Retraction Approach to CPS Transform*

Jakov Kučan
Department of Mathematics
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
(kucan@theory.lcs.mit.edu)

Abstract

We study the continuation passing style (CPS) transform and its generalization, the *computational transform*, in which the notion of computation is generalized from continuation passing to an arbitrary one. To establish a relation between *direct style* and *continuation passing style* interpretation of sequential call-by-value programs, we prove the Retraction Theorem which says that a lambda term can be recovered from its continuationized form via a λ-definable retraction. The Retraction Theorem is proved in the logic of computational lambda calculus for the simply typable terms.

1 Introduction

The notions of a *continuation* and a *continuation passing style* (CPS) transform have been introduced by a number of authors (see [Rey93] for a historical overview). The main motivation for the independent developments of these concepts seemed to be twofold: explaining the behavior of imperative features in functional languages, and compilation of programs with higher order procedures. Further research led to development of CPS denotational semantics [SW74] (see also [Sto77]), and later categorical semantics of computations [Mog89], as well as compilers based on the CPS transform [Ste78] (see also [App92]). In both kinds of applications one of the central goals of

*This research was supported in part by: "Types and Algorithms", Office of Naval Research (N00014-93-1-1015), "Computational efficiency of optimal reduction in lambda calculus", National Science Foundation (CDA-9504288), and "Logic, Complexity, and Programming Languages", National Science Foundation (CCR-9216185).

the research has been to establish a relationship between original terms and their images under the transform.

In this work we view the CPS transform as a formalization of the (continuation passing style) denotational semantics of a call-by-value programming language in the $\beta\eta$ lambda calculus λ_n.

To model call-by-value evaluation in a programming language we choose Moggi's [Mog88] computational lambda calculus, λ_c, for the two reasons: 1) the logic of λ_c is sound for call-by-value reasoning, and 2) the logic of λ_c is complete for the class of models (computational lambda models [Mog88]) in which most commonly used computational effects can be expressed.

One way of asserting the correctness of the CPS transform, as an interpretation of λ_c in λ_n, is the equational correspondence result due to Sabry and Felleisen [SF92].

Theorem 1.1 (Sabry-Felleisen). For any two lambda terms M and N,

$$\lambda_c \vdash M = N \text{ iff } \lambda_n \vdash \overline{M} = \overline{N}.$$

The left-to-right implication in this theorem says that the CPS transform preserves equality, and the right-to-left implication says that the transform also preserves distinctions. Thus, the transform gives an accurate picture of λ_c-equivalence in λ_n.

To formalize the problem we are trying to solve we observe that the left-to-right implication of the theorem also says that the CPS transform defines a function, $\mathcal{T}$, mapping the λ_c-equivalence classes of lambda terms to the λ_n-equivalence classes of lambda terms. The right-to-left implication can be understood as saying that $\mathcal{T}$ is injective, and therefore has a left inverse, $\mathcal{T}^{-1}$. We ask the question whether functions $\mathcal{T}$, or its inverse $\mathcal{T}^{-1}$, are definable. More precisely,

- is there a lambda term P, such that $\lambda_n \vdash (P\ M) = \overline{M}$, and
- is there a lambda term R, such that $\lambda_c \vdash (R\ \overline{M}) = M$?

An elementary argument given in [MR] shows that the answer to the first question must be "no". In this work we give an affirmative answer to the second question. More precisely we prove the following theorem:

Theorem 1.2 (Retraction, for the CPS transform). For any simple type σ there is a lambda term R_σ, such that for all closed lambda terms M of type σ,

$$\lambda_c \vdash (R_\sigma\ \overline{M}) = M.$$

A version of the above Retraction Theorem was proven by Meyer and Wand [MW85], where the conclusion of the theorem holds in the logic of λ_n. However, as the authors themselves have pointed out to us, their result is misleading. We are interested in behavior of a term M under a call-by-value evaluation, and λ_n is not sound for call-by-value reasoning in presence of any computational effects.

An interesting point about the CPS transform, viewed as an interpretation of a call-by-value programs, is that not only it can interpret pure functional programs, but can also be extended to interpret programs with control operators such as `call/cc` and `abort`. Different extensions of a functional language with "impure" features can also be given denotational semantics using a similar transform. For example, an interpretation of programs in a language with mutable store can be given using the *state passing style* (SPS) transform. As shown by Moggi [Mog88], a number of such *computational effects* can be described by the notion of a monad, and the CPS and the SPS transforms can be generalized to, what we call, the *computational transform* [Wad90, SW96].

The equational correspondence for the computational transform holds as well [SW96], and it is natural to ask whether the Retraction Theorem (Theorem 1.2) generalizes. However, the computational transform maps lambda terms to the terms of the "monadic metalanguage", λ_{ml} [Mog91]. The language of λ_{ml} is extended with new constructs that the logic of the computational lambda calculus has no axioms for, so the question of whether there is a lambda term, R, (even in the language of λ_{ml}) such that

$$\lambda_c \vdash (R\ M^\circ) = M,$$

where M° denotes the computational transform of M, is ill formed. In order to study the Retraction Theorem in an abstract setting that can be applied to other transforms, as well as to the CPS transform, we define a *modified computational transform*, $\mathcal{T}_\diamond$, mapping lambda terms to lambda terms extended with two constants $\mathcal{E}$ and $\mathcal{R}$ that satisfy axiom

$$(\mathcal{R}\ (\mathcal{E}\ M)) = M. \tag{r-e}$$

The modified computational transform satisfies the equational correspondence result for the closed terms, and we prove the Retraction Theorem in the logic of λ_c extended with the axiom (r-e).

Theorem 1.3 (Retraction, for $\mathcal{T}_\diamond$). For any simple type σ there is a term R_σ, such that for all closed lambda terms M of type σ,

$$\lambda_c\text{+(r-e)} \vdash (R_\sigma\ M^\diamond) = M,$$

where $M^\diamond$ stands for the modified computational transform of M.

The proof of the above theorem consists of defining interpretations of types and terms, as well as an ε-relation between the interpretations, and proving that if a term M has a type σ, then the meaning of M and the meaning of σ are related by the ε-relation. This framework is in many ways similar to, and was inspired by, the type inference models developed in [Mit88].

