A Reflection on Call-by-Value

Amr Sabry
University of Oregon
Eugene, OR 97403
sabry@cs.uoregon.edu

Philip Wadler
University of Glasgow
Glasgow G12 8QQ, Scotland
wadler@dcs.gla.ac.uk

Technical Report CIS-TR-96-08
April 1996
University of Oregon
Eugene, OR 97403
(541) 346-4408

Copyright ©1996 by Amr Sabry and Philip Wadler

First Edition: April 25, 1996

A Reflection on Call-by-Value

Amr Sabry
University of Oregon
Eugene, OR 97403
sabry@cs.uoregon.edu

Philip Wadler
University of Glasgow
Glasgow G12 8QQ, Scotland
wadler@dcs.glasgow.ac.uk

Abstract

A number of compilers exploit the following strategy: translate a term to continuation-passing style (CPS) and optimize the resulting term using a sequence of reductions. Recent work suggests that an alternative strategy is superior: optimize directly in an extended source calculus. We suggest that the appropriate relation between the source and target calculi may be captured by a special case of a *Galois connection* known as a *reflection*. Previous work has focused on the weaker notion of an *equational correspondence*, which is based on equality rather than reduction. We use as our source language Moggi's computational lambda calculus λ_c, which is an extension of Plotkin's call-by-value calculus λ_v. We show that Moggi's monad translation, Plotkin's CPS translation, and Girard's translation to linear logic can all be regarded as reflections from this source language, and we put forward λ_c as a model of call-by-value computation that improves on λ_v.

1 Introduction

Like the wheel, continuation-passing style, or CPS, is such an excellent idea that it was reinvented many times (Reynolds 1993). It was formalized by Plotkin (1975), and utilized in compilers for higher-order call-by-value languages written by Steele (1978), Kranz *et al.* (1986), Appel (1992), and others.

The front end for all these compilers is similar: translate a term to CPS and optimize. In symbols,

$$M^* \twoheadrightarrow_T P, \tag{1}$$

where M is a term of the source language S, P is a term of the target language T (terms in CPS), $* : S \to T$ is the compiling transform (the CPS translation), and $\twoheadrightarrow_T$ denotes reduction in T.

Recently a number of researchers have suggested an alternative way to build a compiler: rather than translate the term to CPS and optimize, perform the optimizations in an extended source calculus. In symbols,

$$M \twoheadrightarrow_S P^\#, \tag{2}$$

where $\# : T \to S$ is the decompiling transform, and $\twoheadrightarrow_S$ denotes reduction in the (extended) source calculus.

If equation (1) holds exactly when equation (2) holds, we say that maps $*$ and $\#$ form a *Galois connection* from S to T. Galois connections guarantee that for each optimization in the target language there is some corresponding optimization in the source language. We might also reasonably require that compiling is the inverse of decompiling, so $P^{\#*}$ is identical to P. A Galois connection satisfying this additional constraint is called a *reflection* in S of T. Whenever a reflection exists, there must be a subset of S that is isomorphic to all of T; we call this subset the *kernel*.

Our claim is that a *reflection* describes a situation of interest to compiler writers: it relates the intermediate languages of direct and CPS compilers and enables direct compilers to work with a representation that is isomorphic to CPS terms. Previous work, such as Sabry and Felleisen (1993) and Hatcliff and Danvy (1994), has focused on the weaker notion of *equational correspondence*, which is based on equality rather than reduction.

We show that three widely-studied translations may be regarded as reflections. Our source calculus is the computational lambda calculus, λ_c, of Moggi (1988), which extends the call-by-value calculus λ_v of Plotkin (1975). Our first translation is into the monadic meta-language, λ_{ml}, also of Moggi (1988). Our second translation is the CPS translation, also of Plotkin (1975). The monad translation may be regarded as a generalization of the CPS translation, and indeed the latter translation factors through the former. Our third translation is into a linear calculus studied by Wadler (1993) and Maraist *et al.* (1995).

One might hope that the variant CPS translation of Fischer (1972) is also a reflection. An earlier version of this paper (Sabry and Wadler 1996) claimed this was the case, but gave no proof. We should have been more cautious. Here we show that it is impossible for the Fischer CPS translation to be a reflection.

The existence of reflections in λ_c of the monadic calculus, the linear calculus, and the CPS calculus, all of which are well-established models of computations, leads us to put forward λ_c as a model of call-by-value computation that improves on λ_v.

The computational lambda calculus λ_c. Moggi (1988) introduced monads as a general notion of computation. The monadic meta-language λ_{ml} was designed to express any semantics based on monads, and the computation lambda calculus λ_c was designed as an extension of the call-by-value lambda calculus that is sound and complete for any monadic semantics. So, by design, two terms are equal in λ_c if and only if their translations are equal in λ_{ml}.

What is surprising is that little attention was paid to reductions. The original technical report (Moggi 1988) specified theories of reduction and of equality for λ_c, but only the equality theory of λ_c appears in the conference paper (Moggi 1989), and λ_c rates barely a line in the journal version (Moggi 1991). None of these contains a reduction theory for λ_{ml}, but this was considered by Hatcliff and Danvy (1994). However, ours is the first work we know of to relate the reductions of λ_c and λ_{ml}.

Moggi (1988) presents λ_c as an untyped calculus of reductions, and presents λ_{ml} as a typed calculus of equalities. Here we uniformly use untyped calculi of reductions. Everything works equally well for typed calculi of reductions, such as those considered by Hatcliff and Danvy (1994), since our translations preserve types.

Related work. As noted above, a number of researchers have considered ways to reflect the benefits of CPS within an extended source calculus. Sabry and Felleisen (1993) proposed such a calculus, and noted that the kernel arising by normalization with regard to the so-

$$
\begin{array}{ccccccc}
\lambda_v & \rightarrow & \lambda_{ml} & \rightarrow & \lambda_{lin} & \rightarrow & \lambda_v \\
\\
\lambda_c & & & & & & \\
\cup & & & & & & \\
\lambda_{c*} & \cong & \lambda_{ml*} & \cong & \lambda_{lin*} & & \\
\cup & & \cup & & \cup & & \\
\lambda_{c**} & \cong & \lambda_{ml**} & \cong & \lambda_{lin**} & \cong & \lambda_{cps}
\end{array}
$$

Figure 1: Summary of results

called A reductions is closely related to the target of the CPS translation. Some practical ramifications of this were explored by Flanagan, Sabry, Duba, and Felleisen (1993).

Lawall and Danvy (1993) make much of relating forward and inverse translations via two Galois connections and an isomorphism. Their Galois connections are based on artificial orders, and as they note their isomorphism does not preserve these orders. Our choice of order solves these problems and is more natural: we use the usual reduction relation on terms.

Administrative reductions play a central role in many CPS translations, from Plotkin's work onwards. Recent work explains administrative reductions of the target in terms of reductions in the source (Sabry and Felleisen 1993, Flanagan *et al.* 1993, Hatcliff and Danvy 1994). Each of our translations will incorporate administrative reductions, and each of our reflections will have a kernel that can be explained as source terms reduced to administrative normal form.

