Reasoning about Programs in Continuation-Passing Style

Amr Sabry Matthias Felleisen [*]

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

Abstract

Plotkin's λ-value calculus is sound but incomplete for reasoning about $\beta\eta$-transformations on programs in continuation-passing style (CPS). To find a complete extension, we define a new, compactifying CPS transformation and an "inverse" mapping, *un*-CPS, both of which are interesting in their own right. Using the new CPS transformation, we can determine the precise language of CPS terms closed under $\beta\eta$-transformations. Using the *un*-CPS transformation, we can derive a set of axioms such that every equation between source programs is provable if and only if $\beta\eta$ can prove the corresponding equation between CPS programs. The extended calculus is equivalent to an untyped variant of Moggi's computational λ-calculus.

1 Compiling with CPS

Many modern compilers for higher-order functional languages [1, 13, 19, 20] utilize some variant of the Fischer-Reynolds continuation-passing style (CPS) transformation [10, 18]. Once a program is in continuation-passing style, these compilers usually perform code optimizations via local transformations. Typical examples of such optimizations are loop unrolling, procedure inlining, and partial evaluation.

In the terminology of the λ-calculus, optimizations generally correspond to (sequences of) β- and η-reductions. Hence, a natural question to ask is whether reductions on CPS programs correspond to known transformations of source programs. If so, optimizations of CPS programs could be understood and reported in terms of the original program as opposed to its rather complicated CPS version. In particular, compilers that do *not* use the CPS transformation, e.g., Chez Scheme [12] or Zinc [15], could benefit by implementing transformations of source programs that correspond to transformations of CPS programs.

Technically speaking, we are addressing the following question: which calculus can prove $M = N$ for by-value expressions M and N, if $cps(M) = cps(N)$ is provable in the (by-value or by-name) λ-calculus? As Plotkin [17] showed in 1974, the λ_v-calculus does not suffice. Thus we refine this question as follows:

Is there a set of axioms, A, that extend the call-by-value λ-calculus such that:

$$\lambda_v A \vdash M = N \text{ iff } \lambda\beta\eta \vdash cps(M) = cps(N).$$

Such a correspondence theorem would be similar to the correspondence theorems for the λ-calculus and combinatory logic [2: ch 7], and the λ_v-calculus and by-value combinatory logic [11]. In analogy to model theory, we call the left-to-right direction *soundness* and the right-to-left direction *completeness* since the CPS transformation is often taken as the definition of a call-by-value semantics.

To derive A, we proceed in three steps:

1. First, we develop a CPS transformation that produces a canonical form of CPS programs. The new transformation also produces the smallest possible output of all known CPS transformations, without reducing any of the original (source) redexes.

2. Second, we develop an *un*-CPS transformation that maps canonical CPS programs and their derivations back to the original language. As Danvy and Lawall [3, 6] convincingly argue, this translation from CPS to *direct* terms is useful in its own right.

3. Finally, by studying the connection between the CPS and *un*-CPS transformations, we systematically derive A. The extended λ-value calculus is equivalent to an untyped variant of Moggi's [16] computational λ-calculus.

*Both authors were supported in part by NSF grant CCR 89-17022.

1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0288...$1.50

The next section introduces the basic terminology and notation of the λ-calculus and its semantics. The third section is a short history of CPS transformations. In Section 4, we formalize the problem and outline our approach to the solution. Section 5 is about the CPS language and its properties. It also contains the definition of our new CPS transformation. Section 6 presents the "inverse" mapping and its exact relation to the CPS transformation. Our main results, the extension of the $\boldsymbol{\lambda}_v$-calculus and its completeness, are the subject of Section 7. Finally, Section 8 discusses the relevance of the new calculus and future directions of research.

Due to space limitations, we only indicate the ideas behind major proof steps. For details, we refer the reader to our extended technical report (*Rice TR 92-180*).

2 Λ: Calculi & Semantics

The language of the *pure* lambda calculus, Λ [2], consists of variables, λ-abstractions, and applications. The set of terms, M, is generated inductively over an infinite set of variables, *Vars*:

$$\begin{array}{rcll} M & ::= & V \mid (M\ M) & \text{(Terms)} \\ V & ::= & x \mid (\lambda x.M) & \text{(Values)} \\ x & \in & \mathit{Vars} & \end{array}$$

A term is either a value, V, or an application. Values consist of variables, drawn from the set *Vars*, and λ-abstractions.

We adopt Barendregt's [2: ch 2, 3] notation and terminology for this syntax. Thus, in the abstraction $(\lambda x.M)$, the variable x is *bound* in M. Variables that are not bound by a λ-abstraction are *free*; the set of free variables in a term M is $FV(M)$. We identify terms modulo bound variables, and we assume that free and bound variables do not interfere in definitions or theorems. In short, we follow common practice and work with the quotient of Λ under α-equivalence. We write $M \equiv N$ for α-equivalent terms M and N.

The expression $M[x := N]$ is the result of the capture-free substitution of all free occurrences of x in M by N. For example, $(\lambda x.xz)[z := (\lambda y.x)] \equiv (\lambda u.u(\lambda y.x))$. A *context*, C, is a term with a "hole", $[\,]$, in the place of one subexpression. The operation of *filling* the context C with an expression M yields the term $C[M]$, possibly capturing some free variables of M in the process. Thus, the result of filling $(\lambda x.x[\,])$ with $(\lambda y.x)$ is $(\lambda x.x(\lambda y.x))$.

Calculi: A λ-calculus is an equational theory over Λ with a finite number of axiom schemas and inference rules. The most familiar axiom schemas are the following notions of reductions:

$$\begin{array}{rcllr} ((\lambda x.M)\ N) & \longrightarrow & M[x := N] & & (\beta) \\ ((\lambda x.M)\ V) & \longrightarrow & M[x := V] & & (\beta_v) \\ \lambda x.Mx & \longrightarrow & M & x \notin FV(M) & (\eta) \\ \lambda x.Vx & \longrightarrow & V & x \notin FV(V) & (\eta_v) \end{array}$$

The set of inference rules is identical for all λ-calculi. It extends the notions of reductions to an equivalence relation compatible with syntactic contexts:

$$\begin{array}{rcllr} M \longrightarrow N & \Rightarrow & C[M] = C[N] & & (\mathit{Comp}) \\ & & M = M & & (\mathit{Ref}) \\ M = L, L = N & \Rightarrow & M = N & & (\mathit{Trans}) \\ M = N & \Rightarrow & N = M & & (\mathit{Sym}) \end{array}$$

The underlying set of axioms completely identifies a theory. For example, β generates the theory $\boldsymbol{\lambda}$, β_v generates the theory $\boldsymbol{\lambda}_v$, and the union of β and η generates the theory $\boldsymbol{\lambda}\beta\eta$. In general, we write $\boldsymbol{\lambda}A$ to refer to the theory generated by a set of axioms A. When a theory $\boldsymbol{\lambda}A$ proves an equation $M = N$, we write $\boldsymbol{\lambda}A \vdash M = N$. If the proof does not involve the inference rule *Sym*, we write $\boldsymbol{\lambda}A \vdash M \twoheadrightarrow N$.

A notion of reduction R is Church-Rosser (CR) if $\boldsymbol{\lambda}R \vdash M = N$ implies that there exists a term L such that both M and N reduce to L, i.e., $\boldsymbol{\lambda}R \vdash M \twoheadrightarrow L$ and $\boldsymbol{\lambda}R \vdash N \twoheadrightarrow L$. A term M is in R-normal form if there are no R-reductions starting with M.