Even though the transforms of interests, namely the CPS transform and the SPS transform, are not special cases of the modified computational transform, we benefit from studying the abstract transform in that we obtain a proof that does not depend on details of a particular transform, and can be applied, by modifying definitions appropriately, to either the CPS or the SPS transform. Hence, we obtain the Retraction Theorems for the CPS and the SPS transforms.

The fore mentioned results apply only to simply typed closed terms. To extend the applicability of these results we can proceed in two directions: we can extend the computational lambda calculus with new language constructs and axioms that define equational behavior of the new terms, and we can extend the type system so that our results apply to a larger class of terms.

Our results can be easily extended to a calculus, extending λ_c, with a datatype such as natural numbers and primitive operators on natural numbers. A more important class of extensions consists of those extensions which introduce a computational effect to λ_c. We have been able to extend the Retraction Theorem to λ_c extended with a divergent element. However, we stop short of proving the Retraction Theorem for λ_c terms extended with recursion.

In an attempt to prove the Retraction Theorem for all (untyped) closed terms, we extend the type system with recursive types and prove the following result. Call a term F a *total* function if for any value V, $F(V)$ is λ_c-equivalent to a value.

Theorem 1.4. Assume terms e and r exist such that e and r are total functions satisfying (in λ_c+(r-e))

$$\begin{array}{rcl} (e\ y) & = & \lambda x.(\mathcal{E}\ (e\ (y\ (r\ x)))) \\ (r\ y) & = & \lambda x.(r\ (\mathcal{R}\ (y\ (e\ x)))) \\ (r\ (e\ x)) & = & x. \end{array}$$

Then for every term M,

$$\lambda_c\text{+(r-e)} \vdash (r\ (\mathcal{R}\ M^\diamond)) = M.$$

Analogous theorems also hold for the CPS and the SPS transforms. The assumption of the above theorem is quite strong, and it remains to be seen whether such terms e and r exist. One should also investigate whether such elements exist in models that would allow interesting applications of the theorem.

2 Definitions

We assume the reader is familiar with elementary concepts of a lambda calculus. For details one is referred to [Bar84]. In this section we will provide concise definitions in order to disambiguate our notation.

2.1 Lambda calculus

Lambda terms are terms formed over an infinite set of variables by lambda abstraction and application. We will use a number of standard conventions when writing lambda terms, such as that application associates to the left, and in general, use parentheses freely to make terms easier to read. We will write (`let` x=M `in` N) to abbreviate the term $((\lambda x.N)\ M)$, and $M \circ N$ for the term $\lambda x.(M\ (N\ x))$. For most part (but not exclusively) we use letters M, N, P, etc. to range over arbitrary lambda terms, and letters U, V and W to range over *values*, that is lambda terms that are either variables, lambda abstractions or constants. Lower-case letters, x, y, z etc. will be used for variables.

We study provable equality between untyped lambda terms. If λ is a set of axions we write $\lambda \vdash M = N$ if the equation $M = N$ can be derived using the rules of lambda congruence from the axioms in λ.

$$((\lambda x.M)\ V) = M\{V/x\} \qquad (\beta_v)$$

$$\lambda x.Vx = V \quad , x \notin \mathrm{FV}(V) \qquad (\eta_v)$$

$$(\texttt{let}\ x\texttt{=}M\ \texttt{in}\ x) = M \qquad (\mathrm{unit})$$

$$\begin{array}{l}(\texttt{let}\ x_2\texttt{=}(\texttt{let}\ x_1\texttt{=}M_1\ \texttt{in}\ M_2)\ \texttt{in}\ M) = \\ \qquad (\texttt{let}\ x_1\texttt{=}M_1\ \texttt{in}\ (\texttt{let}\ x_2\texttt{=}M_2\ \texttt{in}\ M))\end{array} \quad , x_1 \notin \mathrm{FV}(M) \qquad (\mathrm{assoc})$$

$$(M\ N) = (\texttt{let}\ x\texttt{=}M\ \texttt{in}\ (x\ N)) \qquad (\mathrm{let.1})$$

$$(V\ M) = (\texttt{let}\ y\texttt{=}M\ \texttt{in}\ (V\ y)) \qquad (\mathrm{let.2})$$

Table 1: Axioms of λ_c.

For most part of this paper we consider equalities provable in Moggi's λ_c (see Table 1), possibly extended with additional axioms for constants. In particular we will use constants $\mathcal{E}$ and $\mathcal{R}$ satisfying axiom

$$(\mathcal{R}\ (\mathcal{E}\ x)) = x. \qquad (\mathrm{r\text{-}e})$$

2.2 Typing system

We consider a type system for assigning types to untyped lambda terms. Simple types are defined over a base type $\imath$, i.e. $\imath$ is a type and $\sigma\to\tau$ is a type whenever σ and τ are types. The type inference system consists of set of rules (given in Table 2) for deriving sequents of the form $\Gamma \triangleright M\colon\sigma$, where $M\colon\sigma$ is called a *typing assertion* and Γ is a *typing hypothesis*, i.e. a set of typing assertions of the form $x_i\colon\sigma_i$, where we always assume no variable x_i occurs more than once in Γ.

$$\Gamma, x\colon\sigma \triangleright x\colon\sigma \qquad \text{(var)}$$

$$\frac{\Gamma \triangleright M\colon\sigma\to\tau, \quad \Gamma \triangleright N\colon\sigma}{\Gamma \triangleright (M\ N)\colon\tau} \qquad \text{(app)}$$

$$\frac{\Gamma, x\colon\sigma \triangleright M\colon\tau}{\Gamma \triangleright \lambda x.M\colon\sigma\to\tau} \qquad \text{(abs)}$$

Table 2: Type inference rules for simple types.

2.3 Transforms

In this work we concentrate on three transforms mapping lambda terms to lambda terms. First we study the *modified computational transform*, $\mathcal{T}_\diamond$, mapping pure lambda terms to lambda terms extended with two constants $\mathcal{E}$ and $\mathcal{R}$ (see Table 3). The transform $\mathcal{T}_\diamond$ can be viewed as an abstract transform which captures, for our purposes, important properties of the CPS and the SPS transforms. Namely, in the transform $M^\diamond$ of a term M, the order of evaluation (left-to-right and call-by-value) is made explicit. However, the additional structure that makes, in particular, the CPS transform attractive to compiler designers, is not reflected in the definition of $\mathcal{T}_\diamond$.

$$\begin{aligned} x^\diamond &\stackrel{\text{def}}{=} (\mathcal{E}\ x) \\ (\lambda x.M)^\diamond &\stackrel{\text{def}}{=} (\mathcal{E}\ \lambda x.M^\diamond) \\ (M\ N)^\diamond &\stackrel{\text{def}}{=} (\texttt{let}\ f\texttt{=}(\mathcal{R}\ M^\diamond), x\texttt{=}(\mathcal{R}\ N^\diamond)\ \texttt{in}\ (f\ x)) \end{aligned}$$

Table 3: The modified computational transform, $\mathcal{T}_\diamond$.