This article is a revised and extended version of a conference paper with the same title (Sabry and Wadler 1996). The material on the linear calculus and on the Fischer CPS translation is new.

Outline. The remainder of this paper is structured as follows. Section 2 summarizes our results. Section 3 introduces Galois connections and reflections, and explains why they embody a property every compiler writer seeks. Section 4 reviews the traditional translation of λ_v into λ_{ml} and reviews why it fails to be a reflection. Section 5 shows that there is a reflection between λ_c and a variant λ_{ml*} of λ_{ml}. Section 6 factors this reflection through an intermediate calculus λ_{c*} corresponding to the isomorphic image of λ_{ml*} in λ_c. Section 7 extends the results to a translation into linear logic, and Section 8 deals with translations into CPS. Section 9 observes that the variant CPS translation due to Fischer cannot be a reflection. Section 10 describes related work. Section 11 concludes.

2 Summary

If you like to see a summary in advance, read this section now; if you like to see a summary on completion, save it until the end. Depending on your preference, you may thereby read this section once, twice, or never.

Figure 1 illustrates a road-map of the terrain we cover. First we consider the traditional monad translation from λ_v into λ_{ml}. This translation is not a reflection; but expanding λ_v into λ_c, shrinking λ_{ml} into λ_{ml*}, and fine-tuning the translation, finally yields a reflection in λ_c of λ_{ml*}. The existence of a reflection guarantees that there is a kernel λ_{c*} of λ_c that is isomorphic to λ_{ml*}. Calculus λ_c has seven reduction rules, and λ_{c*} arises by normalizing with respect to two of these rules, $(let.1)$ and $(let.2)$.

Second we consider the translation from λ_v to a linear calculus λ_{lin}. The translation has essentially the same properties as the previous monad translation with one exception. The calculus λ_{lin*} is not a restriction of λ_{lin} but includes additional (η)-like reductions which were absent from the original linear calculus. As before making the translation into a reflection requires expanding λ_v into λ_c and guarantees that λ_{lin*} is isomorphic to the kernel computational calculus λ_{c*}.

Finally we consider the traditional CPS translation from λ_v back into λ_v. Again, this translation is not a reflection; but expanding (source) λ_v into λ_c, shrinking (target) λ_v into λ_{cps}, and fine-tuning the translation yields a reflection in λ_c of λ_{cps}. Again, the existence of a reflection guarantees that there is a kernel λ_{c**} of λ_c that is isomorphic to λ_{cps}. Calculus λ_{c**} arises by normalizing λ_{c*} with regard to one further reduction rule, $(assoc)$. Furthermore, the CPS translation factors through both the monad translation and the linear translation; and so there is also a kernel λ_{ml**} of λ_{ml*} and a kernel λ_{lin**} of λ_{lin*} that is also isomorphic to λ_{cps}.

3 Galois Connections and Reflections

Let's review the standard results about Galois connections and reflections (Mac Lane 1971, Davey and Priestley 1990). The standard results need to be adapted slightly, as reduction is a preorder (it is reflexive and transitive) but not a partial order (it is not anti-symmetric).

We write $\longrightarrow$ for a single-step of reduction; $\twoheadrightarrow$ for the reflexive and transitive closure of reduction; $=$ for the reflexive, transitive, and symmetric closure of reduction; and $\equiv$ for syntactic identity up to renaming.

Assume a source calculus S with reduction relation $\twoheadrightarrow_S$, a target calculus T with reduction relation $\twoheadrightarrow_T$. Reductions are directed in such a way that they naturally correspond to evaluation steps or optimizations. Let the maps $* : S \to T$ and $\# : T \to S$ correspond to compiling and decompiling, respectively. Finally, let M, N range over terms of S, and P, Q range over terms of T.

Definition 3.1 *Maps $*$ and $\#$ form a* Galois connection *from S to T whenever*

$$M \twoheadrightarrow_S P^{\#} \text{ if and only if } M^* \twoheadrightarrow_T P.$$

There is an alternative characterization of a Galois connection.

Proposition 3.1 *Maps $*$ and $\#$ form a Galois connection from S to T if and only if the following four conditions hold.*

$$\begin{array}{ll}
\text{(i)} & M \twoheadrightarrow_S M^{*\#}, \\
\text{(ii)} & P^{\#*} \twoheadrightarrow_T P, \\
\text{(iii)} & M \twoheadrightarrow_S N \text{ implies } M^* \twoheadrightarrow_T N^*, \\
\text{(iv)} & P \twoheadrightarrow_T Q \text{ implies } P^{\#} \twoheadrightarrow_S Q^{\#}.
\end{array}$$

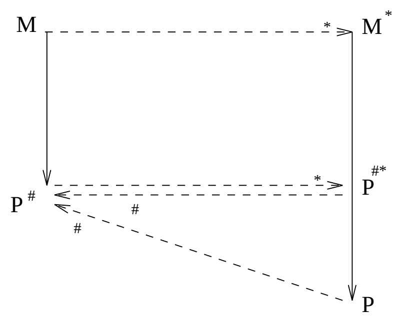

Figure 2: Motivating reflections

If the same four conditions hold with $\twoheadrightarrow_S$ and $\twoheadrightarrow_T$ replaced by $=_S$ and $=_T$, then one has an *equational correspondence*, as defined by Sabry and Felleisen (1993). Hence, every Galois connection implies an equational correspondence, though not conversely.

Our motivation for studying reflections is diagrammed in Figure 2. Consider a source term M compiling to a target term M^*, and consider an optimization $M^* \twoheadrightarrow_T P$ in the target. A Galois connection guarantees the existence of a corresponding optimization $M \twoheadrightarrow_S P^\#$ in the source. Recompiling this yields a reduction $M^* \twoheadrightarrow_T P^{\#*}$ in the target. If this optimisation is to be at least as good as the original optimisation, we require that $P \twoheadrightarrow_T P^{\#*}$. But by Proposition 3.1, for any Galois connection we have $P^{\#*} \twoheadrightarrow_T P$.

We therefore reasonably insist on having not just a Galois connection, but a *reflection*.

Definition 3.2 *Maps $*$ and $\#$ form a* reflection *in S of T if they form a Galois connection and $P \equiv P^{*\#}$.*

For a reflection, $\#$ is necessarily injective.

Galois connections and reflections compose.

Proposition 3.2 *Let $*_1$ and $\#_1$ form a Galois connection (reflection) from S to T, and $*_2$ and $\#_2$ form a Galois connection (reflection) from T to U. Then $*_1*_2$ and $\#_2\#_1$ form a Galois connection (reflection) from S to U.*

Every reflection factors into an *inclusion* and an *order isomorphism*. Write $S^{*\#}$ for the subset of S containing just those terms of the form $M^{*\#}$, and write $Id : S^{*\#} \to S$ for the trivial inclusion function. A reflection $*$ and $\#$ in S of T is an *inclusion* if $\#$ is the identity (and hence $S \supseteq T$), and is an *order isomorphism* if $*$ and $\#$ are inverses (and hence $S \cong T$).

