Continuations May Be Unreasonable: Preliminary Report

Albert R. Meyer* Jon G. Riecke
MIT Laboratory for Computer Science

Abstract: We show that two lambda 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 calculus with call/cc operators. Hence, familiar reasoning about functional terms may be unsound if the terms use continuations explicitly or access them implicitly through new operators. We then examine one method of connecting terms with their continuized form, extending the work of Meyer and Wand [8].

1 Introduction

Continuations raise new issues in reasoning about programs, but there seems to be some disagreement as to how "well-known" some of the difficulties are. Our purpose is to make these difficulties precise, clarifying the challenges to be met by a theory of continuations. We report some progress towards such a theory in the second half of the paper.

Continuations appear in at least three settings in the theory and practice of programming languages. First, continuations form the basis of *continuation-style* denotational semantics, which are used to model languages with *goto*'s and other imperative features [16,21,23]. Second, in languages such as LISP, continuations may be used to direct program flow: the continuation implicitly found in a program may be accessed through the control operators *catch* and *throw* [20] and *call-with-current-continuation* (call/cc) [13]. Finally, continuations have uses in compilers, which may apply a *continuation-passing style* (cps) transform as a fundamental step in compilation [6,14,19].

We concentrate on the third setting of cps transforms, since the problems in reasoning about cps terms will be indicative of the problems arising in other settings. We also conjecture at the end of this section precisely how the three settings interrelate. We review in Section 2 the continuation transform, $\overline{M}$, of a lambda expression M [10]. Much of the work using continuations has this transform at its core (*cf.* [4,5,8,17,19]). For definiteness, we re-

*Both authors were supported in part by NSF Grant No. 8511190-DCR and by ONR grant No. N00014-83-K-0125. Riecke was also supported by an NSF Fellowship.

strict ourselves to a simply typed call-by-value lambda calculus [1,10] with arithmetic and recursion operators, which we call $\boldsymbol{\lambda}_v$. (See the Appendix for a complete definition of the language.) We write $Eval_v(M)$ for the term to which M evaluates and write $\boldsymbol{\lambda}_v \vdash M = N$ when M converts to N under the rules of $\boldsymbol{\lambda}_v$.

The familiar notion of call-by-value conversion plays a key role in understanding the transform. Two theorems, whose exact statements we postpone to Section 2.2, reveal the "correctness" of the transform [6,10]:

Theorem 1 (Informal) *The transform preserves $\boldsymbol{\lambda}_v$-convertibility,* i.e., *if $\boldsymbol{\lambda}_v \vdash M = N$ then $\boldsymbol{\lambda}_v \vdash \overline{M} = \overline{N}$.*

Definition 2 *A* complete program *is a closed lambda term of integer type.*

Theorem 3 (Informal) *Given any complete program, M, the cps version, $\overline{M}$, produces the same "printable value,"* viz., *integer numeral, if any, as M.*

Theorem 3 crystallizes the correctness of the cps-conversion step in compilers (*e.g.*, in the Rabbit compiler for Scheme [19].) If this theorem was false, a compiler that continuized programs could produce object code that produced a different printable output than programs that were directly interpreted!

Both theorems allow some "direct" reasoning about M to be carried over to the "continuized code," $\overline{M}$. Nevertheless there are equations which are true computationally but which do not follow by the kind of symbolic evaluation provided by Theorems 1 and 3. By "true computationally" we mean observationally congruent [7]:

Definition 4 (Informal) *Two terms M, N are* $\boldsymbol{\lambda}_v$-observationally congruent *(written $M \equiv_v N$) if they behave the same way in all complete program contexts under $\boldsymbol{\lambda}_v$-convertibility.*

We have as a simple corollary of Theorem 3 that if $\overline{M} \equiv_v \overline{N}$, then $M \equiv_v N$ because any context distinguishing M and N would transform into a context distinguishing $\overline{M}$ and $\overline{N}$.