Semantics: The semantics of the language Λ is a function *Eval*, from programs to answers. A *program* is a term with no free variables and, in practical languages, an *answer* is a member of the syntactic category of values. Typically, *Eval* is defined via an abstract machine that manipulates abstract counterparts to machine stacks, stores, registers, etc. Examples are the SECD machine [14] and the CEK machine [7].

An equivalent method for specifying the semantics is based on the Curry-Feys Standard Reduction theorem [7, 17]. The Standard Reduction theorem defines a partial function, $\longmapsto$, from programs to programs that corresponds to a single evaluation step of an abstract machine for Λ.

A standard step (i) decomposes the program into a context E and a leftmost-outermost redex R (not inside an abstraction), and (ii) fills E with the contractum of R. The special contexts, E, are *evaluation contexts* and have the following definition for the call-by-value and call-by-name variants of Λ, respectively [7]:

$$\begin{array}{rcl} E_v & ::= & [\,] \mid E_v[(V\ [\,])] \mid E_v[([\,]\ M)] \\ E_n & ::= & [\,] \mid E_n[([\,]\ M)] \end{array}$$

Conceptually, the hole of an evaluation context, $[\,]$, points to the current instruction, which must be a β_v or β redex. The decomposition of M into $E[(V\ N)]$ where

$(V\ N)$ is a redex means that the current instruction is $(V\ N)$ and that the rest of the computation (the continuation! [7]) is E. Since, a call-by-name language never evaluates arguments, evaluation contexts do not include contexts of the shape $E_n[(V\ [\,])]$.

Given evaluation contexts, the definitions of the standard reduction functions for call-by-value and call-by-name respectively are as follows:

$$\begin{array}{rcl} E_v[((\lambda x.M)\ V)] & \longmapsto_v & E_v[M[x := V]] \\ E_n[((\lambda x.M)\ N)] & \longmapsto_n & E_n[M[x := N]] \end{array}$$

A complete evaluation applies the single-step functions repeatedly and either reaches an answer or diverges. The notation $\longmapsto^*$ denotes the reflexive, transitive closure of the function $\longmapsto$. The semantics of Λ is defined as follows:

$$\begin{array}{rcll} \mathit{Eval}_v(M) & = & V \text{ iff } M \longmapsto_v^* V & \text{(call-by-value)} \\ \mathit{Eval}_n(M) & = & V \text{ iff } M \longmapsto_n^* V & \text{(call-by-name)} \end{array}$$

For the definition of the semantics, η and η_v do not play any role. Their relevance for calculi is clarified in the next paragraph.

Note: The syntax of the call-by-value language Λ can be redefined as follows:

$$\begin{array}{rrcl} \Lambda: & M & ::= & V \mid E[(V\ V)] \\ \mathit{Values}: & V & ::= & x \mid (\lambda x.M) \\ \mathit{EvCont}: & E & ::= & [\,] \mid E[(V\ [\,])] \mid E[([\,]\ M)] \end{array}$$

We use both definitions of the syntax below.

Observational Equivalence: Not only do calculi define the semantics of Λ, but they are also useful for proving the correctness of some *optimizations*. Abstractly, an optimization of a program $C[M]$ is the replacement of M by a "more efficient" expression N such that a programmer cannot distinguish the observational behavior of the programs $C[M]$ and $C[N]$. The observational behavior of a program includes its termination behavior and its value when it terminates; it does not include execution speed. Formally, two expressions M and N are observationally equivalent, $M \cong_x N$ (for $x = v$ or $x = n$), if the following condition holds:

> For all contexts C such that both $C[M]$ and $C[N]$ are programs, either both $\mathit{Eval}_x(C[M])$ and $\mathit{Eval}_x(C[N])$ are defined or both are undefined.

It is undecidable to determine whether two expressions are observationally equivalent. However, $\boldsymbol{\lambda}_v$ and $\boldsymbol{\lambda}$ are two typical (weak) examples of theories that are sound with respect to observational equivalence.

Theorem 2.1 (Plotkin) *Let $M, N \in \Lambda$.*

1. If $\boldsymbol{\lambda}_v \vdash M = N$ then $M \cong_v N$.

2. If $\boldsymbol{\lambda} \vdash M = N$ then $M \cong_n N$.

The soundness of extensions of $\boldsymbol{\lambda}$ and $\boldsymbol{\lambda}_v$ with η and η_v, respectively, depends on the circumstances. The axiom η_v is sound with respect to call-by-value observational equivalence for Λ. If we extend Λ with constants, η_v may be unsound. For an example, consider a dynamically typed language with numerals and a predicate *integer?*. The latter can distinguish 3 and $\lambda x.3\ x$, yet, the η_v axiom identifies the two terms. In a typed setting, η_v is generally sound, independent of the parameter-passing technique.

The axiom η, on the other hand, fails to be sound with respect to call-by-name observational equivalence even in a pure language. For example, if Ω is a diverging term, then $(\lambda x.\Omega x)$ reduces to Ω but the two are clearly observationally distinct terms. Indeed, η is only sound in a typed language that does not permit the observation of the termination behavior of higher-type expressions.

3 The Origins and Practice of CPS

The idea of transforming programs to "continuation-passing style" appeared in the mid-sixties. For a few years, the transformation remained part of the folklore of computer science until Fischer and Reynolds codified it in 1972.

Fischer [10] studied two implementation strategies for Λ: a heap-based retention strategy, in which all variable bindings are retained until no longer needed, and a stack-based deletion strategy, in which variable bindings are destroyed when control leaves the procedure (or block) in which they were created. He concluded that

> no real power is lost in restricting oneself to a deletion strategy implementation, for any program can be translated into an equivalent one which will work correctly under such an implementation [10: 104].

The translation is the following CPS transformation.

Definition 3.1. (*Fischer CPS*) Let k, m, $n \in \mathit{Vars}$ be variables that do not occur in the argument to $\mathcal{F}$.

$$\begin{array}{rcl} \mathcal{F}:\Lambda & \rightarrow & \Lambda \\ \mathcal{F}[\![V]\!] & = & \lambda k.k\ \Psi[\![V]\!] \\ \mathcal{F}[\![MN]\!] & = & \lambda k.\mathcal{F}[\![M]\!]\ (\lambda m.\mathcal{F}[\![N]\!]\ \lambda n.(m\ k)\ n) \end{array}$$

$$\begin{array}{rcl} \Psi[\![x]\!] & = & x \\ \Psi[\![\lambda x.M]\!] & = & \lambda k.\lambda x.\mathcal{F}[\![M]\!]\ k \end{array}$$

Reynolds [18] investigated definitional interpreters for higher-order languages. Among his goals was the desire to liberate the definition of a language from the parameter-passing technique of the defining language. He developed a constructive (but informal) method to transform an interpreter such that it becomes indifferent to whether the underlying parameter passing technique is call-by-value or call-by-name. His transformation is essentially the same transformation as Fischer's $\mathcal{F}$. Plotkin [17] later proved Reynolds' ideas correct.

Theorem 3.2 (Plotkin [17]) *Let $M \in \Lambda$.*

Simulation: $\Psi[\![Eval_v(M)]\!] = Eval_n(\mathcal{F}[\![M]\!]\ (\lambda x.x))$

Indifference:
$Eval_n(\mathcal{F}[\![M]\!]\ (\lambda x.x)) = Eval_v(\mathcal{F}[\![M]\!]\ (\lambda x.x))$

The Simulation theorem shows that the evaluation of the CPS program produces correct outputs. The Indifference theorem establishes that this evaluation yields the same result under call-by-value and call-by-name.