Proposition 3.3 *Let $*$ and $\#$ form a reflection in S of T.*

1. *Translations $*\#$ and Id form an inclusion in S of $S^{*\#}$.*

2. *Translations $*$ and $\#$ form an order isomorphism between $S^{*\#}$ and T.*

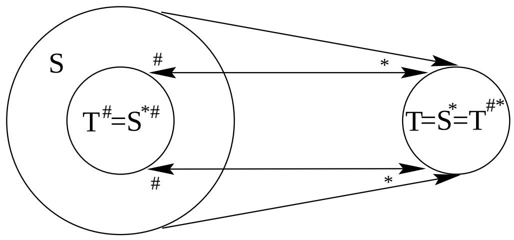

Figure 3: A reflection and its kernel isomorphism

The composition of the inclusion and the isomorphism is the original reflection.

The proposition is illustrated in Figure 3, which shows calculi S and T, and the images S^*, $T^\#$, $S^{*\#}$, and $T^{\#*}$ of these calculi under maps $*$ and $\#$. The kernel $S^{*\#}$ of S is isomorphic to T.

The summary illustrated in Figure 1 demonstrates repeated use of this proposition. Each reflection is factored into an inclusion (shown vertically) and an order isomorphism (shown horizontally).

For a Galois connection, one normally has two additional results, that $M^* \equiv M^{*\#*}$ and $P^\# \equiv P^{\#*\#}$. These proofs depend on anti-symmetry, and hence do not apply here. (As a counter-example to anti-symmetry, we have $YI \longrightarrow I(YI) \longrightarrow YI$ but $YI \not\equiv I(YI)$, where I is the identity and Y is the fixpoint combinator. Note that anti-symmetry can only fail for terms with no normal form.) In any event, both equivalences do follow from the stronger property of being a reflection.

4 Monads: The Problem

Let's review the traditional translation from the call-by-value calculus into the monad calculus, and see why it fails to be a reflection.

Plotkin's call-by-value calculus λ_v is summarized in Figure 4. We let x, y, z range over variables, L, M, N range over terms, and V, W range over values. A term is either a value or an application, and a value is either a variable or an abstraction. The call-by-value nature of the calculus is expressed by limiting the argument to a value in $(\beta.v)$, and by limiting the function to a value in $(\eta.v)$. An important aspect of each calculus we deal with is that it is confluent.

Proposition 4.1 *The reductions of λ_v are confluent.*

Moggi's monadic meta-language λ_{ml} is summarized in Figure 5. This calculus distinguishes *values* from *computations*. Functions may accept and return either values or computations, and are defined by the call-by-name rules (β) and (η). Two terms relate

$$
\begin{array}{llll}
\text{terms} & L, M, N & ::= & V \mid LM \\
\text{values} & V, W & ::= & x \mid \lambda x.\, N
\end{array}
$$

$$
\begin{array}{llcl}
(\beta.v) & (\lambda x.\, N)V & \longrightarrow & N[x := V] \\
(\eta.v) & \lambda x.\,(V x) & \longrightarrow & V, \quad \text{if } x \notin fv(V)
\end{array}
$$

Figure 4: The call-by-value calculus, λ_v

$$
\text{terms} \quad L, M, N \quad ::= \quad x \mid \lambda x.\, M \mid MN \mid \texttt{[}M\texttt{]} \mid \texttt{let } x \Leftarrow M \texttt{ in } N
$$

$$
\begin{array}{llcll}
(\beta) & (\lambda x.\, N)M & \longrightarrow & N[x := M] & \\
(\eta) & \lambda x.\,(M x) & \longrightarrow & M, & \text{if } x \notin fv(M) \\
(\beta.let) & \texttt{let } x \Leftarrow \texttt{[}M\texttt{] in } N & \longrightarrow & N[x := M] & \\
(\eta.let) & \texttt{let } x \Leftarrow M \texttt{ in [}x\texttt{]} & \longrightarrow & M, & \text{if } x \notin fv(M) \\
(assoc) & \texttt{let } y \Leftarrow (\texttt{let } x \Leftarrow L \texttt{ in } M) \texttt{ in } N & \longrightarrow & \texttt{let } x \Leftarrow L \texttt{ in } (\texttt{let } y \Leftarrow M \texttt{ in } N) &
\end{array}
$$

Figure 5: The monadic calculus, λ_{ml}

$$
* : \lambda_v \to \lambda_{ml}
$$

$$
\begin{array}{lcl}
V^* & \equiv & \texttt{[}V^\dagger\texttt{]} \\
(LM)^* & \equiv & \texttt{let } x \Leftarrow L^* \texttt{ in let } y \Leftarrow M^* \texttt{ in } xy \\
(x)^\dagger & \equiv & x \\
(\lambda x.\, N)^\dagger & \equiv & \lambda x.\, N^*
\end{array}
$$

Figure 6: Translation of λ_v into λ_{ml}

values to computations: the term $\texttt{[}M\texttt{]}$ denotes the computation that does nothing save return the value M; and the term $\texttt{let } x \Leftarrow L \texttt{ in } N$ denotes the computation that performs computation L, binds x to the resulting value, and then performs computation N. The interaction of these terms is described by the three rules $(\beta.let)$, $(\eta.let)$, and $(assoc)$.

Proposition 4.2 *The reductions of λ_{ml} are confluent.*

The translation $*$ from λ_v to λ_{ml} is described in Figure 6. Translation $*$ on terms uses an auxiliary translation $\dagger$ on values. The two translations are related by a substitution lemma, $N^*[x := V^\dagger] \equiv (N[x := V])^*$, which is easily checked by induction over the structure of terms.

The translation $*$ is *sound* in that $M \twoheadrightarrow_v N$ implies $M^* \twoheadrightarrow_{ml} N^*$. This is easily checked by looking at the translation of $(\beta.v)$,

$$
\begin{array}{lll}
((\lambda x.\, N)V)^* & \equiv & \texttt{let}\ y \Leftarrow [\lambda x.\, N^*]\ \texttt{in}\ \texttt{let}\ z \Leftarrow [V^\dagger]\ \texttt{in}\ yz \\
& \twoheadrightarrow_{(\beta.let)} & (\lambda x.\, N^*)V^\dagger \\
& \longrightarrow_{(\beta)} & N^*[x := V^\dagger] \\
& \equiv & (N[x := V])^*,
\end{array}
$$

and similarly for $(\eta.v)$.

However, the translation is not *complete* even in the weak sense required by equational correspondence: $M^* =_{ml} N^*$ does not imply $M =_v N$. For example, it is easy to check that

$$((\lambda x.\, xM)L)^* \quad =_{ml} \quad (LM)^*$$

while if L is not a value then the equivalence does not hold in the call-by-value calculus.

Proposition 4.3 *The translation $* : \lambda_v \to \lambda_{ml}$ is sound, but does not generate an equational correspondence.*

5 Monads: The Solution

To refine the translation $* : \lambda_v \to \lambda_{ml}$ into a reflection requires three steps: first, grow λ_v into λ_c; second, shrink λ_{ml} into λ_{ml*}; third, fine tune the translation $*$.

