following theorem originally due to Fischer [9]:

Theorem 2.11 (Adequacy) *If M is a complete program, then*

$$Eval_v(M) = c_l \quad \textit{iff} \quad Eval_v(\overline{M}\ (\lambda x^o.x)) = c_l.$$

Proof: ($\Rightarrow$) Suppose $Eval_v(M) = c_l$; then we know that $M \twoheadrightarrow_v c_l$. By Lemmas 2.9 and 2.10 we then have

$$\overline{M}\ (\lambda x.x) \twoheadrightarrow_v \overline{M} \star (\lambda x.x) \twoheadrightarrow_v \overline{c_l} \star (\lambda x.x) \twoheadrightarrow_v c_l.$$

Thus, $Eval_v(\overline{M}\ (\lambda x.x)) = c_l$.

($\Leftarrow$) Suppose $Eval_v(M)$ is not defined. Then

$$M \rightarrow_v M_1 \rightarrow_v M_2 \rightarrow_v \ldots$$

By Lemmas 2.9 and 2.10, we thus know

$$\overline{M}\ (\lambda x.x) \twoheadrightarrow_v \overline{M} \star (\lambda x.x) \twoheadrightarrow_v \overline{M_1} \star (\lambda x.x) \twoheadrightarrow_v \overline{M_2} \star (\lambda x.x) \twoheadrightarrow_v \ldots$$

so $Eval_v(\overline{M}\ (\lambda x^o.x))$ is not defined either. ■

Chapter 3

Continuations May Be Unreasonable

The Adequacy Theorem establishes a strong connection between the evaluation of terms and their continuized versions. The theorem easily extends to reasoning about complete programs, *viz.*, proving observational congruences. It follows that for complete programs M and N,

$$M \equiv^v_{obs} N \text{ iff } \overline{M}\,(\lambda x.x) \equiv^v_{obs} \overline{N}\,(\lambda x.x).$$

The connection between direct and continuized versions of higher-order terms is less obvious, but one may still see a partial relationship between reasoning on direct versus reasoning on continuized terms:

Corollary 3.1 *If $\overline{M} \equiv^v_{obs} \overline{N}$, then $M \equiv^v_{obs} N$.*

Proof: Suppose M and N were distinguishable by some context $C[\cdot]$. Then by the Adequacy Theorem, the context $\overline{C[\cdot]}\,(\lambda x.x)$ would distinguish $\overline{M}$ and $\overline{N}$, a contradiction. ■

In particular, if one can distinguish two terms by a context, the transforms of those terms will also be distinguishable.

The problem with the continuation transform is that the converse of Corollary 3.1 does *not* hold: observational congruence on direct terms does not coincide with congruence on continuized terms. Similar anomalies occur in the other two settings. For example, suppose we augment $\boldsymbol{\lambda}_v$ with the `call/cc`-like operators $\mathcal{C}$ and $\mathcal{A}$ defined in [7, 8]. Terms that are observationally congruent in $\boldsymbol{\lambda}_v$ may become distinguishable using contexts containing these new operators. In the case of continuation semantics, there are observationally congruent terms that are equivalent in a direct semantics but not equivalent in a continuation semantics. Reasoning principles based on observational congruence may thus become *unsound* in settings involving continuations.

In the continuation transform setting, the anomaly is manifested at terms of higher type. In particular, two higher-order closed terms may be observationally congruent but their transforms may not be.

Theorem 3.2 *There exist two closed,* pure *(i.e., containing no constants, conditionals, or recursion) terms, namely*

$$\begin{aligned} M_1 &= \lambda x^{o\to o\to o}.\lambda y^{o\to o}.\lambda z^o.(\lambda w.x\ z\ w)\ (y\ z) \\ M_2 &= \lambda x^{o\to o\to o}.\lambda y^{o\to o}.\lambda z^o.x\ z\ (y\ z), \end{aligned}$$

with $M_1 \equiv^v_{obs} M_2$ but $\overline{M_1} \not\equiv^v_{obs} \overline{M_2}$.

Proof: To show $M_1 \equiv^v_{obs} M_2$, we proceed in a purely operational fashion using Theorem 2.3.[1] We first show that $M_1 \preceq_v M_2$. Pick any values V_1, V_2, and V_3—then $M_2 \twoheadrightarrow_v V'$, $M_2\ V_1 \twoheadrightarrow_v V''$ and $M_2\ V_1\ V_2 \twoheadrightarrow_v V'''$, so all vectors $\vec{V}$ of length 0, 1, or 2 make the statement

$$M_1\ \vec{V} \twoheadrightarrow_v V_0' \text{ implies } M_2\ \vec{V} \twoheadrightarrow_v V_1'$$

hold. Now suppose $M_1\ V_1\ V_2\ V_3 \twoheadrightarrow_v c_l$. Then

$$M_1\ V_1\ V_2\ V_3 \twoheadrightarrow_v (\lambda d. V_1\ V_3\ d)\ (V_2\ V_3) \twoheadrightarrow_v c_l$$

so it must be the case that $V_2\ V_3 \twoheadrightarrow_v V'$ and $V_1\ V_3 \twoheadrightarrow_v V''$ for some values V' and V''. Therefore,

$$\begin{aligned} M_2\ V_1\ V_2\ V_3 &\twoheadrightarrow_v V_1\ V_3\ (V_2\ V_3) \\ &\twoheadrightarrow_v V''\ V' \\ &\twoheadrightarrow_v c_l. \end{aligned}$$

Thus, by Theorem 2.3, $M_1 \preceq_v M_2$. Using a similar argument, one can show $M_2 \preceq_v M_1$.

To show that $\overline{M_1} \not\equiv^v_{obs} \overline{M_2}$, we first reduce $\overline{M_1}$ and $\overline{M_2}$ using $=_v$:

$$\begin{aligned} \overline{M_1} &=_v \lambda\kappa_0.\kappa_0\ (\lambda x.\lambda\kappa_1.\kappa_1\ (\lambda y.\lambda\kappa_2.\kappa_2\ (\lambda z.\lambda\kappa_3.y\ z\ (\lambda n.x\ z\ (\lambda m.m\ n\ \kappa_3))))) \\ \overline{M_2} &=_v \lambda\kappa_0.\kappa_0\ (\lambda x.\lambda\kappa_1.\kappa_1\ (\lambda y.\lambda\kappa_2.\kappa_2\ (\lambda z.\lambda\kappa_3.x\ z\ (\lambda m.y\ z\ (\lambda n.m\ n\ \kappa_3))))) \end{aligned}$$