The next transform we study is the call-by-value version of the Fischer-Reynolds CPS transform. The definition we use (as well as the overline notation) is taken from [Plo75] (see Table 4).

Analogous to the CPS transform, used to give a denotational semantics of programs with mutable store, instead of control operators, is the state passing style transform (SPS). In the definition, given in Table 5, we used the pairing constructs as abbreviations. Namely, ($\texttt{let}\ \langle x_1, x_2\rangle\texttt{=}M\ \texttt{in}\ N$)

$$
\begin{aligned}
\overline{x} &\stackrel{\mathrm{def}}{=} \lambda k.k\ x \\
\overline{\lambda x.M} &\stackrel{\mathrm{def}}{=} \lambda k.k\ \lambda x.\overline{M} \\
\overline{(M\ N)} &\stackrel{\mathrm{def}}{=} \lambda k.\overline{M}\ (\lambda m.\overline{N}\ (\lambda n.m\ n\ k))
\end{aligned}
$$

Table 4: The CPS transform.

abbreviates the expression

$$(\texttt{let}\ x\texttt{=}M\ \texttt{in}\ (\texttt{let}\ x_1\texttt{=}\pi_1(x)\ \texttt{in}\ (\texttt{let}\ x_2\texttt{=}\pi_2(x)\ \texttt{in}\ N))).$$

$$
\begin{aligned}
x^* &\stackrel{\mathrm{def}}{=} \lambda s.\langle x, s\rangle \\
(\lambda x.M)^* &\stackrel{\mathrm{def}}{=} \lambda s.\langle \lambda x.M^*, s\rangle \\
(M\ N)^* &\stackrel{\mathrm{def}}{=} \lambda s.(\texttt{let}\ \langle x_1, s_1\rangle\texttt{=}(M^*\ s), \\
&\qquad\qquad \langle x_2, s_2\rangle\texttt{=}(N^*\ s_1)\ \texttt{in}\ (x_1\ x_2\ s_2))
\end{aligned}
$$

Table 5: The SPS transform.

Even though we give untyped definitions of the transforms, we believe the transforms should be understood in the context of a typed language. This view is supported by the monadic framework, developed by Moggi [Mog88], in which programs are interpreted as "computations". We leave out the details of Moggi's monadic interpretation, as well as the definitions of the typed transforms, since they are not central to our development, but rather sketch the intuitive picture to help motivate some of our definitions.

We think of a transform mapping terms of type σ (intuitively programs) to terms of type $T(\sigma')$ (intuitively computations), where T is a unary type constructor (that depends on the particular transform), and σ' is defined inductively using T to be $\imath' = \imath$ and $(\tau_1 \rightarrow \tau_2)' = \tau_1' \rightarrow T(\tau_2')$. This can be made precise by defining the typed version of the transform mapping typing sequents to typing sequents.

3 Retraction Theorem

In this section we state and prove the Retraction Theorem for the abstract transform, $\mathcal{T}_\diamond$, as well as, for the CPS and the SPS transforms. While one might not find the transform $\mathcal{T}_\diamond$ interesting in itself, for us it serves the purpose. More precisely, we develop a framework which enables us to prove the Retraction Theorem for the transform $\mathcal{T}_\diamond$, and is free of the details specific to the CPS and the SPS transforms. Nevertheless, the framework can be effortlessly modified to prove the Retraction Theorem for each of these transforms. Thus we believe that focusing on the more abstract transform, $\mathcal{T}_\diamond$, improves the clarity of our presentation.

3.1 Retractions

The Retraction Theorem asserts the definability of the inverse of the CPS transform (as well as other transforms of interest). We now give a construction of the type-indexed family of lambda terms that define the retraction. Note that we give the following definitions using terms $\mathcal{E}$ and $\mathcal{R}$, but the definitions should be understood as parameterized by these terms. That is, the inverse of the modified computational transform will be defined using constants $\mathcal{E}$ and $\mathcal{R}$, and in the definition of the inverse of the CPS transform, these constants will be replaced by the terms $\mathcal{E}^K$ and $\mathcal{R}^K$. Likewise for the SPS transform, terms $\mathcal{E}^S$ and $\mathcal{R}^S$ will be used.

If we think of transforms in the context of a typed language, mapping terms (of type σ) to terms representing computations (of type $T(\sigma')$), intuitively, we can understand the pairs of terms defined below as retraction-embedding pairs between types σ and $T(\sigma')$. One can also formally define a notion of a type σ being a retract of a type τ, and in the sense of such definition, given below, we exhibit terms R_σ and E_σ that form a retraction-embedding pair between types σ and $T(\sigma')$. Moreover, we will show that (suitable versions of) terms R_σ define the required inverses of the transforms we study.

Definition 3.1. A type σ is said to be a *retract* of a type τ if there is a pair of lambda terms $R_{\sigma,\tau}$ and $E_{\sigma,\tau}$ of types $\tau \rightarrow \sigma$ and $\sigma \rightarrow \tau$ respectively, such that $(R_{\sigma,\tau}\ (E_{\sigma,\tau}\ x)) = x$ is provable.

Definition 3.2. For each simple type, σ, define terms e_σ and r_σ inductively on the structure of σ as follows:

$$e_\iota \stackrel{\text{def}}{=} r_\iota \stackrel{\text{def}}{=} \lambda x.x,$$

and at higher types we have

$$\begin{array}{rcl} e_{\tau_1 \rightarrow \tau_2} & \stackrel{\text{def}}{=} & \lambda f.\lambda x.(\mathcal{E}\ (e_{\tau_2}\ (f\ (r_{\tau_1}\ x)))) \\ r_{\tau_1 \rightarrow \tau_2} & \stackrel{\text{def}}{=} & \lambda f.\lambda x.(r_{\tau_2}\ (\mathcal{R}\ (f\ (e_{\tau_1}\ x)))). \end{array}$$

Lemma 3.3. λ_c+(r-e) $\vdash (r_\sigma\ (e_\sigma\ x)) = x$

Proof: Easy by induction on σ. ■

Note: Above lemma holds whenever we replace constants $\mathcal{E}$ and $\mathcal{R}$ with any values $\mathcal{E}'$ and $\mathcal{R}'$ such that $\lambda_c \vdash (\mathcal{R}'\ (\mathcal{E}'\ x)) = x$.