The main disadvantage of the Fischer CPS transformation is the excessive number of redexes it introduces in the output. For example,

$$\begin{aligned}&\mathcal{F}[\![((\lambda x.x)\ (y\ y))]\!] = \\ &\quad \lambda k.((\lambda k.k\ \lambda k.\lambda x.((\lambda k.kx)\ k)) \\ &\qquad (\lambda m.((\lambda k.((\lambda k.ky)\ \lambda m.\lambda k.ky\ \lambda n.(m\ k)\ n)) \\ &\qquad\qquad (\lambda n.(m\ k)\ n)))).\end{aligned}$$

Although the original term contains one λ-abstraction and no β_v-redexes, its CPS counterpart contains a large number of both. Plotkin [17] referred to the new redexes as *administrative* redexes because an evaluator must always reduce them before re-establishing β_v-redexes that were present in the source term.

From both a theoretical and a practical perspective, the presence of the administrative redexes is undesirable. On the theoretical side, they complicate reasoning about CPS programs. For example, Plotkin [17] finds it necessary to define an improved CPS transformation exclusively for the proof of Theorem 3.2 above. On the practical side, code generation phases in compilers favor smaller, i.e., more manageable, programs. Hence, "practical" CPS transformations [1, 5, 13, 19, 20] use special algorithms to minimize the size of their outputs.

In essence, all practical CPS transformations are conceptually equivalent to the following two-pass CPS transformation:[1]

- First, "mark" the new λ-abstractions in the output of the Fischer CPS to identify administrative redexes, and then
- reduce all administrative redexes.

Source redexes should remain intact because unrestricted reductions could cause non-termination. The remainder of this section codifies these ideas in a simple manner.

Formally, the first pass of the two-pass CPS is the following modified Fischer CPS transformation.

[1]See also Danvy and Filinski's development of these ideas [5].

Definition 3.3. (*Modified Fischer CPS*) Let $k, m, n \in$ *Vars* be as in Definition 3.1.

$$\begin{aligned}\mathcal{F}[\![V]\!] &= \overline{\lambda} k.k\ \Psi[\![V]\!] \\ \mathcal{F}[\![MN]\!] &= \overline{\lambda} k.\mathcal{F}[\![M]\!]\ (\overline{\lambda} m.\mathcal{F}[\![N]\!]\ \overline{\lambda} n.(m\ k)\ n)\end{aligned}$$

$$\begin{aligned}\Psi[\![x]\!] &= x \\ \Psi[\![\lambda x.M]\!] &= \overline{\lambda} k.\lambda x.\mathcal{F}[\![M]\!]\ k\end{aligned}$$

An overline decorates λ-abstractions that were not present in the original term. An administrative reduction is simply one that involves decorated abstractions:

$$\begin{array}{rcll}((\overline{\lambda} x.M)\ N) & \longrightarrow & M[x := N] & (\overline{\beta}) \\ (\overline{\lambda} x.Mx) & \longrightarrow & M \quad x \notin FV(M) & (\overline{\eta})\end{array}$$

The complete definition of the two-pass CPS transformation, $\mathcal{F}2_k$, is parametrized over a continuation k.

Definition 3.4. (*Two-Pass CPS*) $\mathcal{F}2_k[\![M]\!] = P$ iff $\lambda\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![M]\!]\ k) = P$ where P is in $\overline{\beta}\overline{\eta}$-normal form.

The following proposition establishes that $\mathcal{F}2_k$ is well-defined.

Proposition 3.5 *$\mathcal{F}2_k$ is a total function.*

Proof. By Lemma 3.7, $\overline{\beta}\overline{\eta}$-normal forms are unique. Therefore, the relation $\mathcal{F}2_k$ is a function. Moreover, by Lemma 3.6, all reductions paths starting at $\mathcal{F}[\![M]\!]$ for $M \in \Lambda$ terminate. Hence, $\mathcal{F}2_k$ is a total function. ∎

Lemma 3.6 *Let $M \in \Lambda$. If $\lambda\overline{\beta}\overline{\eta} \vdash \mathcal{F}[\![M]\!] \equiv M_0 \longrightarrow M_1 \longrightarrow M_2 \cdots$, then:*

1. *for all M_i, the bound variable of a $\overline{\lambda}$-abstraction occurs exactly once in the body,*
2. *for all $i \geq 0$, M_{i+1} has one less $\overline{\lambda}$-abstraction than M_i, and*
3. *for some finite n, M_n is in $\overline{\beta}\overline{\eta}$-normal form.*

Proof Sketch. The first claim is initially true by construction, and is preserved by $\overline{\beta}\overline{\eta}$-reductions. It implies that reductions cannot eliminate or duplicate subterms. Therefore, the second claim holds. The last claim follows by induction on the number of $\overline{\lambda}$-abstractions in $\mathcal{F}[\![M]\!]$. ∎

It remains to establish that if $\mathcal{F}[\![M]\!]$ reduces to two normal forms P and Q, then P and Q are identical.

Lemma 3.7 *Let P and Q be in $\overline{\beta}\overline{\eta}$-normal form. If $\lambda\overline{\beta}\overline{\eta} \vdash \mathcal{F}[\![M]\!] \twoheadrightarrow P$ and $\lambda\overline{\beta}\overline{\eta} \vdash \mathcal{F}[\![M]\!] \twoheadrightarrow Q$, then $P \equiv Q$.*

Proof. The proof is a consequence of the Church-Rosser theorem for $\beta\eta$ [2]. ∎

The output of $\mathcal{F}2_k$ is extremely compact. For example, applying $\mathcal{F}2_k$ to $(((\lambda x.\lambda y.x)\ a)\ b)$ yields the term:

$$M \overset{df}{\equiv} ((\lambda x.((\lambda y.kx)\ b))\ a)$$

For the same example, both Steele's Rabbit transformation [20] and the Danvy/Filinski transformation [5] yield the term:[2]

$$N \overset{df}{\equiv} ((\lambda k_1 x.(k_1\ \lambda k_2 y.k_2 x))\ (\lambda m.mkb)\ a).$$

In the term N, every procedure accepts its continuation at the same time it accepts its argument. Therefore, the management of *all* continuations in the term N must occur at run-time. Specifically, the evaluation of M requires two β-reductions:

$$M \longrightarrow ((\lambda y.ka)\ b) \longrightarrow ka,$$

while the evaluation of N requires three (n-ary) β-reductions:

$$\begin{aligned} N &\longrightarrow ((\lambda m.mkb)\ (\lambda k_2 y.k_2 a)) \\ &\longrightarrow ((\lambda k_2 y.k_2 a)\ k\ b) \\ &\longrightarrow ka. \end{aligned}$$

Since the extra (administrative) reduction in the evaluation of N is completely predicatable from the source term, the function $\mathcal{F}2_k$ optimizes it away.

4 Transforming CPS programs

With the elimination of all administrative redexes, we can turn our attention to "interesting" $\beta\eta$ transformations on CPS programs.

Plotkin [17] was the first to offer some insights about the relation between reductions on source terms and CPS terms. In a comparative study of equational theories for call-by-value languages and call-by-name languages, he proved the following theorem.

Theorem 4.1 (Plotkin [17]) *Let* $M, N \in \Lambda$.

1. $\lambda_v \vdash M = N$ *implies* $\lambda_v \vdash \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!]$;
2. $\lambda_v \vdash \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!]$ *does not imply* $\lambda_v \vdash M = N$;
3. $\lambda_v \vdash \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!]$ *iff* $\lambda \vdash \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!]$.