(where the types have been omitted for clarity.) Intuitively, the difference between $\overline{M_1}$ and $\overline{M_2}$ comes from a difference in the way M_1 and M_2 are reduced when applied to arguments: M_1 evaluates $(y\ z)$ first, while M_2 evaluates $(x\ z)$ first. The typable context

$$\begin{aligned} C[\cdot] &= [\cdot]\ N_0, \text{ where} \\ N_0 &= \lambda p.p\ (\lambda a.\lambda b.c_1)\ N_1, \text{ where} \\ N_1 &= \lambda q.q\ (\lambda a.\lambda b.c_2)\ N_2, \text{ where} \\ N_2 &= \lambda r.r\ c_1\ (\lambda a.a) \end{aligned}$$

distinguishes $\overline{M_1}$ and $\overline{M_2}$, since $C[\overline{M_1}]$ terminates with result c_2 and $C[\overline{M_2}]$ terminates with result c_1:

$$\begin{aligned} C[\overline{M_1}] &=_v (\lambda a.\lambda b.c_2)\ c_1\ (\lambda n.(\lambda a.\lambda b.c_1)\ c_1\ (\lambda m.m\ n\ (\lambda a.a))) \\ &=_v c_2 \\ C[\overline{M_2}] &=_v (\lambda a.\lambda b.c_1)\ c_1\ (\lambda m.(\lambda a.\lambda b.c_2)\ c_1\ (\lambda n.m\ n\ (\lambda a.a))) \\ &=_v c_1. \end{aligned}$$

Thus $\overline{M_1} \not\equiv^v_{obs} \overline{M_2}$.[2] ■

Using a marked language (*cf.* Appendix), one can show that the *untyped* versions of M_1 and M_2 are congruent in any untyped context. Nevertheless, a simple typable context using only numerals distinguishes their transforms.

---

[1] Other techniques exist for verifying congruences: one may rely upon either an adequate or fully-abstract denotational semantics or upon an equational system sound for $\equiv^v_{obs}$ yet strong enough to prove the congruence [12, 20]. Either method rests upon a nontrivial adequacy or soundness proof. Plotkin [18] claims both methods can be used to prove $M_1 \equiv^v_{obs} M_2$, using either pre-domains [22] or Moggi's λ_p [16], but I have not worked through the proofs of adequacy of the pre-domain semantics or soundness of λ_p for $\equiv^v_{obs}$.

[2] In fact, a stronger statement is true: $M_1 \not\preceq_v M_2$ and $M_2 \not\preceq_v M_1$.

The Adequacy Theorem clarifies *why* $\overline{M_1} \not\equiv^v_{obs} \overline{M_2}$: a context with "illegal" continuations distinguishes the continuized terms. One could sensibly argue that $\overline{M_1}$ and $\overline{M_2}$ should *not* be distinguished, since the distinguishing context will never arise under the intended uses of $\overline{M_1}$ and $\overline{M_2}$. But granting this, the theorem nevertheless points out a legitimate concern: what methods shall we use to prove that two terms are congruent with respect to all "legal" contexts, and what exactly are the legal contexts? This question might arise if we wanted to justify a post-transform code optimization in which transformed code $\overline{M}$ was replaced by an "optimized" expression N equivalent to $\overline{M}$ in all legal contexts. For any $\overline{N_0}$, N itself need not equal $\overline{N_0}$.

It is not surprising that Theorem 3.2 has an analog in the `call/cc` setting. Consider, for example, the language $\boldsymbol{\lambda}_c$ with the `call/cc`-like operator $\mathcal{C}$ and the abort operator $\mathcal{A}$ [6, 7, 8]. More precisely, $\boldsymbol{\lambda}_c$ has the same syntax as the untyped version of $\boldsymbol{\lambda}_v$ (*i.e.*, where no variables are decorated with types, and terms need not be well-typed), with the additional terms $\mathcal{C}\ M$ and $\mathcal{A}\ M$. The reduction relation for $\boldsymbol{\lambda}_c$, $\rightarrow_c$, is defined by the rules

$$\begin{array}{lcl@{\qquad}lcl} (\mathcal{A}\ M)\ N & \rightarrow_c & \mathcal{A}\ M & (\mathcal{C}\ M)\ N & \rightarrow_c & \mathcal{C}\ (\lambda\kappa.M\ (\lambda m.\kappa\ (m\ N))) \\ V\ (\mathcal{A}\ M) & \rightarrow_c & \mathcal{A}\ M & V\ (\mathcal{C}\ M) & \rightarrow_c & \mathcal{C}\ (\lambda\kappa.M\ (\lambda v.\kappa\ (V\ v))) \end{array}$$

and the outermost computation rules (which are only applicable in empty contexts)

$$\mathcal{A}\ M \rhd_c M \qquad\qquad \mathcal{C}\ M \rhd_c M\ (\lambda x.\mathcal{A}\ x)$$

in addition to the (untyped versions of) rules of $\rightarrow_v$. Let $\twoheadrightarrow_c$ be the reflexive, transitive closure of $(\rightarrow_c \cup \rhd_c)$, and let $\equiv^c_{obs}$ denote the observational congruence relation on terms of $\boldsymbol{\lambda}_c$ when observing numerals. Then

Theorem 3.3 *If M_1 and M_2 are the terms above, $M_1 \not\equiv^c_{obs} M_2$.*

Proof: Let $C[\cdot]$ be the context $[\cdot]\ (\lambda x.\Omega)\ (\lambda y.\mathcal{C}\ (\lambda x.c_1))\ c_1$. Here, Ω is any divergent term (such as $\mu f.f$.) This context forces $C[M_2]$ to diverge but makes $C[M_1]$ converge to c_1:

$$\begin{array}{rcl} C[M_1] & \twoheadrightarrow_c & (\lambda w.(\lambda x.\Omega)\ c_1\ w)\ ((\lambda y.\mathcal{C}\ (\lambda x.c_1))\ c_1) \\ & \twoheadrightarrow_c & (\lambda w.(\lambda x.\Omega)\ c_1\ w)\ (\mathcal{C}\ (\lambda x.c_1)) \\ & \twoheadrightarrow_c & \mathcal{C}\ (\lambda\kappa.(\lambda x.c_1)\ (\lambda v.\kappa\ ((\lambda w.(\lambda x.\Omega)\ c_1\ w)\ v))) \\ & \rhd_c & (\lambda\kappa.(\lambda x.c_1)\ (\lambda v.\kappa\ ((\lambda w.(\lambda x.\Omega)\ c_1\ w)\ v)))\ (\lambda x.\mathcal{A}\ x) \\ & \twoheadrightarrow_c & (\lambda x.c_1)\ (\lambda v.(\lambda x.\mathcal{A}\ x)\ ((\lambda w.(\lambda x.\Omega)\ c_1\ w)\ v)) \\ & \twoheadrightarrow_c & c_1 \end{array}$$

$$\begin{array}{rcl} C[M_2] & \twoheadrightarrow_c & ((\lambda x.\Omega)\ c_1)\ ((\lambda y.\mathcal{C}\ (\lambda x.c_1))\ c_1) \\ & \twoheadrightarrow_c & \Omega\ ((\lambda y.\mathcal{C}\ (\lambda x.c_1))\ c_1) \\ & \twoheadrightarrow_c & \Omega\ ((\lambda y.\mathcal{C}\ (\lambda x.c_1))\ c_1) \\ & \twoheadrightarrow_c & \ldots \end{array}$$

Thus, $M_1 \not\equiv^c_{obs} M_2$. ∎

The particular terms M_1 and M_2 can also be used to point out problems with continuation semantics. If one bases the semantics of $\boldsymbol{\lambda}_v$ on the transform, *i.e.* the meaning of a

term M is the meaning of $\overline{M}$ in some well-chosen model, two terms may be observationally congruent but fail to be equivalent in the model. The terms M_1 and M_2 again provide the desired example.