Finally we define terms $R_\sigma = \lambda x.(r_\sigma\ (\mathcal{R}\ x))$ and $E_\sigma = \lambda x.(\mathcal{E}\ (e_\sigma\ x))$. Later we will show that R_σ is an inverse of the transform $\mathcal{T}_\diamond$. To define an inverse of the CPS transform we first define terms $\mathcal{R}^K \stackrel{\text{def}}{=} \lambda x.(x\ \lambda z.z)$ and $\mathcal{E}^K \stackrel{\text{def}}{=} \lambda xk.(k\ x)$. It is easy to show (in λ_c) that $(\mathcal{R}^K\ (\mathcal{E}^K\ x)) = x$. We define the retraction-embedding pair (R^K_σ, E^K_σ) using terms $\mathcal{R}^K$ and $\mathcal{E}^K$ instead of constants $\mathcal{R}$ and $\mathcal{E}$ in the above definitions. Similarly for the SPS transform we define terms $\mathcal{R}^S \stackrel{\text{def}}{=} \lambda x.(\pi_1\ (x\ \mathtt{init}))$ and $\mathcal{E}^S \stackrel{\text{def}}{=} \lambda xs.\langle x, s\rangle$, where `init` represents some initial state of the store, and define terms R^S_σ and E^S_σ using these terms instead.

3.2 Interpretations of Terms and Types

The framework we develop to prove the Retraction Theorem is closely related to Mitchell's type inference models [Mit88]. As in the definition of a type inference model we define interpretations of terms and types, as well as a relation between the two. However, the definition of a type inference model assumes one works with the full $\beta\eta$-equality, and we need to relax the definitions to accommodate reasoning in the weaker logic of λ_c.

We will first sketch the definitions for the general framework and then fill in the details that apply to each particular transform we study.

Interpretation of terms: Assume $(\mathcal{D}, \text{App})$ is an applicative structure, that is, $\mathcal{D}$ is a set and App is a binary operation on $\mathcal{D}$. Assume also that there is a distinguished subset, $\mathcal{V}(\mathcal{D}) \subseteq \mathcal{D}$, of *values.* Given an environment, ρ, mapping variables to $\mathcal{V}(\mathcal{D})$ we define an interpretation (relative to ρ) that to each term M assigns an element $[\![M]\!]\rho$ of $\mathcal{D}$. In addition we assume that

$$[\![(M\ \ N)]\!]\rho = \text{App}([\![M]\!]\rho, [\![N]\!]\rho).$$

In particular we will define $\mathcal{D}$ to be the set of equivalence classes of terms and the set of values to be the equivalence classes of terms that are values. (Note: below, we may be informal and identify terms with their equivalence classes.) The interpretation function will then be defined by the transform in consideration.

Interpretation of types: The types will be interpreted as certain subsets of $\mathcal{D}$ called *type sets.*

In particular we will chose type sets to contain only the equivalence classes of terms of the form $(\mathcal{E}\ V)$ for some value V. The interpretation of types will be defined (using the ε-relation defined below) inductively:

$$\llbracket \imath \rrbracket \stackrel{\text{def}}{=} \text{set of all terms of the form } (\mathcal{E}\ V)$$

$$\llbracket \sigma \to \tau \rrbracket \stackrel{\text{def}}{=} \text{set of all } M \text{ (of the form } (\mathcal{E}\ V)) \text{ such that } \forall N \in \llbracket \sigma \rrbracket\text{:}$$

- $\textsc{App}(M, N)\ \varepsilon\ \llbracket \tau \rrbracket$
- $(R_\tau\ \textsc{App}(M, N)) = ((R_{\sigma\to\tau}\ M)\ (R_\sigma\ N))$

Relating the interpretations: To relate meanings of terms and types, instead of using the simple set-membership relation, we define an *extended membership* relation $\varepsilon \supseteq \in$. This relation will in general depend on the structure of the transform in consideration, and intuitively, will serve the purpose of separating out "important" part of the transformed term.

Truth and validity: Having defined the above notions, we say that a typing assertion $M\!:\sigma$ is *true* (with respect to ρ), written $\rho \models M\!:\sigma$, if $\llbracket M \rrbracket \rho\ \varepsilon\ \llbracket \sigma \rrbracket$. The notions of satisfaction and validity are defined in the standard way relative to the definition of truth. Namely, ρ *satisfies* a typing hypothesis Γ, written $\rho \models \Gamma$, if every typing assertion in Γ is true with respect to ρ, and a typing sequent $\Gamma \rhd M\!:\sigma$ is *valid*, written $\Gamma \models M\!:\sigma$, if $M\!:\sigma$ is true for every ρ that satisfies Γ.

Our aim is to prove a soundness lemma for the type inference system that would imply the Retraction Theorem. Before we state the lemma, we single out two conditions that are necessary for the lemma to hold. Namely we require that for all $M\ \varepsilon\ \llbracket \sigma \to \tau \rrbracket$ and $N\ \varepsilon\ \llbracket \sigma \rrbracket$:

$$\textsc{App}(M, N)\ \varepsilon\ \llbracket \tau \rrbracket \tag{$\dagger$}$$

$$(R_\tau\ \textsc{App}(M, N)) = ((R_{\sigma\to\tau}\ M)\ (R_\sigma\ N)), \tag{$\ddagger$}$$

where R_σ, etc. are defined in Section 3.1 and equality is provable equality in λ_c+(r-e). The main reason why these two conditions are singled out is that, having proved the Retraction Theorem for the modified computational transform, we modify the definition of the interpretations of terms and types, as well as the ε-relation, to reflect properties of the CPS transform, and only the conditions (†) and (‡) need to be proved again to show that the Retraction Theorem holds for the CPS transform. Similarly for the SPS transform.

3.3 Modified Computational Transform

In the previous section we outlined the definitions of our framework, and we can now fill in the details. The following definitions are given, in particular, to prove the Retraction Theorem for the modified computational transform, $\mathcal{T}_\diamond$, but we will also indicate how these definitions need to be changed in the subsequent sections to prove the Retraction Theorem for the CPS and the SPS transforms.