[2]This is slightly inaccurate. In both Steele's Rabbit and the Danvy/Filinski CPS transformations, the continuation is the second parameter to a procedure. Thus, their output is actually:

$$((\lambda x k_1.(k_1\ \lambda y k_2.k_2 x))\ a\ (\lambda m.mbk)).$$

Even though this term only contains source redexes, we could still optimize it by equational reasoning as the following derivation shows:

$$\begin{aligned} & ((\lambda x k_1.(k_1 \lambda y k_2.k_2 x))\ a\ (\lambda m.mbk)) \\ \longrightarrow\ & ((\lambda m.mbk)\ (\lambda y k_2.k_2 a)) \\ \longrightarrow\ & ((\lambda y k_2.k_2 a)\ b\ k) \\ \longrightarrow\ & ka \\ \longleftarrow\ & ((\lambda y.ka)\ b) \\ \longleftarrow\ & ((\lambda x.((\lambda y.kx)\ b))\ a). \end{aligned}$$

Indeed, the "net" effect of such transformations is that of performing administrative reductions only!

In short, β-reductions prove more equations on CPS terms than β_v-reductions prove on source terms. The effect of η-reductions on CPS terms is unknown. Our goal is to remedy this situation by deriving a set of reductions A such that:

$$\lambda_v A \vdash M = N \quad \text{iff } \lambda\beta\eta \vdash \mathcal{F}2_k[\![M]\!] = \mathcal{F}2_k[\![N]\!]$$

We illustrate some of the complications that this problem poses with a specific reduction on CPS terms:

$$((\lambda x.((x\ k)\ z))\ (\lambda k.\lambda y.ky)) \longrightarrow (((\lambda k.\lambda y.ky)\ k)\ z).$$

By inspection, the left-hand side is:

$$\mathcal{F}2_k[\![((\lambda x.xz)\ (\lambda y.y))]\!].$$

A quick glance at the right hand side reveals that it contains an administrative redex and hence cannot be $\mathcal{F}2_k[\![M]\!]$ for any $M \in \Lambda$. The right hand side is, however, provably equal to a number of CPS terms;

$$\begin{aligned} \lambda\beta\eta \vdash\ & \mathcal{F}2_k[\![((\lambda a.((\lambda x.xa)\ (\lambda y.y)))\ z)]\!] \\ =\ & \mathcal{F}2_k[\![((\lambda y.y)\ z)]\!] \\ =\ & (((\lambda k.\lambda y.ky)\ k)\ z). \end{aligned}$$

Assuming we choose $((\lambda y.y)\ z)$ as the official "inverse" of the right hand side, then the CPS reduction corresponds to the following β_v-reduction:

$$((\lambda x.xz)\ (\lambda y.y)) \longrightarrow ((\lambda y.y)\ z)$$

on source terms. The other choice corresponds to a β_v-*expansion* which is clearly undesirable.

Inspired by the above example, we proceed as follows:

1. We explicitly define the set of CPS terms. The definition relies a one-pass CPS transformation equivalent to $\mathcal{F}2_k$ (Section 5).
2. We define an "inverse" CPS transformation and formalize its precise relationship to the CPS transformation (Section 6).
3. We derive the set A. For each notion of reduction $P \longrightarrow Q$ on CPS terms, we apply the inverse transformation to P and Q and get the source terms M and N. If $\lambda_v \vdash M = N$, then we are done. Otherwise, we add appropriate reductions to A (Section 7).

5 A compactifying CPS transformation and the CPS language

The one-pass CPS transformation should combine the modified Fischer transformation with the application of

$\overline{\beta}$- and $\overline{\eta}$-reductions. An informal description of what these reductions accomplish will clarify the nature of such a function.

The most informative kind of administrative redexes appears in the translation of $((\lambda x.M)\ V)$ in an arbitrary continuation K:

$$\begin{array}{l}((\overline{\lambda} k.((\overline{\lambda} k.k\ (\overline{\lambda} k.\lambda x.\mathcal{F}[\![M]\!]\ k)) \\ \qquad (\overline{\lambda} m.((\overline{\lambda} k.k\ \Psi[\![V]\!])\ (\overline{\lambda} n.(mk)n))))) \\ \quad K).\end{array}$$

The expression reduces to:

$$(((\overline{\lambda} k.\lambda x.\mathcal{F}[\![M]\!]\ k)\ K)\ \Psi[\![V]\!]).$$

At this point, the following $\overline{\beta}$-reduction takes place:

$$(((\overline{\lambda} k.\lambda x.\mathcal{F}[\![M]\!]\ k)\ K)\ \Psi[\![V]\!]) \longrightarrow ((\lambda x.\mathcal{F}[\![M]\!]\ K)\ \Psi[\![V]\!])$$

i.e., the image of the abstraction absorbs the continuation of the application. For the source terms, this means that the *body* of an abstraction in application position absorbs the syntactic representation of the continuation, which is the evaluation context [7]. Thus, a program of the shape $E[((\lambda x.M)\ V)]$ where E represents K, must be translated as if it had been written as $((\lambda x.E[M])\ V)$.

Put differently, our CPS transformation "symbolically" evaluates redexes by lifting them to the root of the program. For applications of values to values inside of λ-abstractions, this means of course that it takes the evaluation contexts with respect to the closest λ, which will become the root of the program once the redex is discharged. The resulting transformation, $\mathcal{C}_k$, is parametrized over a variable k that represents the continuation of the entire program.

Definition 5.1. ($\mathcal{C}_k, \Phi, \mathcal{K}_k$) The CPS transformation uses three mutually recursive functions: $\mathcal{C}_k$ to transform terms, Φ to transform values, and $\mathcal{K}_k$ to transform evaluation contexts. Let $k, u \in$ *Vars* be variables that do occur in the argument to $\mathcal{C}_k$.

$$\begin{array}{rcl}\mathcal{C}_k : \Lambda & \longrightarrow & \Lambda \\ \mathcal{C}_k[\![V]\!] & = & (k\ \Phi[\![V]\!]) \\ \mathcal{C}_k[\![E[(x\ V)]]\!] & = & ((x\ \mathcal{K}_k[\![E]\!])\ \Phi[\![V]\!]) \\ \mathcal{C}_k[\![E[((\lambda x.M)\ V)]]\!] & = & ((\lambda x.\mathcal{C}_k[\![E[M]]\!])\ \Phi[\![V]\!])\end{array}$$

$$\begin{array}{rcl}\Phi[\![x]\!] & = & x \\ \Phi[\![\lambda x.M]\!] & = & \overline{\lambda} k.\lambda x.\mathcal{C}_k[\![M]\!]\end{array}$$

$$\begin{array}{rcl}\mathcal{K}_k[\![[\,]]\!] & = & k \\ \mathcal{K}_k[\![E[(x\ [\,])]]\!] & = & (x\ \mathcal{K}_k[\![E]\!]) \\ \mathcal{K}_k[\![E[((\lambda x.M)\ [\,])]]\!] & = & (\lambda x.\mathcal{C}_k[\![E[M]]\!]) \\ \mathcal{K}_k[\![E[([\,]\ M)]]\!] & = & (\overline{\lambda} u.\mathcal{C}_k[\![E[(u\ M)]]\!])\end{array}$$

An informal examination of the definition of $\mathcal{C}_k$ reveals that the translation of every expression refers to an expression of smaller "size".

Definition 5.2. (*Size*) The *size* of a term M, $|M|$, is the number of variables in M (including binding occurrences). The *size* of a context E, $|E|$, is the number of variables in E (including binding occurrences) plus 2. (Because the empty context replaces a redex which has at least size 2, the size of the empty context is 2.)