Less contrived examples appear in the literature. Meyer and Sieber, for instance, point out that two ALGOL blocks may be observationally congruent but not congruent if **goto** statements are allowed [14]. Since jumps are usually definable in a continuation semantics, the two blocks will not be semantically equivalent. Reasoning principles based on a continuation semantics may thus lead one to conclude facts that are not true about the actual behavior of code.

The failure of familiar reasoning principles seems to be known (albeit informally) in the community of compiler designers. In the presence of control operators or cps-converted code, typical compiler optimizations are unsound and procedure calls are often treated as "black holes." But one need not conclude from the failure of *some* reasoning principles that the situation for continuations is a black hole. There are interesting reasoning principles which hold in continuation settings. For example, consider the $\boldsymbol{\lambda}_v$ terms

$$\begin{aligned} P_1 &= \lambda a.\lambda b.(\lambda x.x)\,((\lambda y.y)\,(a\ b)) \\ P_2 &= \lambda a.\lambda b.(\lambda x.x)\,(a\ b) \end{aligned}$$

that are not provably equivalent using $=_v$. In $\boldsymbol{\lambda}_c$ these two terms are observationally congruent, a fact proven by Felleisen [5] who has developed further principles for proving observational congruences in this setting. A setting involving continuations seems to require a new *theory* for reasoning about code. Such a theory remains to be found.

Chapter 4

Conclusion

Reasoning about the behavior of cps-converted code requires additional assumptions if converted terms are to behave as their direct versions. Theorem 3.2 makes this formal: continuations not arising in continuized contexts may distinguish cps-converted terms. Two possible approaches for a theory of continuations may be based upon this observation.

One approach to a theory of continuations attempts to capture the notion of a "legal" continuation. An algebraic method along these lines is developed in [15] using *retractions*.[1]

Definition 4.1 (Informal) *A* **retraction pair** (i, j) *is a pair of functions such that for any* x, $j\ (i\ x) = x$.

Meyer and Wand define retraction pairs (at all types) that, when applied to a continuized term, supply the right continuations at the right time. Specifically, in the simply-typed, call-by-name λ-calculus with no constants ($\boldsymbol{\lambda}_n$, with $\beta\eta$ equational reasoning $=_n$), Meyer and Wand prove

Theorem 4.2 (Meyer, Wand) *For any type* α, *there exist* $\boldsymbol{\lambda}_n$*-definable retraction pairs* (i_α, j_α) *and* (I_α, J_α), *where* $i_\alpha : \alpha \to \alpha'$, $j_\alpha : \alpha' \to \alpha$, $I_\alpha : \alpha' \to ((\alpha' \to o) \to o)$, *and* $J_\alpha : ((\alpha' \to o) \to o) \to \alpha'$, *namely*

$$
\begin{aligned}
I_\alpha &= \lambda x^{\alpha'}.\lambda\kappa^{\alpha'\to o}.\kappa\ x \\
J_\alpha &= \begin{cases} \lambda x^{(o\to o)\to o}.x\ (\lambda a^o.a) & \text{if } \alpha = o \\ \lambda x^{(\alpha'\to o)\to o}.\lambda b^{\sigma'}.\lambda\kappa^{\tau'\to o}.x\ (\lambda a^{\alpha'}.a\ b\ \kappa) & \text{if } \alpha = \sigma\to\tau \end{cases} \\
i_\alpha &= \begin{cases} \lambda x^o.x & \text{if } \alpha = o \\ \lambda x^{\sigma\to\tau}.\lambda a^{\sigma'}.I_\tau\ (i_\tau\ (x\ (j_\sigma\ a))) & \text{if } \alpha = \sigma\to\tau \end{cases} \\
j_\alpha &= \begin{cases} \lambda x^o.x & \text{if } \alpha = o \\ \lambda x^{(\sigma\to\tau)'}.\lambda a^{\sigma}.j_\tau\ (J_\tau\ (x\ (i_\sigma\ a))) & \text{if } \alpha = \sigma\to\tau \end{cases}
\end{aligned}
$$

Moreover, $M =_n j_\alpha(J_\alpha\ \overline{M})$ *for any closed, pure term* M.

By applying the retractions, one can thus recover the meaning of a direct term from its continuized form.[2]

[1] Inclusive predicates have also been used to establish connections between the direct and continuation semantics of a language [24, 26, 29]. The inclusive predicate approach seems necessary in cases where the denotational domains are built recursively.

[2] Even in the simplified setting of λ_n, we cannot expect to have $\overline{M} = i(M)$ for *any* i. This follows because there are two pure, closed terms M, N where $M =_n N$ but $\overline{M}$ and $\overline{N}$ λ_n-convert to distinct normal forms, namely the terms $M = \lambda a.\lambda b.\lambda c.(\lambda z.a)\ (b\ c)$ and $N = \lambda a.\lambda b.\lambda c.a$. If $\vdash i(M) = \overline{M}$ and $\vdash i(N) = \overline{N}$, then it would follow that $\vdash \overline{M} = \overline{N}$ which, by Statman's typical ambiguity theorem [27], is equationally inconsistent. In the case of λ_v, we similarly cannot have $\overline{M} = i(M)$ for any i by Theorem 3.2.

Theorem 4.2 can be misleading as soon as recursion is added to the language. In the pure simply-typed calculus, call-by-name and call-by-value convertibility coincide since no term causes a divergent computation [2]. Because call-by-*name* equational reasoning is not sound for the observational congruence theory of $\boldsymbol{\lambda}_v$, the retraction pairs above may not be appropriate for $\boldsymbol{\lambda}_v$. In fact, the retraction pairs are no longer retractions: one can only show that $M \preceq_v J_\alpha(I_\alpha\ M)$ and $M \preceq_v j_\alpha(i_\alpha\ M)$. We conjecture that a similar reformulation of Theorem 4.2 holds.[3]

Conjecture 4.3 *For any closed term M of type α, $M \preceq_v j_\alpha(J_\alpha\ \overline{M})$.*

This conjecture does not hold if we reverse the $\preceq_v$:

Theorem 4.4 *Let $\alpha = (o \to o \to o \to o) \to (o \to o) \to o \to (o \to o)$ and let*

$$S = \lambda x.\lambda y.\lambda z.x\ z\ (y\ z)$$

be of type α. Then $j_\alpha(J_\alpha\ \overline{S}) \not\preceq_v S$.