Step one grows λ_v into λ_c, which is summarized in Figure 7. The new calculus was carefully designed by Moggi to model directly the effect of translation into λ_{ml}. This is achieved by adding to λ_c a term $\texttt{let}\ x = M\ \texttt{in}\ N$ which mimics the term $\texttt{let}\ x \Leftarrow M\ \texttt{in}\ N$ of λ_{ml}; by adding to λ_c reductions corresponding to each reduction of λ_{ml}; and by adding to λ_c two more reductions, $(let.1)$ and $(let.2)$, which mimic the effect of the translation from λ_v into λ_{ml}. Let P, Q range over non-values. Rules $(let.1)$ and $(let.2)$ are restricted to act on non-values, since values yield a reduction in the opposite direction via $(\beta.let)$.

You may wonder: Is the new form $(\texttt{let}\ x = M\ \texttt{in}\ N)$ necessary, or could it be represented by $(\lambda x.\, N)M$ instead? The latter is possible, but then the rules $(\beta.let)$, $(\eta.let)$, $(assoc)$, $(let.1)$, and $(let.2)$ all become more difficult to read. Further, we prefer for historical reasons to stick to Moggi's formulation.

You may also wonder: Do the rules $(let.1)$ and $(let.2)$ point in the right direction? They do: the direction is dictated by the desire that reduction be confluent. For example, reversing $(let.1)$, we have $P \equiv (\texttt{let}\ y = (\texttt{let}\ x = L\ \texttt{in}\ M)\ \texttt{in}\ yN)$ reduces to $((\texttt{let}\ x = L\ \texttt{in}\ M)N)$. We also have:

$$P \longrightarrow_{(assoc)} \texttt{let}\ x = L\ \texttt{in}\ (\texttt{let}\ y = M\ \texttt{in}\ yN) \longrightarrow_{(rev.let.1)} \texttt{let}\ x = L\ \texttt{in}\ MN$$

which shows that confluence fails if $(let.1)$ is reversed. The system as given is confluent, as was shown by Moggi (1988).

Proposition 5.1 *The reductions of λ_c are confluent.*

Step two shrinks λ_{ml} into λ_{ml*}, which is summarized in Figure 8. The grammar is the smallest one that contains all terms in the image of the translation $* : \lambda_v \to \lambda_{ml}$ of the

$$
\begin{array}{llll}
\text{terms} & L, M, N & ::= & V \mid P \\
\text{values} & V, W & ::= & x \mid \lambda x.\, N \\
\text{non-values} & P, Q & ::= & LM \mid \texttt{let}\ x = M\ \texttt{in}\ N
\end{array}
$$

$$
\begin{array}{llcll}
(\beta.v) & (\lambda x.\, N)V & \longrightarrow & N[x := V] & \\
(\eta.v) & \lambda x.\, (V x) & \longrightarrow & V, & \text{if } x \notin fv(V) \\
(\beta.let) & \texttt{let}\ x = V\ \texttt{in}\ N & \longrightarrow & N[x := V] & \\
(\eta.let) & \texttt{let}\ x = M\ \texttt{in}\ x & \longrightarrow & M, & \text{if } x \notin fv(M) \\
(assoc) & \texttt{let}\ y = (\texttt{let}\ x = L\ \texttt{in}\ M)\ \texttt{in}\ N & \longrightarrow & \texttt{let}\ x = L\ \texttt{in}\ (\texttt{let}\ y = M\ \texttt{in}\ N) & \\
(let.1) & PM & \longrightarrow & \texttt{let}\ x = P\ \texttt{in}\ xM & \\
(let.2) & VQ & \longrightarrow & \texttt{let}\ y = Q\ \texttt{in}\ Vy &
\end{array}
$$

Figure 7: The computational calculus, λ_c

$$
\begin{array}{llll}
\text{terms} & L, M, N & ::= & \texttt{[}V\texttt{]} \mid P \\
\text{values} & V, W & ::= & x \mid \lambda x.\, N \\
\text{non-values} & P, Q & ::= & VW \mid \texttt{let}\ x \Leftarrow M\ \texttt{in}\ N
\end{array}
$$

$$
\begin{array}{llcll}
(\beta.v) & (\lambda x.\, N)V & \longrightarrow & N[x := V] & \\
(\eta.v) & \lambda x.\, (V x) & \longrightarrow & V, & \text{if } x \notin fv(V) \\
(\beta.let) & \texttt{let}\ x \Leftarrow \texttt{[}V\texttt{]}\ \texttt{in}\ N & \longrightarrow & N[x := V] & \\
(\eta.let) & \texttt{let}\ x \Leftarrow M\ \texttt{in}\ \texttt{[}x\texttt{]} & \longrightarrow & M, & \text{if } x \notin fv(M) \\
(assoc) & \texttt{let}\ y \Leftarrow (\texttt{let}\ x \Leftarrow L\ \texttt{in}\ M)\ \texttt{in}\ N & \longrightarrow & \texttt{let}\ x \Leftarrow L\ \texttt{in}\ (\texttt{let}\ y \Leftarrow M\ \texttt{in}\ N) &
\end{array}
$$

Figure 8: The simplified monadic calculus, λ_{ml*}

$$
* : \lambda_c \to \lambda_{ml*}
$$

$$
\begin{array}{lcl}
V^* & \equiv & \texttt{[}V^\dagger\texttt{]} \\
(PM)^* & \equiv & \texttt{let}\ x \Leftarrow P^*\ \texttt{in}\ (xM)^* \\
(VQ)^* & \equiv & \texttt{let}\ y \Leftarrow Q^*\ \texttt{in}\ (Vy)^* \\
(VW)^* & \equiv & V^\dagger W^\dagger \\
(\texttt{let}\ x = M\ \texttt{in}\ N)^* & \equiv & \texttt{let}\ x \Leftarrow M^*\ \texttt{in}\ N^* \\
(x)^\dagger & \equiv & x \\
(\lambda x.\, M)^\dagger & \equiv & \lambda x.\, M^*
\end{array}
$$

$$
\# : \lambda_{ml*} \to \lambda_c
$$

$$
\begin{array}{lcl}
x^\# & \equiv & x \\
(\lambda x.\, M)^\# & \equiv & \lambda x.\, M^\# \\
(VW)^\# & \equiv & V^\# W^\# \\
(\texttt{[}V\texttt{]})^\# & \equiv & V^\# \\
(\texttt{let}\ x \Leftarrow M\ \texttt{in}\ N)^\# & \equiv & \texttt{let}\ x = M^\#\ \texttt{in}\ N^\#
\end{array}
$$

Figure 9: Reflection in λ_c of λ_{ml*}

previous section, and that is closed under the reductions of λ_{ml}. The new grammar differs from the old in two key ways: applications MN in λ_{ml} are restricted to the form VW in λ_{ml*}, and computations $[M]$ in λ_{ml} are restricted to the form $[V]$ in λ_{ml*}.