Observational congruence has a rich theory associated with it. The principles of continuity and fixed point induction in denotational semantics, along with pure lambda calculus reasoning, can be used to prove observational congruences. These tools help justify the informal observational congruence reasoning used by programmers and also make their assumptions (*e.g.*, "stack discipline," "lexical scoping") precise and clear.

For example, let

$$\begin{aligned} M_1 &= \lambda a.\, \lambda b.\, \lambda c.\, (\lambda x.\, (b\ a)\ x)\ (c\ a), \\ M_2 &= \lambda a.\, \lambda b.\, \lambda c.\, (b\ a)\ (c\ a). \end{aligned}$$

One can prove (though the proof is nontrivial) that these terms reduce to the same integers in all contexts under call-by-value; they are thus observationally congruent even though they are not $\boldsymbol{\lambda}_v$-convertible.

The problem is that their transforms are not observationally congruent.

Theorem 5 *Let M_1, M_2 be the terms above. Then*

$$M_1 \equiv_v M_2 \quad \textit{but} \quad \overline{M_1} \not\equiv_v \overline{M_2}.$$

That is, M_1 and M_2 are congruent under evaluation by a standard direct interpreter, but their cps transforms yield distinguishable visible outcomes in some context. Hence, reasoning which is sound when M_1 and M_2 are interpreted directly may be unsound when they are continuized. Note that these terms are typable with the same most general (ML-style [9]) type, and so they will provide the desired counterexamples for both typed and untyped lambda calculi.

In Section 3 we clarify *why* $\overline{M_1} \not\equiv_v \overline{M_2}$: 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 $\overline{M_1}, \overline{M_2}$. 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 are the legal contexts anyway? This question might arise if we wanted to justify a post-transform code optimization in which transformed code $\overline{M}$ was replaced by a "optimized" expression N equivalent to $\overline{M}$ in all legal contexts. N itself need not equal $\overline{N_0}$ for any N_0.

Moreover, the example in Theorem 5 takes on further significance in the call/cc setting. In a lambda calculus with call/cc-like operators, *e.g.*, $\boldsymbol{\lambda}_c$ [4,5,3], we can distinguish terms that $\boldsymbol{\lambda}_v$ cannot. More precisely, let $M \equiv_c N$ be the notation for observational congruence using contexts with the operators of $\boldsymbol{\lambda}_c$. Then

Theorem 6 *Let M_1 and M_2 be the terms above. Then $M_1 \not\equiv_c M_2$.*

Proof: Using the operator $\mathcal{C}$ [4,5], the context

$$C[\cdot] = [\cdot]\, 1\, (\lambda x.\Omega)\, (\lambda y.\mathcal{C}(\lambda x.1))$$

forces $C[M_2]$ to diverge but makes $C[M_1]$ converge to 1. ■

A setting involving continuations thus seems to require a different theory for understanding the behavior of code. We have been told that this kind of example is "well-known" and that it is understood in the community of compiler designers that typical compiler optimizations are unsound, and that procedure calls must be treated as "black holes," in the presence of continuations.

However, we need not conclude from the failure of some reasoning principles that the situation for continuations is a black hole. There are interesting reasoning principles, beyond $\boldsymbol{\lambda}_v$-conversion, which hold in a continuation setting. For example, let

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

One can verify that $\boldsymbol{\lambda}_v \vdash \overline{P_1} = \overline{P_2}$. In light of the conjecture below, this suggested that in fact $P_1 \equiv_c P_2$, which was confirmed by Felleisen [2] who has developed further principles for proving $\equiv_c$.

The unsoundness of familiar functional reasoning indicates that a *theory* of continuations remains to be found. In the first half of this paper, we formally state the theorems mentioned in this section. The second half of the paper reports partial results towards a theory of continuations, exploring the Meyer-Wand approach of *retractions* [8].

The three settings of continuation transform, continuation semantics, and call/cc-like congruence indicated above are clearly analogous. We conjecture that a precise match may be found among them. More formally,

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

$$\begin{aligned} \overline{M} \equiv_v \overline{N} \quad & \textit{iff}\ \ D[\![\overline{M}]\!] = D[\![\overline{N}]\!] \\ & \textit{iff}\ \ C[\![M]\!] = C[\![N]\!] \\ & \textit{iff}\ \ M \equiv_c N. \end{aligned}$$

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 call/cc-like operator. We have not yet examined the conjecture with enough care to be very confident about it, but so far it has proved to be a reliable guide to the existence of examples such as M_1, M_2 and P_1, P_2 above.