We write "=" to denote an equality provable in λ_c+(r-e).

Definition 3.4. Let ρ be a substitution mapping variables to values. Define the *interpretation* function $[\![\cdot]\!]$ to be

$$[\![M]\!]\rho \stackrel{\text{def}}{=} \rho(M^\diamond).$$

Definition 3.5. *Type sets* are sets of terms of the form $(\mathcal{E}\ V)$, where V is a value.

Note: this definition is to be understood as parameterized by the term $\mathcal{E}$, that is, when we consider, say, the CPS transform we will use the term $\mathcal{E}^K$ instead.

Definition 3.6. If M and N are terms, we define the application in the codomain of the transform as follows:

$$\text{App}(M, N) \stackrel{\text{def}}{=} (\texttt{let}\ f\texttt{=}(\mathcal{R}\ M), x\texttt{=}(\mathcal{R}\ N)\ \texttt{in}\ (f\ x)).$$

It is easy to see that $[\![(M\ N)]\!]\rho = \text{App}([\![M]\!]\rho, [\![N]\!]\rho)$.

Definition 3.7. Let S be a type set and let M be a term. We write $M\ \varepsilon\ S$ if there is a value V and terms $P_1, \ldots, P_k$ such that

$$M = (\texttt{let}\ x_1\texttt{=}P_1, \ldots, x_k\texttt{=}P_k\ \texttt{in}\ (\mathcal{E}\ V)),$$

and $(\mathcal{E}\ V) \in S$.

Definition 3.8. The *interpretation* of types is defined inductively on the structure of type expressions. Namely,

$$\begin{array}{rcl} [\![\imath]\!] & \stackrel{\text{def}}{=} & \text{set of all terms of the form } (\mathcal{E}\ V) \\ [\![\sigma\to\tau]\!] & \stackrel{\text{def}}{=} & \text{set of all } M\text{, of the form } (\mathcal{E}\ V)\text{, such that } \forall N \in [\![\sigma]\!], \\ & & \bullet\ \ \textsc{App}(M,N)\ \varepsilon\ [\![\tau]\!] \\ & & \bullet\ \ (R_\tau\ \textsc{App}(M,N)) = ((R_{\sigma\to\tau}\ M)\ (R_\sigma\ N)) \end{array}$$

Recall that $R_\sigma = r_\sigma \circ \mathcal{R}$.

Note: We should understand this definition as parameterized by $\mathcal{E}$, $\mathcal{R}$, $\textsc{App}$, and ε. When we consider the CPS or the SPS transform, the appropriate definitions will be used instead.

Equipped with the definitions we can prove the Soundness Lemma, but first we need some auxiliary results.

Lemma 3.9. For each type σ and any value V, $(\mathcal{E}\ (e_\sigma\ V)) \in [\![\sigma]\!]$.

Proof: Easy by induction on σ. ■

Lemma 3.10. For any two terms M and N such that $M\ \varepsilon\ [\![\sigma\to\tau]\!]$ and $N\ \varepsilon\ [\![\sigma]\!]$,

$$\textsc{App}(M,N)\ \varepsilon\ [\![\tau]\!],$$

and

$$(R_\tau\ \textsc{App}(M,N)) = ((R_{\sigma\to\tau}\ M)\ (R_\sigma\ N)).$$

Proof: First observe that the statement of this lemma is stronger than what is given in Definition 3.8 above. The definition only requires that the two statements hold only for $N \in [\![\sigma]\!]$, and in the lemma we show that these two statements hold for all $N\ \varepsilon\ [\![\sigma]\!]$. Both conditions can be easily proved using the definitions and axioms of λ_c+(r-e). ■

The Soundness Lemma consists of two parts, (S.1) and (S.2). The first part asserts the soundness of our interpretation with respect to typing rules of the simply typed lambda calculus, and the second part is, in fact, the statement of the Retraction Theorem.

Lemma 3.11 (Soundness). Let Γ be a typing hypothesis and let ρ be a substitution that satisfies Γ. Let ρ' be a substitution such that for each $x_i\colon\sigma_i \in \Gamma$, $\rho'(x_i) = (r_{\sigma_i}\ x_i)$. Then, if $\Gamma \triangleright M\colon\sigma$,

$$\Gamma \models M\!:\sigma \tag{S.1}$$

and

$$\lambda_c\text{+(r-e)} \vdash (R_\sigma\ \rho(M^\diamond)) = \rho'(M) \tag{S.2}$$

Proof: We prove the lemma by induction on the derivation of $\Gamma \triangleright M\!:\sigma$. The (var) case follows by assumptions, and the (app) case follows directly by Lemma 3.10 and induction hypotheses.

The (abs) case is slightly more involved. Assume $M \equiv \lambda x.N$ and $\sigma \equiv \tau_1 \rightarrow \tau_2$, and that $\Gamma \triangleright \lambda x.N\!:\tau_1 \rightarrow \tau_2$ was derived from $\Gamma, x\!:\tau_1 \triangleright N\!:\tau_2$ using the (abs) rule.

To show (S.1), first observe that $\rho((\lambda x.N)^\diamond) = (\mathcal{E}\ \rho(\lambda x.N^\diamond))$, so we need to show that $\rho((\lambda x.N)^\diamond) \in [\![\tau_1 \rightarrow \tau_2]\!]$. Let $(\mathcal{E}\ V)$ be any term in $[\![\tau_1]\!]$. Then

$$\textsc{App}(\rho((\lambda x.N)^\diamond), (\mathcal{E}\ V)) = \rho\{V/x\}(N^\diamond),$$

and therefore since $\rho\{V/x\}$ satisfies $\Gamma, x\!:\tau_1$, by induction hypothesis (S.1), $\textsc{App}(\rho((\lambda x.N)^\diamond), (\mathcal{E}\ V))\ \varepsilon\ [\![\tau_2]\!]$. Moreover we can compute

$$(R_{\tau_2}\ \textsc{App}(\rho((\lambda x.N)^\diamond), (\mathcal{E}\ V))) = ((R_{\tau_1 \rightarrow \tau_2}\ \rho((\lambda x.N)^\diamond))\ (R_{\tau_1}\ (\mathcal{E}\ V))),$$

using the definition of the retraction-embedding pairs, and both induction hypotheses (S.1) and (S.2). This establishes (S.1). To show (S.2) simply compute

$$(R_{\tau_1 \rightarrow \tau_2}\ \rho((\lambda x.N)^\diamond)) = \rho'(\lambda x.N)$$

using properties of retraction-embedding pairs, Lemma 3.9 and induction hypothesis (S.2). ■

The Retraction Theorem follows directly from this lemma.