Proof: In the proof of Theorem 3.2, we saw that

$$\overline{S} =_v \lambda\kappa_0.\kappa_0\ (\lambda x.\lambda\kappa_1.\kappa_1\ (\lambda y.\lambda\kappa_2.\kappa_2\ (\lambda z.\lambda\kappa_3.x\ z\ (\lambda m.y\ z\ (\lambda n.m\ n\ \kappa_3)))))$$

Using the fact that $(i\ V)$ is $=_v$ to a value, we can find a simpler form for $j(J\ \overline{S})$:

$$\begin{aligned} j_\alpha(J_\alpha\ \overline{S}) &=_v j_\alpha(\lambda x.\lambda\kappa_1.\kappa_1\ (\lambda y.\lambda\kappa_2.\kappa_2\ (\lambda z.\lambda\kappa_3.x\ z\ (\lambda m.y\ z\ (\lambda n.m\ n\ \kappa_3))))) \\ &=_v \lambda a_1.j(J\ (\lambda\kappa_1.\kappa_1\ (\lambda y.\lambda\kappa_2.\kappa_2\ (\lambda z.\lambda\kappa_3.(i\ a_1)\ z\ (\lambda m.y\ z\ (\lambda n.m\ n\ \kappa_3)))))) \\ &=_v \lambda a_1.\lambda a_2.j(J\ (\lambda\kappa_2.\kappa_2\ (\lambda z.\lambda\kappa_3.(i\ a_1)\ z\ (\lambda m.(i\ a_2)\ z\ (\lambda n.m\ n\ \kappa_3))))) \\ &=_v \lambda a_1.\lambda a_2.\lambda a_3.j(J\ (\lambda\kappa_3.(i\ a_1)\ (i\ a_3)\ (\lambda m.(i\ a_2)\ (i\ a_3)\ (\lambda n.m\ n\ \kappa_3)))) \\ &=_v \lambda a_1.\lambda a_2.\lambda a_3.\lambda a_4. \\ &\qquad j_o(J_o\ (\lambda\kappa.(i\ a_1)\ (i\ a_3)\ (\lambda m.(i\ a_2)\ (i\ a_3)\ (\lambda n.m\ n\ (\lambda a.a\ (i\ a_4)\ \kappa))))) \end{aligned}$$

Thus, in the typable context

$$C[\cdot] = (\lambda x.c_1)\ ([\cdot]\ (\lambda a.\Omega)\ V_1\ V_2)$$

where V_1 and V_2 are closed values, $C[S]$ does not halt but $C[j(J\ \overline{S})] \twoheadrightarrow_v c_1$. ■

It also remains open whether there is a $\boldsymbol{\lambda}_v$-definable j such that $M \equiv^v_{obs} j(\overline{M})$ or even whether an interpretation of such a j exists in one of the standard semantical models of $\boldsymbol{\lambda}_v$.

Another approach to a theory of continuations involves finding general methods for proving observational congruences like P_1 and P_2. A theory in this spirit might exploit the analogy between the three settings of continuation transform, continuation semantics, and `call/cc`-like congruence. We conjecture that a precise match may be found among them.

Conjecture 4.5 *For appropriate choice of direct semantics $D[\![\cdot]\!]$, continuation semantics $C[\![\cdot]\!]$, continuation transform $\overline{M}$, and observational congruence relation $\equiv^c_{obs}$ using* `call/cc`*-like operators in contexts,*

$$\begin{aligned} \overline{M} \equiv^v_{obs} \overline{N} \quad &\textit{iff}\ \ D[\![\overline{M}]\!] = D[\![\overline{N}]\!] \\ &\textit{iff}\ \ C[\![M]\!] = C[\![N]\!] \\ &\textit{iff}\ \ M \equiv^c_{obs} N. \end{aligned}$$

[3] The announcement in [13] of this result is withdrawn.

Establishing this conjecture clearly requires finding a suitably matched triple of transform, continuation semantics, and `call/cc`-like operators, *e.g.*, we obviously must not try to match up a call-by-value transform with a call-by-name direct semantics of a language with `call/cc`-like operators.

Developing reliable principles for reasoning about continuations is the ultimate goal of this research, and it is unclear (at this time) which of these two approaches will yield general principles. Both avenues are being pursued.

Appendix A

Standard Theorems for the Language

The appendix is a compendium of some standard facts about the language $\boldsymbol{\lambda}_v$. Similar results appear in [2, 19, 20] for call-by-name languages; the techniques for proving these facts carry over largely to the case of $\boldsymbol{\lambda}_v$. The results are stated and proved with little comment.

A.1 Church-Rosser Theorem

We follow the proof in [2], using a technique due to Tait and Martin-Löf.

Definition A.1 *The relation $\Rightarrow_p$, the* **parallel reduction relation**, *is defined inductively as follows:*

$$M \Rightarrow_p M \qquad \mathsf{pred}\, c_0 \Rightarrow_p c_0$$

$$\mathsf{succ}\, c_j \Rightarrow_p c_{j+1} \qquad \mathsf{pred}\, c_{j+1} \Rightarrow_p c_j$$

$$\frac{P \Rightarrow_p P'}{\mathsf{cond}\ c_0\ P\ Q \Rightarrow_p P'} \qquad \frac{Q \Rightarrow_p Q'}{\mathsf{cond}\ c_{l+1}\ P\ Q \Rightarrow_p Q'}$$

$$\frac{B \Rightarrow_p B',\ P \Rightarrow_p P',\ Q \Rightarrow_p Q'}{\mathsf{cond}\ B\ P\ Q \Rightarrow_p \mathsf{cond}\ B'\ P'\ Q'} \qquad \frac{M \Rightarrow_p M'}{\lambda x.M \Rightarrow_p \lambda x.M'}$$

$$\frac{M \Rightarrow_p M',\ N \Rightarrow_p N'}{M\ N \Rightarrow_p M'\ N'} \qquad \frac{M \Rightarrow_p M',\ N \Rightarrow_p V}{(\lambda x.M)\ N \Rightarrow_p M'[x := V]}$$

$$\frac{M \Rightarrow_p M'}{\mu f.M \Rightarrow_p \mu f.M'} \qquad \frac{M \Rightarrow_p M',\ \mu f.M \Rightarrow_p N}{\mu f.M \Rightarrow_p M'[f := N]}$$

Lemma A.2 *If $N \Rightarrow_p N'$ and v is any variable, then $M[v := N] \Rightarrow_p M[v := N']$.*

Proof: By structural induction on M. There are two cases to consider in the base case:

Case 1: $M = v$; then $M[v := N] = N \Rightarrow_p N' = M[v := N']$.

Case 2: $M = v'$ for v' some constant or variable not equal to x. Then

$$M[v := N] = v' \Rightarrow_p v' = M[v := N'].$$

There are six cases in the induction case:

Case 1: $M = \lambda v.P$; then $M[v := N] = M \Rightarrow_p M = M[v := N']$.