Hence, the function $\mathcal{C}_k$ is well-defined.

Proposition 5.3 *The function $\mathcal{C}_k$ is total.*

Proof Sketch. The proof is by induction on the size of the argument to $\mathcal{C}_k$ or $\mathcal{K}_k$ and proceeds by cases on possible inputs to the two functions. ∎

As expected the output of $\mathcal{C}_k$ does not contain any administrative redexes.

Lemma 5.4 *If $M \in \Lambda$, $\mathcal{C}_k[\![M]\!]$ is in $\overline{\beta}\overline{\eta}$-normal form.*

Proof Sketch. By induction on the size of the argument to $\mathcal{C}_k$ or $\mathcal{K}_k$. ∎

Most importantly, the output of the new CPS transformation is equivalent to the output of $\mathcal{F}2_k$.

Proposition 5.5 *Let $M \in \Lambda$. Then,*

$$\lambda\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![M]\!]\ k) \twoheadrightarrow \mathcal{C}_k[\![M]\!].$$

Proof Sketch. The essential steps in the proof are:

1. Define a natural extension of the modified Fischer transformation that acts on contexts such that $\mathcal{F}[\![[\,]]\!] = \overline{\lambda} k.k$.

2. By a simple induction on the structure of E, prove:
$$\lambda\overline{\beta} \vdash (\mathcal{F}[\![E[M]]\!]\ k) = (\mathcal{F}[\![M]\!]\ (\mathcal{F}[\![E]\!]\ k)).$$

3. By induction on the size of M, prove that:
$$\lambda\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![M]\!]\ k) = \mathcal{C}_k[\![M]\!].$$

4. The result follows from the last claim because $\mathcal{C}_k[\![M]\!]$ is in $\overline{\beta}\overline{\eta}$-normal form (Lemma 5.4) and $\overline{\beta}\overline{\eta}$-normal forms are unique (Lemma 3.7). ∎

With the completion of the analysis of $\mathcal{C}_k$, the decorating overlines become irrelevant. Therefore, in the remainder of the paper, we ignore the distinction between λ and $\overline{\lambda}$.

Next, we turn to the definition of our universe of discourse, the set of CPS terms. It must include all terms that contribute to the proofs of equations of the form:

$$\lambda\beta\eta \vdash \mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!].$$

Since $\beta\eta$ is *CR* [2], it is sufficient to consider equations of the form:

$$\lambda\beta\eta \vdash \mathcal{C}_k[\![M]\!] \twoheadrightarrow P.$$

Hence, the interesting set of CPS terms is:

$$S \overset{df}{\equiv} \{P \mid \exists M \in \Lambda . \boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] \twoheadrightarrow P\}.$$

The definition of the transformation $\mathcal{C}_k$ provides some insight about an inductive characterization of the set of CPS terms. According to the right hand sides of the equations in Definition 5.1 all terms in the CPS language are an application of a continuation to a value. Values are either variables or abstractions (continuation transformers). Continuations are either variables, or the result of the application of a value to a continuation, or a regular lambda abstraction. Therefore, we claim that S is generated by the following grammar.

Definition 5.6. (*CPS grammar*) Let $x \in Vars \backslash \{k\}$:

$$\begin{array}{rrcl} cps(\Lambda): & P & ::= & (K\ W) \\ cps(Values): & W & ::= & x \mid (\lambda k.K) \\ cps(EvCont): & K & ::= & k \mid (W\ K) \mid (\lambda x.P) \end{array}$$

Note: The special status reserved for the variable k ensures that the continuation parameter occurs exactly once in each abstraction $\lambda k.K$. When working with the quotient of the language under α-equivalence, the special status of k disappears.

The following theorem justifies the above claim by establishing the equivalence of the two definitions of the set of CPS terms.

Theorem 5.7 $S = cps(\Lambda)$.

Proof. It is easy to prove that for $M \in \Lambda$, $\mathcal{C}_k[\![M]\!] \in cps(\Lambda)$. The rest follows from Lemmas 5.8 and 5.9. ∎

Since $\beta\eta$-reductions preserve the syntactic categories in the CPS language, the set S is a subset of $cps(\Lambda)$.

Lemma 5.8 *Let $P_1 \in cps(\Lambda)$, $W_1 \in cps(Values)$, and $K_1 \in cps(EvCont)$. Then,*

- *if $\boldsymbol{\lambda}\beta\eta \vdash P_1 \longrightarrow P_2$ then $P_2 \in cps(\Lambda)$;*
- *if $\boldsymbol{\lambda}\beta\eta \vdash W_1 \longrightarrow W_2$ then $W_2 \in cps(Values)$;*
- *if $\boldsymbol{\lambda}\beta\eta \vdash K_1 \longrightarrow K_2$ then $K_2 \in cps(EvCont)$.*

For the opposite implication, i.e., that $cps(\Lambda)$ is a subset of S, it is sufficient that every $P \in cps(\Lambda)$ is reachable from Λ via $\mathcal{C}_k$ and $\beta\eta$.

Lemma 5.9 *For all $P \in cps(\Lambda)$, there exists an $M \in \Lambda$ such $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] \twoheadrightarrow P$.*

Proof. The proof is by lexicographic induction on $\langle \tilde{G}, |G| \rangle$ where G is an element P of $cps(\Lambda)$ or an element K of $cps(EvCont)$, $\tilde{G}$ is the number of abstractions of the form $\lambda k.K$ in G, and $|G|$ is the number of variables (including binding occurrences) in G. ∎

6 An Inverse CPS Transformation

Based on the inductive definition of the CPS language, the specification of an "inverse" to the CPS transformation is almost straightforward: the source term corresponding to the application of a continuation K to a value W is simply $E[V]$ where E is the evaluation context that syntactically represents the continuation K and V is the value that corresponds to W. The definition of the function $\mathcal{C}^{-1}$ (*un*-CPS) uses two auxiliary functions to translate continuations to evaluation contexts and values in the CPS language to values in the source language. Both definitions are straightforward.

Definition 6.1. $(\mathcal{C}^{-1}, \Phi^{-1}, \mathcal{K}^{-1})$

$$\begin{array}{rcl} \mathcal{C}^{-1} : cps(\Lambda) & \rightarrow & \Lambda \\ \mathcal{C}^{-1}[\![(K\ W)]\!] & = & \mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!]] \\ \\ \Phi^{-1}[\![x]\!] & = & x \\ \Phi^{-1}[\![(\lambda k.k)]\!] & = & \lambda x.x \\ \Phi^{-1}[\![(\lambda k.W\ K)]\!] & = & \lambda x.\mathcal{C}^{-1}[\![(W\ K)\ x]\!] \\ \Phi^{-1}[\![(\lambda k.\lambda x.P)]\!] & = & \lambda x.\mathcal{C}^{-1}[\![P]\!] \\ \\ \mathcal{K}^{-1}[\![k]\!] & = & [\,] \\ \mathcal{K}^{-1}[\![(x\ K)]\!] & = & \mathcal{K}^{-1}[\![K]\!][(x\ [\,])] \\ \mathcal{K}^{-1}[\![((\lambda k.K_1)\ K_2)]\!] & = & \mathcal{K}^{-1}[\![K_1[k := K_2]]\!] \\ \mathcal{K}^{-1}[\![(\lambda x.P)]\!] & = & ((\lambda x.\mathcal{C}^{-1}[\![P]\!])\ [\,]) \end{array}$$

Intuitively, $\mathcal{C}^{-1}$ is the "inverse" of $\mathcal{C}$, Φ^{-1} is the "inverse" of Φ, and $\mathcal{K}^{-1}$ is the "inverse" of $\mathcal{K}$. Moreover, $\mathcal{C}^{-1}$, Φ^{-1}, and $\mathcal{K}^{-1}$ apply to the syntactic categories $cps(\Lambda)$, $cps(Values)$, and $cps(EvCont)$ respectively and yield terms in the appropriate syntactic categories in the source language.