Theorem 3.12 (Retraction, for $\mathcal{T}_\diamond$). For any closed term M of (simple) type σ,

$$\lambda_c\text{+(r-e)} \vdash (R_\sigma\ M^\diamond) = M.$$

3.4 The CPS Transform

To prove the the Retraction Theorem for the CPS transform, as indicated before, we will use exactly the same framework, but will modify the definitions using appropriate definitions of application in the codomain of the transform and ε-relation. First recall that we define the retraction-embedding pair $(\mathcal{R}^K, \mathcal{E}^K)$ to be

$$\mathcal{E}^K \stackrel{\text{def}}{=} \lambda x k.(k\ x) \quad \text{and} \quad \mathcal{R}^K \stackrel{\text{def}}{=} \lambda x.(x\ (\lambda z.z)).$$

It is easy to see that $\lambda_c \vdash (\mathcal{R}^K\ (\mathcal{E}^K\ x)) = x$, which is the only abstract property of $\mathcal{E}$ and $\mathcal{R}$ we use.

We define the interpretation of terms (relative to a substitution ρ) using the CPS transform, that is,

$$[\![M]\!]\rho \stackrel{\text{def}}{=} \rho(\overline{M}).$$

The application in the codomain of the transform is defined to be

$$\text{App}^K(M, N) \stackrel{\text{def}}{=} \lambda k.M\ (\lambda m.N\ (\lambda n.m\ n\ k)),$$

so that $\overline{(M\ N)} = \text{App}^K(\overline{M}, \overline{N})$. Finally, we define the extended membership relation, ε^K, to be

$$M\ \varepsilon^K\ S \quad \text{iff} \quad M = \lambda k.((\texttt{let}\ x_1\texttt{=}P_1, \ldots, x_l\texttt{=}P_l\ \texttt{in}\ (\mathcal{E}^K\ V))\ k),$$

for some terms P_i, a value V such that $(\mathcal{E}^K\ V) \in S$, and a fresh variable k.

It should be understood that all the definitions used in the preceding section are now defined using $\mathcal{E}^K$, $\mathcal{R}^K$, App^K and ε^K, instead of $\mathcal{E}$, $\mathcal{R}$, App and ε.

With the new definitions, we prove the following lemma, analogous to Lemma 3.10, asserting that conditions (†) and (‡) hold.

Lemma 3.13. For any two terms M and N such that $M\ \varepsilon^K\ [\![\sigma \to \tau]\!]$ and $N\ \varepsilon^K\ [\![\sigma]\!]$,

$$\text{App}^K(M, N)\ \varepsilon^K\ [\![\tau]\!],$$

and

$$(R^K_\tau\ \text{App}^K(M, N)) = ((R^K_{\sigma\to\tau}\ M)\ (R^K_\sigma\ N)).$$

Proof: The proof is straightforward using the definitions and axioms of λ_c. ■

Having shown the above lemma, the rest of the proof of the Soundness Lemma for the CPS transform is exactly the same as in the case of the modified computational transform, and as a corollary we obtain the retraction result.

Theorem 3.14 (Retraction, for the CPS transform). For any closed term M of (simple) type σ,

$$\lambda_c \vdash (R^K_\sigma\ \overline{M}) = M.$$

3.5 The SPS Transform

To adopt our framework to the SPS transform we define $\mathcal{E}^S$, $\mathcal{R}^S$, $\textsc{App}^S$ and ε^S in place of $\mathcal{E}$, $\mathcal{R}$, $\textsc{App}$ and ε, and prove that the conditions (†) and (‡) still hold. Recall that the terms $\mathcal{E}^S$ and $\mathcal{R}^S$ are defined to be

$$\mathcal{E}^S \stackrel{\text{def}}{=} \lambda x s.\langle x, s\rangle \quad \text{and} \quad \mathcal{R}^S \stackrel{\text{def}}{=} \lambda x.(\pi_1\ (x\ \texttt{init})),$$

where `init` is some initial state of the store.

The interpretation of terms is defined using the SPS transform. Namely,

$$[\![M]\!]\rho \stackrel{\text{def}}{=} \rho(M^*).$$

The application in the codomain of the transform is define to be

$$\textsc{App}^S(M, N) \stackrel{\text{def}}{=} \lambda s.(\texttt{let}\ \langle x_1, s_1\rangle\texttt{=}(M^*\ s), \langle x_2, s_2\rangle\texttt{=}(N\ s_1)\ \texttt{in}\ (x_1\ x_2\ s_2)),$$

so that $(M\ N)^* = \textsc{App}^S(M^*, N^*)$. Finally we define the extended membership relation, ε^S, as follows:

$$M\ \varepsilon^S\ S \quad \text{iff} \quad M = \lambda s.(\texttt{let}\ x_1\texttt{=}P_1, \ldots x_l\texttt{=}P_l\ \texttt{in}\ \langle V, s\rangle),$$

for some terms P_i, some value V such that $(\mathcal{E}^S\ V) \in S$, and fresh s.

Furthermore, we interpret the definitions of type sets, interpretation of types, and retraction-embedding pairs defined earlier, as if given using $\mathcal{E}^S$, $\mathcal{R}^S$, $\textsc{App}^S$ and ε^S, in place of $\mathcal{E}$, $\mathcal{R}$, $\textsc{App}$ and ε. With the new definitions we can show that conditions (†) and (‡) are still satisfied. This yields the Retraction Theorem for the SPS transform.

Theorem 3.15 (Retraction, for the SPS transform). For any closed term M of (simple) type σ,

$$\lambda_c \vdash (R^S_\sigma\ M^*) = M.$$

3.6 Extensions

Thus far we have only proved the retraction results for the pure simply typed terms. In order to make these results more applicable we would like to extend the theorems to a larger class of terms. We have essentially two directions in which we can proceed. We can extend the class of terms by adding constants or term constructors (including possibly new axioms that define functional behavior of the new terms), and secondly, we can extend the type system to one that can type a larger class of terms.