Case 2: $M = \mu v.P$; then $M[v := N] = M \Rightarrow_p M = M[v := N']$.

Case 3: $M = \lambda y.P$. By induction, $P[x := N] \Rightarrow_p P[x := N']$; thus,

$$M[x := N] \Rightarrow_p M[x := N'].$$

Case 4: $M = \mu f.P$. Similar to the previous case.

Case 5: $M = \mathsf{cond}\ P_1\ P_2\ P_3$; by induction, $P_i[x := N] \Rightarrow_p P_i[x := N']$. Thus,

$$M[x := N] \Rightarrow_p M[x := N'].$$

Case 6: $M = (P_1\ P_2)$; by induction, $P_i[x := N] \Rightarrow_p P_i[x := N']$, so

$$M[x := N] \Rightarrow_p M[x := N'].$$

This completes the proof. ■

Lemma A.3 *Suppose $M \Rightarrow_p M'$ and $N \Rightarrow_p N'$. If v is a λ-variable and N is a value, then $M[v := N] \Rightarrow_p M'[v := N']$. If v is a μ-variable, then $M[v := N] \Rightarrow_p M'[v := N']$.*

Proof: By induction on the definition of $M \Rightarrow_p M'$. In the base case, there are four cases:

Case 1: $M' = M$. By Lemma A.2, $M[v := N] \Rightarrow_p M'[v := N']$.

Case 2: $M = \mathsf{succ}\, c_j$ and $M' = c_{j+1}$. Then $M[v := N] = M \Rightarrow_p M' = M'[v := N']$.

Case 3: $M = \mathsf{pred}\, c_0$ and $M_1 = c_0$. Similar to the previous case.

Case 4: $M = \mathsf{pred}\, c_{j+1}$ and $M_1 = c_j$. Similar to the previous case.

This completes the base case. In the induction case, there are ten cases:

Case 1: $M = \mathsf{cond}\ c_0\ P_2\ P_3$ and $M' = P_2'$. By induction, $P_2[v := N] \Rightarrow_p P_2'[v := N']$. Thus, $M[v := N] \Rightarrow_p M'[v := N']$.

Case 2: $M = \mathsf{cond}\ c_{l+1}\ P_2\ P_3$ and $M' = P_3'$. Similar to the previous case.

Case 3: $M = \mathsf{cond}\ P_1\ P_2\ P_3$ and $M' = \mathsf{cond}\ P_1'\ P_2'\ P_3'$. By induction, we know that $P_i[v := N] \Rightarrow_p P_i'[v := N']$. Thus, $M[v := N] \Rightarrow_p M'[v := N']$.

Case 4: $M = \lambda x.P$ and $M' = \lambda x.P'$. If $v = x$, then

$$M[v := N] = M \Rightarrow_p M' = M'[v := N'].$$

If $v \neq x$, then by induction $P[v := N] \Rightarrow_p P'[v := N']$, so $M[v := N] \Rightarrow_p M'[v := N']$.

Case 5: $M = P\ Q$ and $M' = P'\ Q'$. Similar to Case 3.

Case 6: $M = (\lambda v.P)Q$ and $M' = P'[v := Q']$, where $P \Rightarrow_p P'$, $Q \Rightarrow_p Q'$, and Q' is a value. By the induction hypothesis, $Q[v := N] \Rightarrow_p Q'[v := N']$. Also, since v is a λ-variable, N must a value, so $Q'[v := N']$ must be a value. We can thus use the rules of $\Rightarrow_p$:

$$M[v := N] \Rightarrow_p P'[v := Q'[v := N']] = M'[v := N'].$$

Case 7: $M = (\lambda x.P)Q$ and $M' = P'[x := Q']$, where $v \neq x$, $P \Rightarrow_p P'$, $Q \Rightarrow_p Q'$, and Q' is a value. By the induction hypothesis, $P[v := N] \Rightarrow_p P'[v := N']$ and similarly for Q. If v is a λ-variable, then $Q'[v := N']$ is a value since N is a value by hypothesis; if v is a μ-variable, $Q'[v := N']$ is a value no matter what N is since Q' is a value. Thus,

$$M[v := N] \Rightarrow_p P'[v := N'][x := Q'[v := N']] \Rightarrow_p P'[x := Q'][v := N'] = M'[v := N'].$$

Case 8: $M = \mu f.P$ and $M' = \mu f.P'$. Similar to Case 4.

Case 9: $M = \mu v.P$ and $M' = P'[v := Q']$, where $P \Rightarrow_p P'$, $M \Rightarrow_p Q'$. Note that v cannot be free in Q', since it is not free in M. Thus,

$$M[v := N] = M \Rightarrow_p M' = M'[v := N'].$$

Case 10: $M = \mu f.P$ and $M' = P'[f := Q']$, where $f \neq v$, $P \Rightarrow_p P'$ and $M \Rightarrow_p Q'$. By induction, $M[v := N] \Rightarrow_p Q'[v := N']$ and $P[v := N] \Rightarrow_p P'[v := N']$. Thus,

$$M[v := N] \Rightarrow_p P'[v := N'][f := Q'[v := N']] = P'[f := Q'][v := N'] = M'[v := N'].$$

This completes the proof. ■

Lemma A.4 *The relation $\Rightarrow_p$ is Church-Rosser.*

Proof: Suppose $M \Rightarrow_p M_1$ and $M \Rightarrow_p M_2$. To show that there is an M_3 with $M_1 \Rightarrow_p M_3$ and $M_2 \Rightarrow_p M_3$, proceed by induction on the proof of $M \Rightarrow_p M_1$. In the base case, there are four cases:

Case 1: $M_1 = M$. Pick $M_3 = M_2$; this satisfies the conditions.

Case 2: $M = \mathsf{succ}\, c_j$ and $M_1 = c_{j+1}$. Pick $M_3 = c_{j+1}$; since M_2 can only be M or M_1, this choice of M_3 suffices.

Case 3: $M = \mathsf{pred}\, c_0$ and $M_1 = c_0$. Pick $M_3 = c_0$; as with the previous case, this M_3 meets the conditions since M_2 can only be M or M_1.

Case 4: $M = \mathsf{pred}\, c_{j+1}$ and $M_1 = c_j$. Pick $M_3 = c_j$; again, this choice suffices.

This completes the base case. In the induction case, there are eight cases to consider:

Case 1: $M = \mathsf{cond}\, c_0\, P_2\, P_3$ and $M_1 = P_2'$. Then M_2 is either P_2'' or $\mathsf{cond}\, c_0\, P_2''\, P_3''$. By the induction hypothesis, there is a P_2''' with $P_2' \Rightarrow_p P_2'''$ and $P_2'' \Rightarrow_p P_2'''$. Then picking M_3 to be P_2''' works.

Case 2: $M = \mathsf{cond}\, c_{l+1}\, P_2\, P_3$ and $M' = P_3'$. Similar to the previous case.