Lemma 6.2 $\mathcal{C}^{-1}[\![P]\!] \in \Lambda$, $\Phi^{-1}[\![W]\!] \in$ *Values, and* $\mathcal{K}^{-1}[\![K]\!] \in$ *EvCont.*

Proof Sketch. The proof is based on the same idea as the proof of Lemma 5.9. ∎

For two distinct reasons, $\mathcal{C}^{-1}$ cannot be a complete inverse of $\mathcal{C}_k$. First, some CPS terms are the image of more than one source term. Second, some CPS terms are not the image of any source term. The first fact is a property of the function $\mathcal{C}_k$ that reduces administrative redexes on the fly. The second one is due to the closure of the set of CPS terms under $\beta\eta$-reductions. We discuss each point in detail below.[3]

[3]Danvy and Lawall [3, 6] define a direct style transformation that maps CPS terms into source terms. To get an inverse of the Danvy/Filinski CPS [5], they have to restrict the domain of their direct style transformation to images of Λ terms. Hence, the Danvy-Lawall inverse is not applicable in situations that involve $\beta\eta$-transformations of CPS programs.

The effect of administrative reductions: The function $\mathcal{C}_k$ incorporates the reduction of all administrative redexes from the output of the Fischer CPS. Hence, if $\mathcal{F}[\![M]\!]$ and $\mathcal{F}[\![N]\!]$ reduce to a common term by administrative reductions only, $\mathcal{C}_k[\![M]\!]$ is *identical* to $\mathcal{C}_k[\![N]\!]$. The definition of the function $\mathcal{C}_k$ shows that, in two cases, different inputs are indeed mapped to the same output.

- The first equivalence is:

  $$\mathcal{C}_k[\![E[((\lambda x.M)\ N)]]\!] \equiv \mathcal{C}_k[\![((\lambda x.E[M])\ N)]\!].$$

  The equation illustrates how the CPS transformation uses its knowledge about the continuation of an application. As indicated in Section 4, it "lifts" the application to top level and merges the continuation with the body of the application.

- The second equivalence is:

  $$\mathcal{C}_k[\![E[((M\ N)\ L)]]\!] \equiv \mathcal{C}_k[\![((\lambda x.E[(x\ L)])\ (M\ N))]\!]$$

  This equation captures another essential element of CPS transformations. According to folklore in the functional compiler-building community [4], the first aspect of a CPS transformation is to give the value of every application a name. In the above equation, the argument to $\mathcal{C}_k$ in the right hand side is a "flattened" version of the left hand side in which the nested application $(M\ N)$ is factored out and given a name.

In summary, we define two reductions on Λ that capture the effect of the administrative reductions performed by $\mathcal{C}_k$:

$$\begin{array}{rcll} E[((\lambda x.M)\ N)] & \longrightarrow & ((\lambda x.E[M])\ N) & (\beta_{lift}) \\ E[((M\ N)\ L)] & \longrightarrow & ((\lambda x.E[(x\ L)]) & (\beta_{flat}) \\ & & (M\ N)) & \end{array}$$

By the variable conventions, $x \notin FV(E, L)$ in the above reductions.

The reductions β_{lift} and β_{flat} define equivalence classes of source terms that map to the same CPS term. The function $\mathcal{C}^{-1}$ maps this CPS term to a particular representative of the equivalence class: the element in $\beta_{lift}\beta_{flat}$-normal form.

Lemma 6.3 *Let $P \in cps(\Lambda)$. Then, $\mathcal{C}^{-1}[\![P]\!]$ is in $\beta_{lift}\beta_{flat}$-normal form.*

Proof Sketch. The proof is by induction on the same lexicographic measure as the proof of Lemma 5.9. ∎

It follows that $\mathcal{C}^{-1}$ is an inverse of $\mathcal{C}_k$ on the subset of source terms in $\beta_{lift}\beta_{flat}$-normal form.

Theorem 6.4 *Let $M \in \Lambda$, $P \in cps(\Lambda)$.*

1. $\lambda\beta_{lift}\beta_{flat} \vdash M \twoheadrightarrow (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]$.
2. $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!] \equiv M$ *for* $M = \mathcal{C}^{-1}[\![P]\!]$.

The closure of the set of CPS terms under $\beta\eta$-reductions: Because of arbitrary $\beta\eta$-reductions, the set of CPS terms includes terms that are not the image of any source term. For example, the following η-reduction generates such a term:

$$\mathcal{C}_k[\![\lambda x.x]\!] = \lambda k.\lambda x.kx \longrightarrow \lambda k.k$$

The function $\mathcal{C}^{-1}$ (conceptually) coerces $\lambda k.k$ first to $\lambda k.\lambda x.kx$, i.e., $\mathcal{C}^{-1}[\![\lambda k.k]\!] = \mathcal{C}^{-1}[\![\lambda k.\lambda x.kx]\!] = \lambda x.x$.

Thus, P is generally not identical to $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!]$. However, $\mathcal{C}_k$ is the inverse of $\mathcal{C}^{-1}$ on the subset of CPS terms that are images of source terms.

Theorem 6.5 *Let $P \in cps(\Lambda)$, $M \in \Lambda$.*

1. $\lambda\beta\eta \vdash (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!] = P$.
2. $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!] \equiv P$ *for* $P = \mathcal{C}_k[\![M]\!]$.

Proof Sketch. The proof is by induction on the same lexicographic measure as the proof of Lemma 5.9. ∎

7 Completeness and Soundness

Using the partial inverse of the CPS transformation, we can systematically derive a set of additional axioms A for $\boldsymbol{\lambda}_v$ such that $\boldsymbol{\lambda}_v A$ is complete for $\beta\eta$ reasoning about CPS programs. Once we have the new axiom set, we prove its soundness in the second subsection. In the last subsection, we briefly discuss the connection to Moggi's computational λ-calculus.

7.1 Completeness

By inspection of the inductive definition of the CPS language, the possible β- and η-reductions on CPS terms are as follows:

$$\begin{array}{rcll} ((\lambda x.P)\ W) & \longrightarrow & P[x := W] & (\beta_w) \\ ((\lambda k.K_1)\ K_2) & \longrightarrow & K_1[k := K_2] & (\beta_k) \\ (\lambda k.Wk) & \longrightarrow & W & (\eta_w) \\ (\lambda x.Kx) & \longrightarrow & K \quad x \notin FV(K) & (\eta_k) \end{array}$$

Put differently, for $cps(\Lambda)$, $\beta = \beta_w \cup \beta_k$ and $\eta = \eta_w \cup \eta_k$. We outline the derivation of reductions corresponding to η_k. Let $(\lambda x.Kx) \longrightarrow K$ where $x \notin FV(K)$. Applying $\mathcal{K}^{-1}$ to both sides of the reduction, we get:

$$((\lambda x.\mathcal{C}^{-1}[\![Kx]\!])\ [\,]) \longrightarrow \mathcal{K}^{-1}[\![K]\!].$$

To understand how the left hand side could reduce to the right hand side, we proceed by case analysis on K:

- $K \equiv k$: The reduction becomes:

  $$((\lambda x.x)\ [\,]) \longrightarrow [\,].$$

  Since the empty context generally stands for an arbitrary expression, A should therefore contain the reduction:

  $$((\lambda x.x)\ M) \longrightarrow M \qquad (\beta_{id})$$

$$
\begin{array}{rcllr}
((\lambda x.M)\ V) & \longrightarrow & M[x := V] & & (\beta_v) \\
(\lambda x.Vx) & \longrightarrow & V & x \notin FV(V) & (\eta_v) \\
E[((\lambda x.M)\ N)] & \longrightarrow & ((\lambda x.E[M])\ N) & x \notin FV(E) & (\beta_{lift}) \\
E[((M\ N)\ L)] & \longrightarrow & ((\lambda x.E[(x\ L)])\ (M\ N)) & x \notin FV(E, L) & (\beta_{flat}) \\
((\lambda x.x)\ M) & \longrightarrow & M & & (\beta_{id}) \\
((\lambda x.E[(y\ x)])\ M) & \longrightarrow & E[(y\ M)] & x \notin FV(E[y]) & (\beta_\Omega)
\end{array}
$$

Figure 1: Source Reductions: $A \overset{df}{\equiv} \{\eta_v, \beta_{lift}, \beta_{flat}, \beta_{id}, \beta_\Omega\}$

- $K \equiv (y\ K_1)$: The reduction becomes:

  $$((\lambda x.\mathcal{K}^{-1}[\![K_1]\!][(y\ x)])\ [\ ]) \longrightarrow \mathcal{K}^{-1}[\![K_1]\!][(y\ [\ ])].$$

  By a similar argument as above, we must add the following reduction to A:

  $$((\lambda x.E[(y\ x)])\ M) \longrightarrow E[(y\ M)] \qquad (\beta_\Omega)$$

- $K \equiv ((\lambda k.K_1)\ K_2)$: The reduction becomes:

  $$((\lambda x.\mathcal{C}^{-1}[\![K_1[k := K_2]\ x]\!])\ [\ ]) \longrightarrow \mathcal{K}^{-1}[\![K_1[k := K_2]]\!].$$

  Since the term $(K_1[k := K_2]\ x)$ has one less $\lambda k.K$ abstraction than K, the inductive hypothesis provides an appropriate equivalence.

- $K \equiv \lambda y.P$: By an β_v-reduction, the left hand side $((\lambda x.((\lambda y.\mathcal{C}^{-1}[\![P]\!])\ x))\ [\ ])$ reduces to $((\lambda x.\mathcal{C}^{-1}[\![P]\!][y := x])\ [\ ])$, which is identical to the right hand side.

The cases for the other reductions on CPS terms are similar but with more complicated proofs. The resulting set of source reductions, A, includes all the previously derived reductions and η_v: see Figure 1.

Figure 2 summarizes the source reductions corresponding to each CPS reduction. A β_w-reduction corresponds essentially to a β_v-reduction on source terms. Similarly, an η_w-reduction corresponds to η_v. However, the reduction of a β_w redex might create some administrative redexes for the management of continuations. The reductions of these redexes correspond to $\beta_{lift}\beta_{id}\beta_\Omega$-reductions on source terms.

By pasting together the proofs of the lemmas corresponding to each reduction, we get the following completeness theorem.

Theorem 7.1 (Completeness) *If* $\lambda\beta\eta \vdash P \twoheadrightarrow Q$ *then* $\lambda_v A \vdash \mathcal{C}^{-1}[\![P]\!] \twoheadrightarrow \mathcal{C}^{-1}[\![Q]\!]$.

7.2 Soundness

The set of source reductions in Figure 1 is *sound* with respect to the equational theory over CPS terms. In other words, for a source reduction $M \longrightarrow N$, we have $\lambda\beta\eta \vdash \mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!]$. In fact, we can prove the stronger results of Figure 3. The Soundness theorem is a direct consequence of these results.

Theorem 7.2 (Soundness) *If* $\lambda_v A \vdash M \twoheadrightarrow N$ *then* $\lambda\beta\eta \vdash \mathcal{C}_k[\![M]\!] \twoheadrightarrow \mathcal{C}_k[\![N]\!]$.

7.3 Correspondence

Unfortunately, if $\mathcal{C}_k[\![M]\!]$ reduces to $\mathcal{C}_k[\![N]\!]$, it is *not* necessarily the case that M reduces to N. A proof that $\mathcal{C}_k[\![M]\!]$ reduces to $\mathcal{C}_k[\![N]\!]$ in the CPS calculus translates to a proof that $\mathcal{C}^{-1}[\![\mathcal{C}_k[\![M]\!]]\!]$ reduces to $\mathcal{C}^{-1}[\![\mathcal{C}_k[\![N]\!]]\!]$ in the source calculus. Unless N is in $\beta_{lift}\beta_{flat}$-normal form, the latter proof does not imply that $M \twoheadrightarrow N$. Still, M is always provably equal to N.

Theorem 7.3 (Correspondence) *The calculi* $\lambda\beta\eta$ *and* $\lambda_v A$ *are equivalent in the following sense:*

1. $\lambda_v A \vdash M = (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]$.
2. $\lambda\beta\eta \vdash P = (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!]$.
3. $\lambda_v A \vdash M = N$ *iff* $\lambda\beta\eta \vdash \mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!]$.
4. $\lambda\beta\eta \vdash P = Q$ *iff* $\lambda_v A \vdash \mathcal{C}^{-1}[\![P]\!] = \mathcal{C}^{-1}[\![Q]\!]$.

Note: Other CPS transformations. The correspondence theorem does not depend on any specific aspects of $\mathcal{C}_k$ or $\mathcal{F}$. Rather the result is valid for *any* CPS transformation *cps* that satisfies the following condition for $M \in \Lambda$:

$$\lambda\beta\eta \vdash \mathcal{F}[\![M]\!] = cps(M).$$

7.4 The computational λ-calculus

The calculus $\lambda_v A$ is equivalent to an untyped variant of Moggi's computational λ-calculus λ_c [16]. Specifically, we ignore the types of expressions, eliminate product and computational expressions, re-interpret Moggi's **let**-expression as the usual abbreviation for a λ-application, and apply his **let**-axioms to the expanded expressions. The basic reductions of our variant of λ_c are β_v, η_v, β_{id} plus the additional reductions of Figure 4.

$$\begin{array}{llllllllll}
\text{If} & \lambda\beta_w & \vdash & P \longrightarrow Q & \text{then} & \lambda\beta_v\beta_{lift}\beta_{id}\beta_\Omega & \vdash & \mathcal{C}^{-1}[\![P]\!] & \twoheadrightarrow & \mathcal{C}^{-1}[\![Q]\!] \\
\text{If} & \lambda\beta_k & \vdash & P \longrightarrow Q & \text{then} & & & \mathcal{C}^{-1}[\![P]\!] & \equiv & \mathcal{C}^{-1}[\![Q]\!] \\
\text{If} & \lambda\eta_w & \vdash & P \longrightarrow Q & \text{then} & \lambda\eta_v & \vdash & \mathcal{C}^{-1}[\![P]\!] & \twoheadrightarrow & \mathcal{C}^{-1}[\![Q]\!] \\
\text{If} & \lambda\eta_k & \vdash & P \longrightarrow Q & \text{then} & \lambda\beta_v\beta_{id}\beta_\Omega & \vdash & \mathcal{C}^{-1}[\![P]\!] & \twoheadrightarrow & \mathcal{C}^{-1}[\![Q]\!]
\end{array}$$