Extending the Retraction Theorem to an extension of λ_c with constants of a base type and primitive operators such as numerals is quite straightforward. However, adding arbitrary constants of higher order types may be more difficult. The difficulty lies in ensuring the closure conditions imposed on type sets by the addition of such constants are satisfied. For example if a constant c of type $\sigma \rightarrow \tau$ is added to λ_c, we need to make sure that if $M \;\varepsilon\; \sigma$ then $\textsc{App}(\mathsf{c}^\diamond, M) \;\varepsilon\; \tau$. While such closure conditions are determined based on the type of new constants, the proof they are satisfied will, in general, depend on the functional behavior of the new constants.

Divergence: The difference between call-by-name and call-by-value evaluation strategies becomes apparent only in presence of actual computational effects. So far we have only considered pure simply typed terms. In this setting every closed term is equivalent to a value in both logics of λ_c and λ_n. Therefore, if we were to stop here, it would be unjustified to claim significant improvement over the original Meyer-Wand Retraction Theorem.

The simplest computational effect we can add to the language is divergence. In presence of divergence λ_n reasoning is no longer sound for call-by-value languages, so for any applications in λ_c extended with divergence, we really need the stronger version of the Retraction Theorem provable in the weaker logic of λ_c. While extension of the Retraction Theorem to a language with divergence, which we now present, is quite straightforward, it is important since it illustrates the difference between Meyer and Wand's and our formulation of the Retraction Theorem.

Divergence is represented by the divergent element, Ω, that is added to the language of λ_c as a constant, but it is not considered a value. The axioms for Ω specify that an application diverges if either the operators or the operand diverges. Moreover, these axioms identify all divergent terms. The axioms are:

$$(\Omega \; M) = \Omega \quad \text{and} \quad (M \; \Omega) = \Omega.$$

One can verify that the resulting equational logic is consistent and that it cannot prove $\Omega = V$ for any value V. The type system is extended with the axiom

$$\Gamma \triangleright \Omega \colon \sigma \qquad (\Omega)$$

which says that Ω has every type. The modified computational transform is defined on Ω to be

$$\Omega^{\diamond} \stackrel{\text{def}}{=} \Omega.$$

Since Ω has every type, to prove the Retraction Theorem for $\lambda_c{+}\Omega$ we need to extend the Soundness Lemma for the case of typing axiom (Ω). In other words we need to show (S.1): that $\Omega^{\diamond} \mathrel{\varepsilon} [\![\sigma]\!]$, and (S.2): that $(R_\sigma\ \Omega^{\diamond}) = \Omega$ for every type σ. The second condition follows trivially from the definition of $\Omega^{\diamond}$ and the axioms for Ω. To prove the first condition observe that

$$\Omega = (\texttt{let}\ x\texttt{=}\Omega\ \texttt{in}\ (\mathcal{E}\ (e_\sigma\ x))),$$

and by Lemma 3.9, $(\mathcal{E}\ (e_\sigma\ x)) \in [\![\sigma]\!]$ for every σ.

The very same reasoning can be applied to extend the Retraction Theorem for the CPS transform to $\lambda_c{+}\Omega$, where the CPS transform is defined on Ω to be $\overline{\Omega} \stackrel{\text{def}}{=} \lambda k.\Omega\ k$.

Recursive types: It is well known that all terms can be typed using the recursive type system. In order to extend the Retraction Theorem to all closed terms we study the recursive types.

The recursive type discipline introduces types of the form $\mu t.\sigma$ (where we use t to denote a type variable). In order to extend our results to λ_c extended with recursive types we need to define retraction-embedding pairs (r_σ, e_σ) at new types. In particular how does one define $e_{\mu t.\sigma}$ and $r_{\mu t.\sigma}$, or even e_t and r_t?

To motivate a solution, consider the following example. Let $\tau = \mu t.t{\rightarrow}t$. Then in the recursive type discipline one can type $\triangleright(\lambda x.x\ x) \colon \tau{\rightarrow}\tau$. Assume we have defined terms e_τ and r_τ, and we try to compute

$$\begin{array}{rcl}
(r_{\tau\rightarrow\tau}\ (\mathcal{R}\ (\lambda x.x\ x)^{\diamond})) & = & (r_{\tau\rightarrow\tau}\ (\mathcal{R}\ (\mathcal{E}\ \lambda x.x\ x))) \\
& = & (r_{\tau\rightarrow\tau}\ \lambda x.x\ x) \\
& = & \lambda x.(r_\tau\ (\mathcal{R}\ ((e_\tau\ x)\ (e_\tau\ x)))) \\
& = & ?
\end{array}$$

To complete this derivation, one would like to have $(e_\tau\ x) = \mathcal{E} \circ e_\tau \circ x \circ r_\tau$, so we can continue

$$\begin{array}{rcl} \cdots & = & \lambda x.(r_\tau\ (\mathcal{R}\ (\mathcal{E}\ (e_\tau\ (x\ (r_\tau\ (e_\tau\ x))))))) \\ & = & \lambda x.x\ x \end{array}$$

What we see from this example is that the two occurrences of x in $\lambda x.x\ x$ "act" as having types $\tau{\rightarrow}\tau$ and τ, respectively. Similarly, we would like the two occurrences of e_τ in $\lambda x.(r_\tau\ (\mathcal{R}\ ((e_\tau\ x)\ (e_\tau\ x))))$ to "act" as $e_{\tau\rightarrow\tau}$ and e_τ. A solution to our problem is to find a uniform definition for e's and r's at all types. Namely, we want a retraction-embedding pair (r, e) that satisfies the following definition.

Definition 3.16. A term F is called a *total function* if F is a value and, for any value V, $(F\ V)$ is provably equal to a value.

A pair of total functions (r, e) is a *uniform* retraction-embedding pair if e and r satisfy system of equations

$$\begin{array}{rcl} (r\ x) & = & r \circ \mathcal{R} \circ x \circ e \\ (e\ x) & = & \mathcal{E} \circ e \circ x \circ r \\ r \circ e & = & \lambda x.x. \end{array} \qquad (\star)$$

While it remains open whether there is a pair of terms satisfying the above conditions, we will assume we are given such a pair of functions and, under this assumption, show how the Retraction Theorem can be extended to recursive types. Moreover, since the recursive type system can type *all* terms, as a corollary we obtain the following theorem.