Case 3: $M = \mathsf{cond}\, P_1\, P_2\, P_3$ and $M_1 = \mathsf{cond}\, P_1'\, P_2'\, P_3'$, where $P_i \Rightarrow_p P_i'$. Then M_2 is either P_2'', P_3'', or $\mathsf{cond}\, P_1''\, P_2''\, P_3''$. By the induction hypothesis, there are P_i''' with $P_i' \Rightarrow_p P_i'''$ and $P_i'' \Rightarrow_p P_i'''$. Then picking M_3 to be either P_2''', P_3''', or $\mathsf{cond}\, P_1'''\, P_2'''\, P_3'''$ (as appropriate) works.

Case 4: $M = \lambda x.P$ and $M_1 = \lambda x.P'$, where $P \Rightarrow_p P'$. Then M_2 must also be of the form $\lambda x.P''$. By induction, pick P''' where $P' \Rightarrow_p P'''$ and $P'' \Rightarrow_p P'''$. Then $M_3 = \lambda x.P'''$ will work.

Case 5: $M = (\lambda x.P)Q$ and $M_1 = P'[x := Q']$, where $P \Rightarrow_p P'$, $Q \Rightarrow_p Q'$, and Q' is a value. There are two subcases:

Subcase i: $M_2 = (\lambda x.P'')Q''$. By induction, there are P''' and Q''' with $P' \Rightarrow_p P'''$ and $P'' \Rightarrow_p P'''$; pick Q''' similarly. Since Q' is a value, Q''' must also be a value. Pick $M_3 = P'''[x := Q''']$; $M_2 \Rightarrow_p M_3$ easily, and $M_1 \Rightarrow_p M_3$ by Lemma A.3.

Subcase ii: $M_2 = P''[x := Q'']$. By induction, there are two terms P''' and Q''' with $P' \Rightarrow_p P'''$ and $P'' \Rightarrow_p P'''$; pick Q''' similarly. Since Q' is a value, Q''' must also be a value. Pick $M_3 = P'''[x := Q''']$; then both $M_2 \Rightarrow_p M_3$ and $M_1 \Rightarrow_p M_3$ by Lemma A.3.

Case 6: $M = P\ Q$ and $M_1 = P'\ Q'$, where $P \Rightarrow_p P'$ and $Q \Rightarrow_p Q'$. There are two subcases:

Subcase i: $M_2 = P''\ Q''$. By induction, pick P''' with $P' \Rightarrow_p P'''$ and $P'' \Rightarrow_p P'''$; pick Q''' similarly. Then $M_3 = P'''\ Q'''$ works.

Subcase ii: $P = \lambda x.R$ and $M_2 = R''[x := Q'']$, for Q'' a value. Then $P' = \lambda x.R'$. By induction, pick R''' with $R' \Rightarrow_p R'''$ and $R'' \Rightarrow_p R'''$; pick Q''' similarly. As above, note that Q''' must be a value. Picking M_3 to be $R'''[x := Q''']$ works, since $M_1 \Rightarrow_p M_3$ easily and $M_2 \Rightarrow_p M_3$ by Lemma A.3.

Case 7: $M = (\mu f.P)$ and $M_1 = \mu f.P'$.

Subcase i: $M_2 = \mu f.P''$. By induction, pick P''' as before; then $M_3 = \mu f.P'''$ works.

Subcase ii: $M_2 = P''[f := Q'']$, where $P \Rightarrow_p P''$ and $M \Rightarrow_p Q''$. By induction, pick P''' as before, and let $M_3 = P'''[f := Q'']$; $M_1 \Rightarrow_p M_3$ by the rules of $\Rightarrow_p$, and $M_2 \Rightarrow_p M_3$ by Lemma A.3.

Case 8: $M = (\mu f.P)$ and $M_1 = P'[f := Q']$, where $P \Rightarrow_p P'$ and $M \Rightarrow_p Q'$.

Subcase i: $M_2 = \mu f.P''$. By induction, pick P''' as before and pick Q''' where $Q' \Rightarrow_p Q'''$ and $M_2 \Rightarrow_p Q'''$. Then $M_3 = P'''[f := Q''']$ works, since $M_1 \Rightarrow_p M_3$ by Lemma A.3 and $M_2 \Rightarrow_p M_3$ by the rules of $\Rightarrow_p$.

Subcase ii: $M_2 = P''[f := Q'']$, where $P \Rightarrow_p P''$ and $M \Rightarrow_p Q''$. By induction, pick P''' and Q''' as before, and let $M_3 = P'''[f := Q''']$; then $M_1 \Rightarrow_p M_3$ and $M_2 \Rightarrow_p M_3$ by Lemma A.3.

■

Definition A.5 *$M \Rightarrow_v N$ iff $M =_v N$ using no instance of the symmetry axiom.*

Lemma A.6 *$M \Rightarrow_p^* N$ iff $M \Rightarrow_v N$.*

Proof: Let $\overrightarrow{=}_v$ be the relation of doing 0 or 1 $=_v$ steps without using the symmetry axiom. When treated as sets, the relations satisfy

$$\overrightarrow{=}_v \subseteq \Rightarrow_p \subseteq \Rightarrow_v .$$

Since $\Rightarrow_v$ is the transitive closure of $\overrightarrow{=}_v$, it is also the transitive closure of $\Rightarrow_p$. ■

Theorem A.7 *The relation $\Rightarrow_v$ is Church-Rosser.*

Proof: Since $\Rightarrow_p$ is Church-Rosser, its transitive closure $\Rightarrow_p^*$ is also [2]. By Lemma A.6, $\Rightarrow_v$ is Church-Rosser. ■

The most important consequence of the Church-Rosser theorem is

Theorem 2.2 *If $M =_v N$, then $M \equiv^v_{obs} N$.*

Proof: If $M \Rightarrow_v N$, then $C[M] \Rightarrow_v C[N]$. Thus, if either $C[M]$ or $C[N]$ reduce to c_l under $\twoheadrightarrow_v$, both of them will by Theorem A.7. The theorem then follows by an easy induction on the number of occurrences of the symmetry rule. ■

A.2 Applicative Congruence

At each step, the relation $\rightarrow_v$ reduces only one subterm. We call that subterm the **active** subterm [20]. An examination of the operational rules indicates that

Definition A.8 *The* **active subterm** *of a non-value, closed term M is*

- *M if M is of the form $(\mathsf{succ}\, c_l)$, $(\mathsf{pred}\, c_l)$, $(\mathsf{cond}\ c_l\ M_0\ M_1)$, $(\mu f.M_0)$, or $((\lambda x.M_0)\ V)$ for V a value; or*
- *The active subterm in M', where M' is closed and not a value, if M has the form $(\mathsf{succ}\, M')$, $(\mathsf{pred}\, M')$, $(\mathsf{cond}\ M'\ M_0\ M_1)$, $(M'\ M_0)$, or $((\lambda x.M_0)\ M')$.*