The reduction rules are restricted accordingly. Since all applications have the form VW, rules (β) and (η) and rules $(\beta.v)$ and $(\eta.v)$ have the same effect on the calculus. This provides an analogue of Plotkin's *indifference* property, which states that call-by-value and call-by-name evaluation yield the same result for terms in CPS.

Proposition 5.2 *The grammar of λ_{ml*} is closed under the reductions of λ_{ml}. That is, if $M \longrightarrow N$ in λ_{ml} and M is in λ_{ml*}, then N is also in λ_{ml*}, and $M \longrightarrow N$ in λ_{ml*}.*

The proof is an easy case analysis. It follows as a corollary from Proposition 4.2 that the reductions of λ_{ml*} are confluent.

Step three adapts the translation $* : \lambda_v \to \lambda_{ml}$ to the new calculi. The straightforward choice is to leave the translation as is, adding a line for `let`.

$$\begin{array}{lcl}
V^* & \equiv & [V^\dagger] \\
(LM)^* & \equiv & \overline{\mathtt{let}}\ x \Leftarrow L^* \\
 & & \mathtt{in}\ \overline{\mathtt{let}}\ y \Leftarrow M^* \\
 & & \mathtt{in}\ xy \\
(\mathtt{let}\ x = M\ \mathtt{in}\ N)^* & \equiv & \mathtt{let}\ x \Leftarrow M^*\ \mathtt{in}\ N^*
\end{array}$$

Here $\dagger$ is as in Figure 6. The meaning of the overbars will be explained shortly.

Alas, this translation is not even sound in our stronger sense: it preserves the equalities of λ_c but not reductions. The key problem is with rule $(let.2)$, which requires $(\beta.let)$ reductions in both directions.

$$\begin{array}{lll}
(VQ)^* & \equiv & \mathtt{let}\ x \Leftarrow [V^\dagger]\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow Q^*\ \mathtt{in}\ xy \\
 & \longrightarrow_{\beta.let} & \mathtt{let}\ y \Leftarrow Q^*\ \mathtt{in}\ V^\dagger y \\
 & \twoheadleftarrow_{\beta.let} & \mathtt{let}\ y \Leftarrow Q^*\ \mathtt{in}\ \mathtt{let}\ x' \Leftarrow [V^\dagger]\ \mathtt{in}\ \mathtt{let}\ y' \Leftarrow [y]\ \mathtt{in}\ x'y' \\
 & \equiv & (\mathtt{let}\ y = Q\ \mathtt{in}\ Vy)^*
\end{array}$$

The solution is to consider the $(\beta.let)$ reductions as part of the translation. The two overlined occurrences of `let` introduced in the translation of applications are regarded as *administrative* occurrences. A second stage is added to the translation where all administrative $(\beta.let)$ redexes (that is, ones where the relevant `let` is overlined) are reduced.

This somewhat awkward description as a two-stage translation with administrative redexes may be replaced by the equivalent translation given in the first half of Figure 9. This was derived by a simple case analysis on applications, with according simplification. The obvious homomorphism, shown in the second half of the figure, serves as the inverse translation. It is now straightforward to verify that these constitute a reflection.

Proposition 5.3 *Translations $*$ and $\#$ form a reflection in λ_c of λ_{ml*}.*

To prove this we verify separately each of the four parts of Proposition 3.1. Parts (i) and (ii) are verified by induction over the structure of terms, and parts (iii) and (iv) are verified by induction over the structure of reductions.

6 Monads: The Factorization

Proposition 3.3 guarantees that there must be an isomorphic image of λ_{ml*} within λ_c. It consists of exactly those terms of the form $M^{*\#}$. We name this calculus λ_{c*}, and it is summarized in Figure 10. The grammar is identical to λ_c, except general applications MN are replaced by value applications VW, much as in the move from λ_{ml} to λ_{ml*}.

terms	L, M, N	$::=$	$V \mid P$
values	V, W	$::=$	$x \mid \lambda x.\, N$
non-values	P, Q	$::=$	$VW \mid \mathtt{let}\ x = M\ \mathtt{in}\ N$

Reductions $(\beta.v), (\eta.v), (\beta.let), (\eta.let), (assoc)$, as in Figure 7

Figure 10: The kernel computational calculus, λ_{c*}

$*_1 : \lambda_c \to \lambda_{c*}$

$$
\begin{array}{lcl}
V^* & \equiv & V^\dagger \\
(PM)^* & \equiv & \mathtt{let}\ x = P^*\ \mathtt{in}\ (xM)^* \\
(VQ)^* & \equiv & \mathtt{let}\ y = Q^*\ \mathtt{in}\ (Vy)^* \\
(VW)^* & \equiv & V^\dagger W^\dagger \\
(\mathtt{let}\ x = M\ \mathtt{in}\ N)^* & \equiv & \mathtt{let}\ x = M^*\ \mathtt{in}\ N^* \\
x^\dagger & \equiv & x \\
(\lambda x.\, M)^\dagger & \equiv & \lambda x.\, M^*
\end{array}
$$

$\#_1 : \lambda_{c*} \to \lambda_c$

the trivial inclusion

Figure 11: Inclusion in λ_c of λ_{c*}

$*_2 : \lambda_{c*} \to \lambda_{ml*}$

$$
\begin{array}{lcl}
V^* & \equiv & [V^\dagger] \\
(VW)^* & \equiv & V^\dagger W^\dagger \\
(\mathtt{let}\ x = M\ \mathtt{in}\ N)^* & \equiv & \mathtt{let}\ x \Leftarrow M^*\ \mathtt{in}\ N^* \\
x^\dagger & \equiv & x \\
(\lambda x.\, M)^\dagger & \equiv & \lambda x.\, M^*
\end{array}
$$

$\#_2 : \lambda_{ml*} \to \lambda_{c*}$

same as $\#$ from Figure 9

Figure 12: Isomorphism of λ_{c*} and λ_{ml*}

The terms of λ_{c*} can be characterized as the terms of λ_c in $(let.1)$ and $(let.2)$ normal form. Once a term has been normalized with respect to these two rules, they are never required again — that is, none of the other rules will introduce a $(let.1)$ or $(let.2)$ redex.

As guaranteed by Proposition 3.3, the reflection $* : \lambda_c \to \lambda_{ml*}$ factors into an inclusion $*_1 : \lambda_c \to \lambda_{c*}$ and an order isomorphism $*_2 : \lambda_{c*} \to \lambda_{ml*}$, given in Figures 11 and 12. The proposition even shows how to compute these: $*_1$ is $*\#$, $\#_1$ is the identity, $*_2$ is a restriction of $*$, and $\#_2$ is $\#$. What is a pleasant bonus, not guaranteed by the proposition, is that

$*_1$ has a simple interpretation: it reduces a term of λ_c to $(let.1)$ and $(let.2)$ normal form, hence yielding a term of λ_{c*}.