2 Continuation Transform

2.1 Definition

To specify the continuation transform on pure terms, we need only the clauses [8,10]:

$$\begin{aligned} \overline{x} &= \lambda\kappa.\ \kappa\ x \\ \overline{\lambda x.\ M} &= \lambda\kappa.\ \kappa\ (\lambda x.\ \overline{M}) \\ \overline{MN} &= \lambda\kappa.\ \overline{M}\ (\lambda m.\ \overline{N}\ (\lambda n.\ m\ n\ \kappa)) \end{aligned}$$

It can be read as a transform of untyped terms, or as a polymorphic description of the transform of typed terms (except in the typed case, we must remember that x's on the left- and right-hand sides of the equation have different types.) We defer the continuized versions of recursion, conditional expressions, and constants to the Appendix. With extensions to account for "define" and other constructs, this transform is a slightly less optimized version of the cps-conversion used in the Scheme compiler Rabbit [19].

To see exactly how cps-conversion works, consider the term M_1 given in Section 1. The transform of the subterm $(c\ a)$, in particular, is

$$\lambda\kappa_0.(\lambda\kappa_1.\kappa_1\ c)\ (\lambda m.\ (\lambda\kappa_2.\kappa_2\ a)(\lambda n.\ m\ n\ \kappa_0))$$

(with the κ's indexed for clarity.) Applying the conversion rules of $\boldsymbol{\lambda}_v$, this term is equivalent to the term $\lambda\kappa_0.c\ a\ \kappa_0$. Carrying out the cps-conversion completely for M_1 and M_2 and reducing the transforms to normal form yields

$$\begin{aligned} \overline{M_1} &= \lambda\kappa_0.\kappa_0\ (\lambda a.\lambda\kappa_1.\kappa_1\ (\lambda b.\lambda\kappa_2. \\ &\quad \kappa_2\ (\lambda c.\lambda\kappa_3.c\ a \\ &\qquad (\lambda n.\ b\ a\ (\lambda m.m\ n\ \kappa_3))))), \\ \overline{M_2} &= \lambda\kappa_0.\kappa_0\ (\lambda a.\lambda\kappa_1.\kappa_1\ (\lambda b.\lambda\kappa_2. \\ &\quad \kappa_2\ (\lambda c.\lambda\kappa_3.b\ a \\ &\qquad (\lambda m.\ c\ a\ (\lambda n.m\ n\ \kappa_3))))). \end{aligned}$$

In the typed calculus, the transform also changes the type of an expression. For any type α, define α' inductively by

$$\begin{aligned} o' &= o \\ (\alpha \to \beta)' &= \alpha' \to (\beta' \to o) \to o \end{aligned}$$

where o is the integer type. With a suitable addition of types to the definition of the transform above, if the type of M is α, then the type of $\overline{M}$ is $(\alpha' \to o) \to o$ [8].

2.2 Properties of the Transform

Call-by-value convertibility is preserved by the transform:

Theorem 1 (Plotkin) *If M, N are closed terms and $\boldsymbol{\lambda}_v \vdash M = N$, then $\boldsymbol{\lambda}_v \vdash \overline{M} = \overline{N}$.*

Proof: We modify Plotkin's proof in [10] to work in our typed case. ■

Thus, optimizations provable on direct terms using $\boldsymbol{\lambda}_v$-convertibility will be reproducible on transformed terms.

For practical applications, the "correctness" of the transform plays an important role. Since we intend to use the transform in compilers, we expect transformed complete *programs* to return the correct printable values:

Theorem 3 (Adequacy [6]) *If M is a complete program and c is a constant of type o, then*

$$Eval_v(M) = c \text{ iff } Eval_v(\overline{M}\ (\lambda x^o.x)) = c.$$

Proof: We modify Plotkin's proof [10], a reworking of the original proof in [6]. ■

Complete programs thus behave the same as their continuized versions, so compilers that rely on cps-conversion (*e.g.*, the Rabbit compiler for Scheme [19]) can be regarded as "correct" in this sense.