This definition matches the informal description of what the active subterm should be:

Lemma A.9 *Let M be a closed subterm of a non-value, closed term $C[M]$, where M contains the active subterm of $C[M]$ and $C[\cdot]$ has only one hole. Then if $M \rightarrow_v M'$,*

$$C[M] \rightarrow_v C[M'].$$

Proof: An easy structural induction on $C[\cdot]$. ■

Lemma A.10 *Let M be a closed subterm of a non-value, closed term $C[M]$, where M contains the active subterm of $C[M]$ and $C[\cdot]$ has only one hole. Then if $M \twoheadrightarrow_v M'$,*

$$C[M] \twoheadrightarrow_v C[M'].$$

Proof: By induction on n, where

$$M = M_0 \rightarrow_v M_1 \rightarrow_v M_2 \rightarrow_v \ldots \rightarrow_v M_n = M'.$$

The base case, where $n = 0$, is trivial, so we proceed to the induction case. By the induction hypothesis, $C[M_0] \twoheadrightarrow_v C[M_{n-1}]$. A structural induction on $C[\cdot]$ shows that M_{n-1} contains the active subterm in $C[M_{n-1}]$; thus, by Lemma A.9, $C[M_{n-1}] \rightarrow_v C[M_n]$ so the lemma holds. ■

Lemma A.11 (Activity) *Let M be a closed term of type σ and $C[\cdot]$ be a closed context with holes of type σ. Then $C[M] \twoheadrightarrow_v c_l$ iff either*

1. *$C[M'] \twoheadrightarrow_v c_l$ for any M'; or*
2. *$(\lambda x.C[x])\ M \twoheadrightarrow_v c_l$.*

Proof: ($\Rightarrow$) If M is a value, condition (2) holds immediately. So suppose M is not a value. We use a marking technique due to Bard Bloom. Add to the language $\boldsymbol{\lambda}_v$ the term $\#M$ for any term M, and add the reduction rules (and only these rules)

$$\#V \rightarrow_v V,\ V \text{ a value}$$

$$\frac{M \rightarrow_v N}{\#M \rightarrow_v \#N}$$

to the definition of the $\rightarrow_v$ relation. Note that these rules do not change the computational behavior of $\boldsymbol{\lambda}_v$, *i.e.*, $M \twoheadrightarrow_v c_l$ iff $\mathsf{erase}(M) \twoheadrightarrow_v c_l$ where $\mathsf{erase}(M)$ is the result of erasing all marks in M.

Proceed by induction on the number of occurrences of $\#M$ in $C[\#M]$. The base case ($n = 0$) is trivial. In the induction case, suppose that $C[\#M] \twoheadrightarrow_v c_l$. Let C' be the first term whose active subterm is contained in a subterm of $\#M$; if there is no such C', then condition (1) holds. Let $C' = D[\#M]$, where $D[\cdot]$ has one hole. Since $\#M \twoheadrightarrow_v V$ for some unmarked value V, using the version of Lemma A.10 for the marked language we conclude that $C' \twoheadrightarrow_v D[V]$. Note that there is a context $E[\cdot]$ with $n-1$ holes such that $D[V] = E[\#M]$. The context $E[\cdot]$ has the property that

$$\begin{aligned}(\lambda x.C[x])\, M &\twoheadrightarrow_v (\lambda x.C[x])\, V \\ &\twoheadrightarrow_v C[V] = E[V].\end{aligned}$$

By the induction hypothesis, either $E[M'] \twoheadrightarrow_v c_l$ for any M' or $(\lambda x.E[x])\, M \twoheadrightarrow_v c_l$. If the first condition is true, then $(\lambda x.C[x])\, M \twoheadrightarrow_v E[V] \twoheadrightarrow_v c_l$ so $(\lambda x.C[x])\, M \twoheadrightarrow_v c_l$. If the second condition is true, then $(\lambda x.C[x])\, M \twoheadrightarrow_v E[V] \twoheadrightarrow_v c_l$ since $(\lambda x.E[x])\, M \twoheadrightarrow_v E[V] \twoheadrightarrow_v c_l$.

($\Leftarrow$) Suppose $(\lambda x.C[x])\, (\#M) \twoheadrightarrow_v c_l$. Again, proceed by induction on the number of occurrences of $\#M$ in $C[\#M]$. The base case ($n = 0$) is trivial, so consider the induction case. Examine the reduction sequence for $C[\#M]$, and pick the first C' whose active subterm is contained in a $\#M$; if there is no such C', then $C[M'] \twoheadrightarrow_v c_l$ for any M' so $C[M] \twoheadrightarrow_v c_l$. Let $C' = D[\#M]$, where $D[\cdot]$ is a context with one hole and $\#M$ contains the active subterm in $D[\#M]$. Then

$$D[\#M] \twoheadrightarrow_v D[V] = E[\#M]$$

where V is a value with $\#M \twoheadrightarrow_v V$ and $E[\cdot]$ is an unmarked context with $n-1$ holes. Since

$$(\lambda x.E[x])\, (\#M) \twoheadrightarrow_v c_l$$

by the induction hypothesis $E[\#M] \twoheadrightarrow_v c_l$. Since $C[\#M] \twoheadrightarrow_v E[\#M]$, $C[\#M] \twoheadrightarrow_v c_l$. ∎

Lemma A.12 *Let V_0 and V_1 be closed values of the same type. If $V_0\, V' \preceq_v V_1\, V'$ for any closed value V', then $V_0 \preceq_v V_1$.*

Proof: Again, we use the marking technique. Suppose $C[\#V_0] \twoheadrightarrow_v c_l$ assuming, without loss of generality, that $C[\cdot]$ contains no marked terms. We proceed by induction on n, where an active subterm of the form $((\#V_0)\, V')$ (V' any closed value) appears n times in the reduction.

In the base case, $n = 0$; thus, $C[V_1] \twoheadrightarrow_v c_l$ trivially. In the induction case, pick the first term C' in the reduction sequence with an active subterm of the form $((\#V_0)\ V')$. Let $C' = D[(\#V_0)\ V']$, where $D[\cdot]$ has one hole and the hole is active. We know that $D[(\#V_0)\ V'] \twoheadrightarrow_v c_l$. By hypothesis, $D[V_1\ V'] \twoheadrightarrow_v c_l$. Let $E[\cdot]$ be the context where $D[V_1\ V'] \rightarrow_v E[\#V_0]$ and $E[\cdot]$ has no occurrences of $\#V_0$. Since $E[\#V_0] \twoheadrightarrow_v c_l$ with $(n-1)$ reductions of the form $((\#V_0)\ V'')$ for some closed value V'', by induction we conclude that $E[V_1] \twoheadrightarrow_v c_l$. The lemma now follows since $C[V_1] \twoheadrightarrow_v E[V_1] \twoheadrightarrow_v c_l$. ■

Theorem 2.3 *Let M and N be closed terms of the same type. Then $M \preceq_v N$ iff, for all vectors $\vec{V}$ of closed values,*

$$M\ \vec{V} \twoheadrightarrow_v V_0' \text{ implies } N\ \vec{V} \twoheadrightarrow_v V_1' \text{ and } V_0' = V_1' \text{ if either is a numeral.}$$

Proof: $(\Rightarrow)$ Trivial.