7 Linear Calculus

A linear lambda calculus similar to that studied by Wadler (1993) and Maraist *et al.* (1995) is summarized in Figure 13. The calculus contains two sorts of variables: a, b, c range over *linear* variables which are used exactly once in a term, while x, y, z range over *classical* variables which may appear any number of times. Introduction and elimination of linear implication $\multimap$ corresponds, respectively, to abstraction over linear variables $\lambda a.\, N$ and application LM. Introduction and elimination of the modality ! corresponds to the term forms $!M$ and $\mathtt{let}\ !x = L\ \mathtt{in}\ N$. There are two important side conditions on the grammar: each linear variable is used exactly once, and no linear variable appears free in a term of the form $!M$.

$$\text{terms} \qquad L, M, N \quad ::= \quad a \mid x \mid \lambda a.\, N \mid LM \mid !M \mid \mathtt{let}\ !x = L\ \mathtt{in}\ N$$

$$\begin{array}{lllll}
(\beta.\multimap) & (\lambda a.\, N)M & \longrightarrow & N[a := M] \\
(\beta.!) & \mathtt{let}\ !x = !M\ \mathtt{in}\ N & \longrightarrow & N[x := M] \\
(!\multimap) & (\mathtt{let}\ !x = L\ \mathtt{in}\ M)N & \longrightarrow & \mathtt{let}\ !x = L\ \mathtt{in}\ (MN) \\
(!!) & \mathtt{let}\ !y = (\mathtt{let}\ !x = L\ \mathtt{in}\ M)\ \mathtt{in}\ N & \longrightarrow & \mathtt{let}\ !x = L\ \mathtt{in}\ (\mathtt{let}\ !y = M\ \mathtt{in}\ N)
\end{array}$$

Figure 13: The linear calculus, λ_{lin}

$$* : \lambda_v \longrightarrow \lambda_{lin}$$

$$\begin{array}{lcl}
V^* & \equiv & !V^\dagger \\
(LM)^* & \equiv & (\mathtt{let}\ !x = M^*\ \mathtt{in}\ x)N^* \\
x^\dagger & \equiv & x \\
(\lambda x.\, N)^\dagger & \equiv & \lambda a.\, \mathtt{let}\ !x = a\ \mathtt{in}\ N^*
\end{array}$$

Figure 14: Translation of λ_v into λ_{lin}

The four reduction rules on terms of this calculus correspond to operations on proofs in linear logic. The rules $(\beta\multimap)$ and $(\beta!)$ correspond to simplification of proofs, while the rules $(!\multimap)$ and $(!!)$ correspond to commuting conversions. See Maraist *et al.* (1995) for details.

Informally, evaluation of the term $\mathtt{let}\ !x = L\ \mathtt{in}\ N$ proceeds by first evaluating L to a term of the form $!M$ and then substituting M for x in N. Thus we may view the term $!M$ as suspending evaluation and the term $\mathtt{let}\ !x = L\ \mathtt{in}\ N$ as forcing the corresponding evaluation. The requirement that no linear variable appears free in $!M$ ensures that the substitution of M for x does not violate the requirement that each linear variable is used exactly once.

The translation from λ_v to λ_{lin} of Maraist *et al.* (1995) is described in Figure 14. This translation maps $(\beta.v)$ reductions to reductions in the linear calculus:

$$\begin{array}{lll} ((\lambda x.\,N)V)^* & \equiv & (\mathtt{let}\ !z = !(\lambda a.\,\mathtt{let}\ !x = a\ \mathtt{in}\ N^*)\ \mathtt{in}\ z)\ (!V^\dagger) \\ & \twoheadrightarrow_{(\beta !)} & (\lambda a.\,\mathtt{let}\ !x = a\ \mathtt{in}\ N^*)\ (!V^\dagger) \\ & \twoheadrightarrow_{(\beta \multimap)} & \mathtt{let}\ !x = !V^\dagger\ \mathtt{in}\ N^* \\ & \twoheadrightarrow_{(\beta !)} & N^*[x := V^\dagger] \\ & \equiv & N[x := V]^* \end{array}$$

However this naïve linear translation fails to be a reflection for much the same reasons that the naïve monad translation failed, as described in Section 4. As in λ_{ml}, we can prove in λ_{lin} that

$$((\lambda x.\,xM)L)^* \quad =_{ml} \quad (LM)^*$$

$$\begin{array}{llll} \text{terms} & L, M, N & ::= & !V \mid P \\ \text{values} & V & ::= & x \mid \lambda a.\,\mathtt{let}\ !x = a\ \mathtt{in}\ M \\ \text{non}-\text{values} & P, Q & ::= & V(!V) \mid \mathtt{let}\ !x = M\ \mathtt{in}\ N \end{array}$$

$$\begin{array}{llll} (\beta.\multimap) & (\lambda a.\,\mathtt{let}\ !x = a\ \mathtt{in}\ M)\ (!V) & \longrightarrow & \mathtt{let}\ !x = !V\ \mathtt{in}\ M \\ (\eta.\multimap) & \lambda a.\,\mathtt{let}\ !x = a\ \mathtt{in}\ V(!x) & \longrightarrow & V, \quad \text{if } x \notin fv(V) \\ (\beta.!) & \mathtt{let}\ !x = !V\ \mathtt{in}\ N & \longrightarrow & N[x := V] \\ (\eta.!) & \mathtt{let}\ !x = M\ \mathtt{in}\ !x & \longrightarrow & M, \quad \text{if } x \notin fv(M) \\ (!!) & \mathtt{let}\ !y = (\mathtt{let}\ !x = L\ \mathtt{in}\ M)\ \mathtt{in}\ N & \longrightarrow & \mathtt{let}\ !x = L\ \mathtt{in}\ (\mathtt{let}\ !y = M\ \mathtt{in}\ N) \end{array}$$

Figure 15: The extended linear calculus, λ_{lin*}

$$* : \lambda_c \to \lambda_{lin*}$$

$$\begin{array}{lll} V^* & \equiv & !V^\dagger \\ (PM)^* & \equiv & \mathtt{let}\ !x = P^*\ \mathtt{in}\ (xM)^* \\ (VQ)^* & \equiv & \mathtt{let}\ !y = Q^*\ \mathtt{in}\ (V(!y))^* \\ (VW)^* & \equiv & V^\dagger(!W^\dagger) \\ (\mathtt{let}\ x = M\ \mathtt{in}\ N)^* & \equiv & \mathtt{let}\ !x = M^*\ \mathtt{in}\ N^* \\ x^\dagger & \equiv & x \\ (\lambda x.\,M)^\dagger & \equiv & \lambda y.\,\mathtt{let}\ !x = y\ \mathtt{in}\ M^* \end{array}$$

$$\# : \lambda_{lin*} \to \lambda_c$$

$$\begin{array}{lll} !x^\# & \equiv & x \\ (!(\lambda a.\,\mathtt{let}\ !x = a\ \mathtt{in}\ M))^\# & \equiv & \lambda x.\,M^\# \\ (V(!W))^\# & \equiv & V^\# W^\# \\ (\mathtt{let}\ !x = M\ \mathtt{in}\ N)^\# & \equiv & \mathtt{let}\ x = M^\#\ \mathtt{in}\ N^\# \end{array}$$

Figure 16: Reflection in λ_c of λ_{lin*}

which is not provable in the call-by-value calculus if L is not a value. Analogous to Section 5, the solution to this problem is to extend the source calculus from λ_v to λ_c, to contract the target calculus from λ_{lin} to λ_{lin*}, and to fine-tune the corresponding translation. The new target λ_{lin*} is shown in Figure 15, and the new translation is shown in Figure 16.