3 Operational Semantics and Continuations

When reasoning about code, one usually identifies terms that behave the same way in all contexts [11]. Formally,

Definition 4 *Two terms M, N of the same type are* observationally congruent *if, for any context $C[\cdot]$ (a term with a "hole" in it) such that $C[M]$ and $C[N]$ are complete programs,*

$$\boldsymbol{\lambda}_v \vdash C[M] = c \text{ iff } \boldsymbol{\lambda}_v \vdash C[N] = c.$$

for all numerals c.

In fact, one can modify this definition to observe only one numeral, some numerals, or simply termination: each definition yields the same observational congruence theory. Also, note that the definition can be parameterized by the set of allowable contexts, *e.g.*, typed or untyped contexts. A different set of contexts will in general change the observational congruence theory.

We then have

Theorem 5 *There exist two* pure *(i.e., constant- and recursion-free), closed terms M_1 and M_2 with $M_1 \equiv_v M_2$ but $\overline{M_1} \not\equiv_v \overline{M_2}$.*

Proof: We pick M_1, M_2 as in Section 1. One can then show that

$$\begin{array}{rcl} \boldsymbol{\lambda}_v \vdash C[M_1] = c & \Longrightarrow & \boldsymbol{\lambda}_v \vdash C[M_2] = c, \\ \boldsymbol{\lambda}_v \vdash C[M_2] = c & \Longrightarrow & \boldsymbol{\lambda}_v \vdash C[M_1] = c \end{array}$$

by induction on the length of $\boldsymbol{\lambda}_v$-proof. Thus, $M_1 \equiv_v M_2$.

However, $\overline{M_1} \not\equiv_v \overline{M_2}$: using the typable context

$$\begin{array}{rcl} C[\cdot] & = & [\cdot]\, N_0, \text{ where} \\ N_0 & = & \lambda f.\ f\ 1\ N_1, \text{ where} \\ N_1 & = & \lambda g.\ g\ (\lambda x.\ \lambda y.\ 1)\ N_2, \text{ where} \\ N_2 & = & \lambda h.\ h\ (\lambda x.\ \lambda y.\ \Omega)\ (\lambda x.\ x) \end{array}$$

one can show that $C[\overline{M_2}]$ terminates but $C[\overline{M_1}]$ does not. Here Ω stands for any $\boldsymbol{\lambda}_v$-divergent term. ■

Note that this counterexample is robust: the terms M_1 and M_2 are observationally congruent in *untyped* contexts, but their transforms are distinguishable using a fairly simple typed context.

4 Retraction Approach and Its Limitations

In the setting of denotational semantics, the relation between direct and continuation-style semantics has been studied extensively in [15,17,22]. These papers have focused on specific denotational models and use the method of *inclusive predicates* to connect the direct with the continuation meaning of a term. Yet as noted in [8], the significance of these predicates on higher-order functions is not obvious. A simpler approach, using retractions in a typed setting, was developed in [8], using the pure, typed lambda calculus with call-by-name ($\boldsymbol{\lambda\eta}$) reasoning [1].

Definition 7 (Informal) *A* retraction pair *(i, j) is a pair of functions such that for any x, $j\ (i\ x) = x$.*

Theorem 8 (Meyer, Wand) *For any type α, there exist $\boldsymbol{\lambda\eta}$-definable retraction pairs (i_α, j_α) and (I_α, J_α), where $i_\alpha : \alpha \rightarrow \alpha'$, $j_\alpha : \alpha' \rightarrow \alpha$, $I_\alpha : \alpha' \rightarrow ((\alpha' \rightarrow o) \rightarrow o)$, and $J_\alpha : ((\alpha' \rightarrow o) \rightarrow o) \rightarrow \alpha'$. Moreover,*

$$\boldsymbol{\lambda\eta} \vdash M = j_\alpha(J_\alpha\ \overline{M})$$

for any closed, pure term M.