$(\Leftarrow)$ By induction on types. Consider first the base case, where M and N are of type o. Suppose $C[\cdot]$ is a context in which $C[M] \twoheadrightarrow_v c_l$; then we know by the Activity Lemma that either $C[M'] \twoheadrightarrow_v c_l$ for any M' or $(\lambda x.C[x])\ M \twoheadrightarrow_v c_l$. In the first case, $C[N] \twoheadrightarrow_v c_l$ trivially. In the second case, since M must reduce to some numeral, say $c_{l'}$, it must be the case that $N \twoheadrightarrow_v c_{l'}$. Thus, $C[N] \twoheadrightarrow_v c_l$, so $M \preceq_v N$.

In the induction case, again consider any $C[\cdot]$ where $C[M] \twoheadrightarrow_v c_l$. Then by the Activity Lemma, either $C[M'] \twoheadrightarrow_v c_l$ for any M' or $(\lambda x.C[x])\ M \twoheadrightarrow_v c_l$. In the first case, $C[N] \twoheadrightarrow_v c_l$ trivially. In the second case, $M \twoheadrightarrow_v V_0$ for some closed value V_0. Since for any vector $\vec{V}$ of closed values,

$$M\ \vec{V} \twoheadrightarrow_v V_0' \text{ implies } N\ \vec{V} \twoheadrightarrow_v V_1',$$

it follows (using the empty vector) that $N \twoheadrightarrow_v V_1$ for some closed value V_1. By hypothesis, for any closed value V',

$$(M\ V')\ \vec{V} \twoheadrightarrow_v c_l \text{ implies } (N\ V')\ \vec{V} \twoheadrightarrow_v c_l.$$

By the induction hypothesis, $M\ V' \preceq_v N\ V'$ for any V'. By Lemma A.12, since $M \equiv_{obs}^v V_0$ and $N \equiv_{obs}^v V_1$, $M \preceq_v N$. ■

Bibliography

[1] Andrew Appel and Trevor Jim. Continuation-passing, closure-passing style. In 16th *Symp. Principles of Programming Languages*, pages 293–302, ACM, January 1989.

[2] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in Logic*, North-Holland, 1981. Revised Edition, 1984.

[3] Bard Bloom. Can LCF be topped? In 3rd *Symp. Logic in Computer Science*, pages 282–295, IEEE, 1988.

[4] Bard Bloom and Jon G. Riecke. LCF should be lifted. 1988. Unpublished manuscript.

[5] Matthias Felleisen. April 1988. Private communication.

[6] Matthias Felleisen. λ-V-CS: an extended λ-calculus for Scheme. In *Proc. of Conf. LISP and Functional Programming*, pages 72–85, ACM, July 1988.

[7] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Symp. Logic in Computer Science*, pages 131–141, IEEE, 1986.

[8] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Sci.*, 52:205–237, 1987.

[9] Michael J. Fischer. Lambda calculus schemata. In *Conf. on Proving Assertions About Programs*, pages 104–109, ACM, Las Cruces, NM, 1972.

[10] Daniel P. Friedman. Applications of continuations. January 1988. Presented at POPL 1988.

[11] Michael J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[12] Albert R. Meyer. Semantical paradigms: notes for an invited lecture, with two appendices by Stavros Cosmadakis. In 3rd *Symp. Logic in Computer Science*, pages 236–255, IEEE, 1988.

[13] Albert R. Meyer and Jon G. Riecke. Continuations may be unreasonable. In *Proc. of Conf. LISP and Functional Programming*, pages 63–71, ACM, July 1988.

[14] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables: preliminary report. In 15th *Symp. Principles of Programming Languages*, pages 191–203, ACM, 1988.

[15] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Proceedings of the Conference on Logics of Programs, 1985, Lecture Notes in Computer Science 193*, pages 219–224, Springer-Verlag, 1985.

[16] Eugenio Moggi. *The Partial Lambda Calculus.* Ph.D. thesis, Edinburgh University, 1987.

[17] C.-H. Luke Ong. Fully abstract models of the lazy lambda calculus. In *Proc. of Symp. Foundations of Computer Science*, pages 368–376, IEEE, October 1988.

[18] Gordon D. Plotkin. May 1988. Message to the TYPES@THEORY.LCS.MIT.EDU electronic mailing list.

[19] Gordon D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoretical Computer Sci.*, 1:125–159, 1975.

[20] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–255, 1977.

[21] Gordon D. Plotkin. *A structural approach to operational semantics.* Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[22] Gordon D. Plotkin. Types and partial functions. February 1984. Handwritten manuscript.

[23] Jonathan Rees and William Clinger. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21:37–79, 1986.

[24] John C. Reynolds. On the relation between direct and continuation semantics. In *Proceedings of the Second Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science 14*, pages 141–156, Springer-Verlag, 1974.

[25] Dana Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. 1969. Manuscript, Oxford Univ.

[26] Ravi Sethi and Adrian Tang. Constructing call-by-value continuation semantics. *Journal of the ACM*, 27:580–597, 1980.

[27] Richard Statman. λ-definable functionals and $\beta\eta$-conversion. *Archiv Math. Logik Grundlagenforsch.*, 22:1–6, 1982.

[28] Guy L. Steele. *Rabbit: A Compiler for Scheme.* Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, 1978.

[29] Joseph E. Stoy. The congruence of two programming language definitions. *Theoretical Computer Science*, 13:151–174, 1981.

[30] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[31] C. Strachey and C.P. Wadsworth. *Continuations: A Mathematical Semantics for Handling Full Jumps.* Technical Report PRG-11, Oxford University Computing Laboratory, 1974.

[32] C.P. Wadsworth. The relation between computational and denotational properties for Scott's D_{∞} models. *SIAM J. Computing*, [illegible].

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-459	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Should A Function Continue?

12. PERSONAL AUTHOR(S)
Riecke, J.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	1989 September	31

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Continuations, λ-calculus, operational semantics, denotational semantics

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We show that two λ-calculus terms can be observationally congruent (i.e., agree in all contexts) but their continuation-passing transforms may not be. We also show that two terms may be congruent in all untyped contexts but fail to be congruent in a language with call/cc operators, and that two terms may have the same meaning in a direct semantics but not in a continuation semantics. Hence, familiar resoning about terms may be unsound in a setting with continuations, demonstrating the need for a theory of continuations.

This document contains corrections to the original thesis submitted in January of 1989. Portions of this report previously appeared in [13].

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

★U.S. Government Printing Office: 1985-507-047

Unclassified