Theorem 3.17. Assume total functions e and r exist that satisfy equations $(\star)$. Then, for any closed lambda term M,

$$\lambda_c \vdash (r\ (\mathcal{R}\ M^\diamond)) = M.$$

Of course, the analogous theorems hold for the other transforms as well.

Here we only sketch the main idea in the proof of the above theorem. A detailed proof can be found in [Kuč97].

The recursive type system extends the simple types by adding type variables and type expressions of the form $\mu t.\sigma$. The new inference rules are

$$\frac{\Gamma \triangleright M\colon \sigma\{\mu t.\sigma/t\}}{\Gamma \triangleright M\colon \mu t.\sigma} \qquad (\mu\ \mathrm{I})$$

$$\frac{\Gamma \triangleright M\colon \mu t.\sigma}{\Gamma \triangleright M\colon \sigma\{\mu t.\sigma/t\}} \qquad (\mu\ \mathrm{E})$$

One can understand these rules by considering the type $\mu t.\sigma$ as the type τ, satisfying equation $\tau = \sigma\{\tau/t\}$. Thus we need to define the interpretation, $[\![\tau]\!]\nu$, of τ such that it satisfies the equation

$$[\![\tau]\!]\nu = [\![\sigma\{\tau/t\}]\!]\nu.$$

In other words $[\![\tau]\!]\nu$ should be a fixed point of the function

$$\lambda S.[\![\sigma]\!]\nu\{S/t\}.$$

(We assume the interpretation will satisfy $[\![\sigma\{\tau/t\}]\!]\nu = [\![\sigma]\!]\nu\{[\![\tau]\!]\nu/t\}$.) The difficulty lies in showing that for any σ and ν, the function $\lambda S.[\![\sigma]\!]\nu\{S/t\}$ always has a fixed point. To do so, we define a metric on the space of all type sets so that the resulting metric space is complete. Then we show that each function $\lambda S.[\![\sigma]\!]\nu\{S/t\}$ is a contraction, and thus, by Banach's Fixed-point Theorem, has a unique fixed point. Mac Queen et al. [MPS86] have developed such a framework, of which our development can be viewed as a special case. Namely, our domain consists only of finite elements (typing sequents) ordered under discrete order, thus greatly simplifying general purpose structures used in [MPS86].

4 Concluding Remarks

In this work, we have established a relation between direct style and CPS terms using definable retraction functions. The Retraction Theorem shows that a term can be recovered, up to λ_c-equivalence, from its image under the CPS transform. Therefore, the retraction approach, in fact, only provides a relation between equivalence classes of terms. To contrast our results with others that provide, perhaps even stronger relation between lambda terms and their CPS forms (e.g. [SW96]), we should emphasize that the inverse of the CPS transform we obtain is *definable*. Another important point is that the conclusion of our version of the Retraction Theorem is an equation provable in the logic of λ_c, which is a call-by-value logic, unlike the results in [MW85] and [Fil94] which give similar equalities, but in a call-by-name logic. As a consequence, our results are applicable even where call-by-name reasoning is not sound.

Some open questions: In all practical applications, functional programming languages are equipped with some form of recursion. Therefore, to make the retraction approach applicable in practice, we need to extend our results to a language with recursion. This can be done in two ways: By extending the type system so that the fixed-point operator is definable in the pure language, or by adding a language construct such as constant Y, `letrec`, etc. The first approach, with some partial results, has been discussed earlier.

One difficulty in adding fixed-point operator Y, or a similar language construct, is that additional closure conditions are needed in the definition of type sets, and we haven't been able to construct type sets satisfying these conditions. The other difficulty is determining the correct axiomatization of a fixed-point operator. It appears that the axiom

$$(Y\ V) = (V\ (Y\ V)) \tag{fix}$$

does not suffice. In models of λ_c, fixed-point operator can be defined using the so called *fixpoint object*. Crole and Pitts [CP92] define such an object in models of λ_c, and discuss a logical system for reasoning about fixpoint computations, which may hold the answer to above questions.

Another class of extensions is motivated by the application of the Retraction Theorem developed by Riecke and Viswanathan [RV95], where they show how one can isolate effects of an extension of a language with assignment or control from interfering with pure functional code. A natural question arises, whether it is possible to extend this approach to isolate one computational effect from interfering with code possibly containing a different computational effect. For instance, if M is a program in, say call-by-value PCF with assignment, can we define an operator, call it `encap`, so that, in an extension of call-by-value PCF with both assignment *and* control, (`encap` M) will behave the same as M behaves in the extension of call-by-value PCF with assignment. We believe that an appropriate extension of the Retraction Theorem to a programming language with imperative features may give us such results.

References

[App92] Anrew W. Appel. *Compiling with Continuations.* Cambridge Universisty Press, 1992.

[Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic*. North-Holland Publishing Co., Amsterdam, second edition, 1984.

[CP92] Roy L. Crole and Andrew M. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98(2):171–210, jun 1992.

[Fil94] Andrzej Filinski. Representing monads. In *Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994.

[Kuč97] Jakov Kučan. *Metatheorems about Convertibility in Typed Lambda Calculi: Applications to CPS Transform and "Free Theorems"*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, U.S.A., February 1997.

[Mit88] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

[Mog88] Eugenio Moggi. Computational lambda-caluclus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh, October 1988.

[Mog89] Eugenio Moggi. Computational lambda-caluclus and monads. In *LICS*, 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, October/November 1986.

[MR] Albert R. Meyer and Jon G. Riecke. Continuations may be unreasonable. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 63–71.

[MW85] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs: Proceedings*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224. Springer-Verlag, 1985.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[Rey93] John C. Reynolds. The discoveries of continutions. *Lisp and Symbolic Computation*, 6(3/4), November 1993.

[RV95] Jon G. Riecke and Ramesh Viswanathan. Isolating side effects in sequential languages. In *In Proceedings of Symposium on Principles of Programming Languages*, 1995.

[SF92] Amr Sabry and Mathias Felleisen. Reasoning about programs in continuation-passing style. Technical Report COMP TR 92-180, Rice University, 1992.

[Ste78] Guy L. Steele. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, 1978.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[SW74] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Report PRG–11, Oxford University Computing Laboratory, 1974.

[SW96] Amr Sabry and Philip Wadler. A reflection on call-by-value. In *International Conference on Functional Programing*, Philadelphia, May 1996. ACM Press.

[Wad90] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1990.