In other words, the direct meaning of a pure term may be recovered from the continuized term by applying the retractions.

Theorem 8 is somewhat misleading. In the pure, simply typed calculus, call-by-name and call-by-value convertibility coincide since no term causes a divergent computation [1]. Call-by-name and call-by-value will yield different observational congruence theories once we add nonterminating computations to the language; we will need to keep this in mind if we wish to extend the retraction approach to $\boldsymbol{\lambda}_v$.

It is worth remarking that even in this simplified setting of pure terms, we cannot expect to have $\overline{M} = i(M)$ for *any* i. This follows because there are two pure, closed terms M, N where $\boldsymbol{\lambda\eta} \vdash M = N$ but $\overline{M}$ and $\overline{N}$ $\boldsymbol{\lambda\eta}$-convert to distinct normal forms, namely the terms

$$\begin{aligned} M &= \lambda a.\lambda b.\lambda c.\ (\lambda z.a)\ (b\ c), \\ N &= \lambda a.\lambda b.\lambda c.\ a. \end{aligned}$$

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 [18], is equationally inconsistent.

5 Retraction Approach in Our Language

We now consider applying the retraction approach to $\boldsymbol{\lambda}_v$. The obvious conjectured generalization of Meyer-Wand would be to exhibit a definable retraction pair (i, j) such that $M \equiv_v j(\overline{M})$. (As in the previous section, we cannot expect $i(M) \equiv_v \overline{M}$ by Theorem 5.) Because call-by-*name* reasoning is not sound for the call-by-*value* calculus as soon as divergent terms are definable, the particular definable retraction pairs discovered by Meyer-Wand no longer serve. Specifically, one can only prove

Definition 9 *M observationally approximates N, written $M \precsim_v N$, if, for any context $C[\cdot]$ such that $C[M]$ and $C[N]$ are complete programs,*

$$\boldsymbol{\lambda}_v \vdash C[M] = c \textit{ implies } \boldsymbol{\lambda}_v \vdash C[N] = c.$$

for all numerals c.

Theorem 10 *For any closed pure term M, $M \precsim_v j_\alpha(J_\alpha\ \overline{M})$.*

Note that Theorem 10 does not hold if we reverse the $\precsim_v$:

Theorem 11 *There is a type α and a typing of the term*

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

at type α such that $j_\alpha(J_\alpha\ \overline{S}) \not\precsim_v S$.

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

6 Conclusion

Continuations fundamentally alter the theory of a language. General principles useful for reasoning about continuations and observational congruences remain to be developed.

7 Acknowledgments

We thank Irene Greif, Lalita Jategaonkar, and Mark Reinhold for their comments on drafts of this paper, and Bard Bloom for comments and help with the proof of Theorem 5.

A Call-by-value Language

A.1 Syntax and Operational Rules

Our language closely resembles Plotkin's metalanguage [12], although we restrict the language to simple types (no recursive types, etc.) The syntax is

$x, y, \ldots$	— variables
$f, g, \ldots$	— μ-bound variables
$0, 1, \ldots$	— integer constants
$\mathit{cond}\ M\ N\ Q$	— conditionals
$\mathit{succ}, \mathit{pred}$	— functional constants
$\lambda x.M$	— abstractions
$M N$	— applications
$\mu f.M$	— recursive definitions

(For technical reasons, we distinguish variables bound by μ's and those not; Plotkin also makes this restriction in [12].) We define a *value* to be a variable that cannot be μ-bound, abstraction, or constant.