Figure 2: The Completeness Lemma

$$\begin{array}{lllllllll}
\text{If } \lambda\beta_v & \vdash & M \longrightarrow N & \text{then} & \lambda\beta & \vdash & \mathcal{C}_k[\![M]\!] & \twoheadrightarrow & \mathcal{C}_k[\![N]\!] \\
\text{If } \lambda\eta_v & \vdash & M \longrightarrow N & \text{then} & \lambda\eta_w\eta_k & \vdash & \mathcal{C}_k[\![M]\!] & \twoheadrightarrow & \mathcal{C}_k[\![N]\!] \\
\text{If } \lambda\beta_{lift} & \vdash & M \longrightarrow N & \text{then} & & & \mathcal{C}_k[\![M]\!] & \equiv & \mathcal{C}_k[\![N]\!] \\
\text{If } \lambda\beta_{flat} & \vdash & M \longrightarrow N & \text{then} & & & \mathcal{C}_k[\![M]\!] & \equiv & \mathcal{C}_k[\![N]\!] \\
\text{If } \lambda\beta_{id} & \vdash & M \longrightarrow N & \text{then} & \lambda\eta_k & \vdash & \mathcal{C}_k[\![M]\!] & \twoheadrightarrow & \mathcal{C}_k[\![M]\!] \\
\text{If } \lambda\beta_\Omega & \vdash & M \longrightarrow N & \text{then} & \lambda\eta_k & \vdash & \mathcal{C}_k[\![M]\!] & \twoheadrightarrow & \mathcal{C}_k[\![N]\!]
\end{array}$$

Figure 3: The Soundness Lemma

$$\begin{array}{rcll}
((\lambda x_2.M)\ ((\lambda x_1.M_2)\ M_1)) & \longrightarrow & ((\lambda x_1.((\lambda x_2.M)\ M_2))\ M_1) & (Comp) \\
((M\ N)\ L) & \longrightarrow & ((\lambda x.x\ L)\ (M\ N)) & (let.1) \\
(V\ (M\ N)) & \longrightarrow & ((\lambda x.V\ x)\ (M\ N)) & (let.2)
\end{array}$$

Figure 4: Additional Reductions for the Computational λ-calculus

We prove the equivalence of our calculus and $\boldsymbol{\lambda}_c$ by showing how each calculus proves the reductions of the other. First, there is a set of common rules: β_v, η_v, and β_{id}. Moreover, *Comp* is an instance of β_{lift} and *let*.1 an instance of β_{flat}. Finally, *let*.2 is simply an η_v expansion. This establishes one direction of the equivalence. For the reverse direction, it is sufficient to show that $\boldsymbol{\lambda}_c$ proves β_{lift}, β_{flat}, and β_Ω for the cases when E is empty, $E \equiv (V\ [\ ])$, and $E \equiv ([\ ]\ M)$.

Based on the above argument, the correspondence of the calculi yields the following reformulation of the Correspondence theorem.

Theorem 7.3′ (Correspondence (Reformulation)) *The calculi $\boldsymbol{\lambda}\beta\eta$ and $\boldsymbol{\lambda}_c$ are equivalent in the sense of Theorem 7.3.*

8 Conclusion and Future Research

The interest in calculi is motivated by their soundness with respect to observational equivalence (see Section 2). Therefore, the natural question is whether our extension is sound with respect to the call-by-value observational equivalence relation. Moggi [16] proves the result in a typed setting. It follows that our extension is sound for reasoning about typed CPS programs, for example in a language like (full) ML.

For dynamically typed languages like Lisp or Scheme, the $\boldsymbol{\lambda}_c$-calculus is *unsound* since it includes the axiom η_v (See Section 2). As all other axioms are sound with respect to call-by-value observational equivalence, the relevant reductions for untyped languages are:

$$A^- \overset{df}{\equiv} \{\beta_{lift}, \beta_{flat}, \beta_{id}, \beta_\Omega\}.$$

By the Completeness and Soundness lemmas (Figures 2 and 3 respectively) the corresponding CPS calculus is $\boldsymbol{\lambda}\beta\eta_k$.

Theorem 7.3″ (Correspondence (Untyped)) *The calculi $\boldsymbol{\lambda}\beta\eta_k$ and $\boldsymbol{\lambda}_v A^-$ are equivalent in the sense of Theorem 7.3.*

In summary, our extensions of the λ_v-calculus result in an equational theory over Λ that is sound with respect to the call-by-value observational equivalence, and corresponds to $\boldsymbol{\lambda}\beta\eta$ (or $\boldsymbol{\lambda}\beta\eta_k$) over CPS terms. Moreover, the result extends to languages with ground constants and primitive functions and languages with imperative assignment procedures for data structures.

For languages with Scheme-like control operators, our extension of λ_v-calculus is still sound with respect to observational equivalence. However, the correspondence theorem fails since operators like *call/cc* manipulate their continuation in non-standard ways. To re-establish the correspondence theorem for such languages, we need to find an extension for the λ-control calculus [8, 9] that corresponds to $\boldsymbol{\lambda}\beta\eta$ on CPS terms.

References

1. Appel, A. and T. Jim. Continuation-passing, closure-passing style. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1982, 293–302.

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

3. Danvy, O. Back to direct style. In *4th Proc. European Symposium on Programming.* Springer Lecture Notes in Computer Science, 582. Springer Verlag, Berlin, 1992, 130–150.

4. Danvy, O. Three steps for the CPS transformation. Tech. Rep. CIS-92-2. Kansas State University, 1992.

5. Danvy, O. and A. Filinski. Representing control: A study of the CPS transformation. Tech. Rpt. CIS-91-2. Kansas State University, 1991.

6. Danvy, O. and J. Lawall. Back to direct style II: First-class continuations. In *Proc. 1992 ACM Conference on Lisp and Functional Programming,* 1992, this volume.

7. Felleisen, M. and D.P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, edited by M. Wirsing. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986, 193–217.

8. Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice University, June 1989. *Theor. Comput. Sci.*, 1991, to appear.

9. Felleisen, M., D. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theor. Comput. Sci.* **52**(3), 1987, 205–237. Preliminary version in: *Proc. Symposium on Logic in Computer Science*, 1986, 131–141.

10. Fischer, M.J. Lambda calculus schemata. In *Proc. ACM Conference on Proving Assertions About Programs, SIGPLAN Notices* **7**(1), 1972, 104–109.

11. Gateley, J. and B.F. Duba. Call-by-value combinatory logic and the lambda-value calculus. In *Proc. 1991 Workshop on Mathematical Foundations of Programming Semantics.* Lecture Notes in Computer Science 517, to appear.

12. Hieb R., R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990, 66–77.

13. Kranz, D., et al. ORBIT: An optimizing compiler for Scheme. In *Proc. SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices* **21**(7), 1986, 219–233.

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

15. Leroy, X. The Zinc experiement. Technical Report 117. INRIA, 1990.

16. Moggi, E. Computational lambda-calculus and monads. In *Proc. Symposium on Logic in Computer Science*, 1989, 14–23. Also appeared as: LFCS Report ECS-LFCS-88-66, University of Edinburgh, 1988.

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

18. Reynolds, J.C. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference*, 1972, 717–740.

19. Shivers, O. Control-flow Analysis of Higher-Order Languages or Taming Lambda. Ph.D. dissertation, Carnegie-Mellon University, 1991.

20. Steele, G.L., Jr. RABBIT: A compiler for SCHEME. Memo 474, MIT AI Lab, 1978.