Maraist *et al.* (1995) are concerned with a similar problem, finding translations that are both sound and complete, and adopt a similar solution, expanding the source calculus from VAL to LET. One significant difference is that the source calculus in Maraist et al. contains no $(\eta.v)$ rule, and hence the target calculus λ_{lin} contains no analogous rules. We have remedied this problem by adding the rules $(\eta.\multimap)$ and $(\eta.!)$ to λ_{lin*}. Hence, although the reductions $(\beta.\multimap)$, $(\beta.!)$, and $(!!)$ of λ_{lin*} are restrictions of the corresponding reductions in λ_{lin}, the other two reductions of λ_{lin*} are unrelated to λ_{lin}.

The new reduction rule $(\eta.!)$ is exactly what one would expect, but the rule $(\eta.\multimap)$ is slightly disturbing. The rule one might expect is

$$(\lambda a.\, M a) \quad \longrightarrow \quad M, \quad \text{if } a \notin \mathit{fv}(M)$$

but $(\eta.\multimap)$ is not derivable from this rule, even in combination with the other rules. Nonetheless, $(\eta.\multimap)$ appears sound for the usual models of linear logic, and including it in λ_{lin*} seems to pose no problem. The utility of $(\eta.\multimap)$ suggests that finding a complete set of reduction rules for a linear calculus is a question of interest for the future.

Modulo this small hiccup, everything proceeds smoothly and by analogy with the development for monads. The calculus λ_{lin*} of Figure 15 is sensible.

Proposition 7.1 *The calculus λ_{lin*} is confluent and closed under reduction.*

The transformation $*$ into the linear calculus and its erasure $\#$, shown in Figure 16, form a reflection.

Proposition 7.2 *Translations $*$ and $\#$ form a reflection in λ_c of λ_{lin*}.*

We have as a corolary, from Proposition 3.3, that the kernel λ_{c*} is isomorphic to λ_{lin*}. The isomorphism is spelled out by the maps $*_2$ and $\#_2$ between λ_{c*} and λ_{lin}, shown in Figure 17. The confluence of λ_{lin*}, asserted above, follows immediately from the existence of this isomorphism.

$$*_2 : \lambda_{c*} \to \lambda_{lin*} \qquad\qquad \#_2 : \lambda_{lin*} \to \lambda_{c*}$$

$$\begin{array}{lcll}
V^* & \equiv & !V^\dagger & \text{same as } \# \text{ from Figure 16} \\
(V W)^* & \equiv & V^\dagger(!W^\dagger) & \\
(\texttt{let } x = M \texttt{ in } N)^* & \equiv & \texttt{let } !x = M^* \texttt{ in } N^* & \\
x^\dagger & \equiv & x & \\
(\lambda x.\, M)^\dagger & \equiv & \lambda a.\, \texttt{let } !x = a \texttt{ in } M^* &
\end{array}$$

Figure 17: Isomorphism of λ_{c*} and λ_{lin*}

8 Continuations

The development for continuations is paralleled by the development for monads and the linear calculus. Just as λ_v maps into λ_{ml} via the traditional call-by-value monad translation, so does λ_v map into λ_v via the traditional call-by-value CPS translation:

$$\begin{array}{lcl} V^* & \equiv & \lambda k.\, kV^\dagger \\ (LM)^* & \equiv & \lambda k.\, L^*(\lambda x.\, M^*(\lambda y.\, xyk)) \\ \\ x^\dagger & \equiv & x \\ (\lambda x.\, N)^\dagger & \equiv & \lambda x.\, N^* \end{array}$$

A remarkable property of this translation is that one may always choose k to be exactly the same name, without fear of name clash. Again, this translation is sound but not complete.

To refine the translation $* : \lambda_v \to \lambda_v$ into a reflection also requires three steps: first, grow the source λ_v into λ_c; second, shrink the target λ_v into λ_{cps}; third, fine tune the translation $*$. Step one was already accomplished as part of the monad development. Steps two and three are best considered in reverse order, as the calculus of step two should be the image of the modified translation of step three.

To refine the translation $* : \lambda_v \to \lambda_v$, it is suitably extended for `let`, and has some reductions labeled as administrative.

$$\begin{array}{lcl} V^* & \equiv & \overline{\lambda} k.\, kV^\dagger \\ (LM)^* & \equiv & \overline{\lambda} k.\, L^*(\overline{\lambda} x.\, M^*(\overline{\lambda} y.\, xyk)) \\ (\texttt{let}\ x = M\ \texttt{in}\ N)^* & \equiv & \overline{\lambda} k.\, M^*(\lambda x.\, N^* k) \\ x^\dagger & \equiv & x \\ (\lambda x.\, N)^\dagger & \equiv & \lambda x.\, N^* \end{array}$$

The translation now takes place in three stages. Stage one applies the translation proper. Stage two reduces any administrative $(\beta.v)$ redexes — that is, ones where the relevant λ is overlined. Stage three strips the leading 'λk', which always appears and so is redundant.

This three-stage translation may be replaced by the equivalent translation given in the first half of Figure 19. The old translation relates to the new as follows: $M^{*\text{old}} \equiv \lambda k.\, M^{*\text{new}}$, where k is the distinguished continuation variable. The auxiliary translation $M\!: K$ closely resembles a translation introduced by Plotkin (1975) to capture the effect of administrative reductions.

Step two shrinks the target λ_v into λ_{cps}, which is summarized in Figure 18. The grammar is the smallest one that is in the image of the refined translation $*$. An additional class of non-terminals, K, is added to the grammar, ranging over continuations. This class contains the distinguished free variable k, and contains those lambda expressions which may be substituted for k.

Examination reveals that each of the rules $(\beta.v)$ and $(\eta.v)$ arises in exactly two situations, yielding four rules in the target λ_{cps} corresponding to four of the seven rules in the source λ_c. It is easily verified that the grammar is indeed closed under reduction. Every application has a value as an argument, so call-by-value and call-by-name reductions have the same effect on the language, which explains Plotkin's indifference property.

$$
\begin{array}{llcl}
\text{terms} & L, M, N & ::= & KV \mid VWK \\
\text{values} & V, W & ::= & x \mid \lambda x.\, \lambda k.\, M \\
\text{continuations} & K & ::= & k \mid \lambda x.\, M
\end{array}
$$

$$
\begin{array}{llcl}
(\beta.v) & (\lambda x.\, \lambda k.\, M)VK & \longrightarrow & M[x := V][k := K] \\
(\eta.v) & \lambda x.\, \lambda k.\, Vxk & \longrightarrow & V, \quad \text{if } x \notin fv(V) \\
(\beta.let) & (\lambda x.\, M)V & \longrightarrow & M[x := V] \\
(\eta.let) & \lambda x.\, Kx & \longrightarrow & K, \quad \text{if } x \notin fv(K)
\end{array}
$$