The operational semantics of the language is given by the axioms

$$\begin{array}{rcl}
(\lambda x.M)N & \rightarrow & [N/x]\,M, \\
 & & N \text{ a value} \\
\mathit{cond}\,0\ M\ N & \rightarrow & M \\
\mathit{cond}\,(m+1)\ M\ N & \rightarrow & N \\
\mathit{succ}\ m & \rightarrow & (m+1) \\
\mathit{pred}\,0 & \rightarrow & 0 \\
\mathit{pred}\,(m+1) & \rightarrow & m \\
(\mu f.M) & \rightarrow & [\mu f.M/f]\,M
\end{array}$$

and the rules

$$\frac{M \rightarrow M'}{\mathit{cond}\ M\ N\ Q \rightarrow \mathit{cond}\ M'\ N\ Q}$$

$$\frac{M \rightarrow M'}{\mathit{succ}\ M \rightarrow \mathit{succ}\ M'}$$

$$\frac{M \rightarrow M'}{\mathit{pred}\ M \rightarrow \mathit{pred}\ M'}$$

$$\frac{M \rightarrow M'}{M\ N \rightarrow M'\ N}$$

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

Note that this system defines a partial function, Eval_v, that defines the interpreter. It also defines the equational logic of $\boldsymbol{\lambda}_v$ if one replaces all $\rightarrow$'s by $=$'s and one adds the congruence rule

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

The operational semantics clarifies why the language has two distinct sets of variables: μ-bound variables can become non-values during a computation, but λ-bound variables remain values even when terms are substituted in for them.

A.2 Continuation Transform

The transform is the familiar call-by-value transform, incorporating means for continuizing the constants, μ-expressions, and *cond*. It is a syntactic function on terms defined recursively:

$$\begin{array}{rcl}
\overline{x} & = & \lambda\kappa.\kappa\ x \\
\overline{f} & = & \lambda\kappa.f\ \kappa \\
\overline{m} & = & \lambda\kappa.\kappa\ m \\
\overline{pred} & = & \lambda\kappa.\kappa\ (\lambda x.\lambda\kappa_1.\kappa_1\ (pred\ x)) \\
\overline{succ} & = & \lambda\kappa.\kappa\ (\lambda x.\lambda\kappa_1.\kappa_1\ (succ\ x)) \\
\overline{\lambda x.M} & = & \lambda\kappa.\kappa\ (\lambda x.\overline{M}) \\
\overline{MN} & = & \lambda\kappa.\overline{M}\ (\lambda m.\overline{N}\ (\lambda n.m\ n\ \kappa)) \\
\overline{\mu f.M} & = & \lambda\kappa.(\mu f.\overline{M})\ \kappa \\
\overline{cond\ M\ N\ Q} & = & \lambda\kappa.\ \overline{M} \\
 & & (\lambda m.cond\ m\ (\overline{N}\ \kappa)\ (\overline{Q}\ \kappa))
\end{array}$$

References

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

[2] M. Felleisen. Private communication, April 22, 1988.

[3] M. Felleisen. The theory and practice of first-class prompts. In 15th *Symp. Principles of Programming Languages*, pages 180–190, ACM, 1988.

[4] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. Reasoning with continuations. In *Symp. on Logic in Computer Science*, pages 131–141, IEEE, 1986.

[5] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[6] M. J. Fischer. Lambda calculus schemata. In *Proc. of An ACM Conference on Proving Assertions About Programs*, pages 104–109, Las Cruces, NM, 1972.

[7] A. R. Meyer. Semantical paradigms. In *Symp. on Logic and Computer Science*, IEEE, 1988. Appendices by Stavros Cosmadakis. To appear.

[8] A. R. Meyer and M. Wand. Continuation semantics in typed lambda-calculi (summary). In R. Parikh, editor, *Logics of Programs*, pages 219–224, Volume 193 of *Lecture Notes in Computer Science*, Springer-Verlag, 1985.

[9] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sci.*, 17:348–375, 1978.

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

[11] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–256, 1977.

[12] G. D. Plotkin. Types and partial functions. 1984. Unpublished lecture notes.

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

[14] J. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. of the ACM National Meeting (Boston, 1972)*, pages 717–740, 1972.

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

[16] D. A. Schmidt. *Denotational Semantics, A Methodology for Language Development.* Allyn and Bacon, 1986.

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

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

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

[20] G. L. Steele. *Common Lisp: The Language.* Digital Press, Bedford, MA, 1984.

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

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

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

–Cambridge, MA
April 26, 1988