Figure 18: Continuation-passing style calculus λ_{cps}

$$
* : \lambda_c \to \lambda_{cps}
$$

$$
\begin{array}{lcl}
M^* & \equiv & M : k \\
V : K & \equiv & KV^\dagger \\
(PM) : K & \equiv & P : (\lambda x.\, ((xM) : K)) \\
(VQ) : K & \equiv & Q : (\lambda y.\, ((Vy) : K)) \\
(VW) : K & \equiv & V^\dagger W^\dagger K \\
(\mathtt{let}\ x = M\ \mathtt{in}\ N) : K & \equiv & M : (\lambda x.\, (N : K)) \\
x^\dagger & \equiv & x \\
(\lambda x.\, M)^\dagger & \equiv & \lambda x.\, \lambda k.\, M^*
\end{array}
$$

$$
\# : \lambda_{cps} \to \lambda_c
$$

$$
\begin{array}{lcl}
(KV)^\# & \equiv & K^\flat[V^\natural] \\
(VWK)^\# & \equiv & K^\flat[V^\natural W^\natural] \\
x^\natural & \equiv & x \\
(\lambda x.\, \lambda k.\, M)^\natural & \equiv & \lambda x.\, M^\# \\
k^\flat & \equiv & [\,] \\
(\lambda x.\, N)^\flat & \equiv & \mathtt{let}\ x = [\,]\ \mathtt{in}\ N^\#
\end{array}
$$

Figure 19: Reflection in λ_c of λ_{cps}

Proposition 8.1 *The grammar of λ_{cps} is closed under the reductions of λ_v. That is, if $M \longrightarrow N$ in λ_v and M is in λ_{cps}, then N is also in λ_{cps}, and $M \longrightarrow N$ in λ_{cps}.*

The proof is an easy case analysis; it follows as a corollary that the reductions of λ_{cps} are confluent. Further, the same result holds if the call-by-value calculus λ_v is replaced by the call-by-name calculus λ_n.

The inverse translation # of Figure 19 is now easily derived. It has three parts, one for each component of the target grammar. A term M in λ_{cps} maps to a term $M^\#$ in λ_c; a value V in λ_{cps} maps to a value $V^\natural$ in λ_c; and a continuation K in λ_{cps} maps to an evaluation context $K^\flat$ in λ_c. An evaluation context C is a term with a hole $[\,]$, and if C is an evaluation context then $C[M]$ denotes the result of replacing the hole in C by the term M. The filling operation is straightforward since the holds of our evaluation contexts are never in the scope of a bound variable.

We can now verify that the maps of Figure 19 constitute a reflection.

Proposition 8.2 *Translations $*$ and $\#$ form a reflection in λ_c of λ_{cps}.*

As before, we prove each of the four parts of Proposition 3.1 separately. To prove part (i),

$$M \twoheadrightarrow M^{*\#}$$

requires we strengthen the inductive hypothesis to

$$K^\flat[M] \twoheadrightarrow (M : K)^\#.$$

Each proof is now straightforward, given the following two lemmas, which are equally straightforward.

Lemma 8.3 *Let M, V be in λ_c, and K be in λ_{cps}, then $(M : K)[x := V^\dagger] \equiv (M[x := V]) : K$.*

Lemma 8.4 *Let M, V, K be in λ_{cps}, then:*

$$K^\flat[M^\#[x := V^\natural]] \twoheadrightarrow (M[x := V][k := K])^\#.$$

This completes the parallel development of the reflection; there is also a parallel development of the factorization. Proposition 3.3 guarantees that there must be an isomorphic image of λ_{cps} within λ_c. We name this calculus λ_{c**}, and it is summarized in Figure 20. Just as the grammar of λ_{cps} has three components — terms, values, and continuations — so the grammar of λ_{c**} has three components — terms, values, and contexts. Observe that, despite the introduction of contexts, each term still possesses a unique decomposition in terms of the syntax. (For instance, the term xy corresponds to $K[VW]$, where K is $[\,]$, V is x, and W is y.)

The terms of λ_{c**} can be characterized as the terms of λ_c in $(let.1)$, $(let.2)$, and $(assoc)$ normal form. As in the previous development, once a term has been normalized, no further reductions ever introduce a $(let.1)$ or $(let.2)$ redex. Alas, the same cannot be said of $(assoc)$. The reduction $(\beta.v)$ may indeed introduce a further $(assoc)$ redex, as shown by the counterexample,

$$\begin{array}{rl} & \texttt{let } x = ((\lambda y.\, \texttt{let } z = ab \texttt{ in } c)\ \ d) \texttt{ in } e \\ \longrightarrow & \texttt{let } x = (\texttt{let } z = ab \texttt{ in } c) \texttt{ in } e. \end{array}$$

To gain insight into the problem, consider the corresponding CPS reduction,

$$\begin{array}{rl} & ((\lambda y.\, \lambda k.\, (a\ b\ (\lambda z.\, kc)))\ d\ (\lambda x.\, ke)) \\ \longrightarrow & (a\ b\ (\lambda z.\, (\lambda x.\, ke) c)). \end{array}$$

The image of this in λ_{c**} is

$$\begin{array}{rl} & \texttt{let } x = ((\lambda y.\, \texttt{let } z = ab \texttt{ in } c)\ \ d) \texttt{ in } e \\ \longrightarrow & \texttt{let } z = ab \texttt{ in let } x = c \texttt{ in } e. \end{array}$$

$$
\begin{array}{llll}
\text{terms} & L, M, N & ::= & K[V] \mid K[VW] \\
\text{values} & V, W & ::= & x \mid \lambda x.\, M \\
\text{contexts} & K & ::= & [\,] \mid \mathtt{let}\ x = [\,]\ \mathtt{in}\ M
\end{array}
$$

$$
\begin{array}{llll}
(\beta.v) & K[(\lambda x.\, M)V] & \longrightarrow & M[x := V] : K \\
(\eta.v) & \lambda x.\, (V x) & \longrightarrow & V,\ \text{if } x \notin fv(V) \\
(\beta.let) & \mathtt{let}\ x = V\ \mathtt{in}\ M & \longrightarrow & M[x := V] \\
(\eta.let) & \mathtt{let}\ x = [\,]\ \mathtt{in}\ K[x] & \longrightarrow & K, \text{if } x \notin fv(K)
\end{array}
$$

$$
\begin{array}{lll}
V : K & \equiv & K[V] \\
(VW) : K & \equiv & K[VW] \\
(\mathtt{let}\ x = V\ \mathtt{in}\ M) : K & \equiv & \mathtt{let}\ x = V\ \mathtt{in}\ (M : K) \\
(\mathtt{let}\ x = VW\ \mathtt{in}\ M) : K & \equiv & \mathtt{let}\ x = VW\ \mathtt{in}\ (M : K)
\end{array}
$$

Figure 20: The kernel computational calculus λ_{c**}

$$
\begin{array}{lll}
 & *_1 : \lambda_c \to \lambda_{c**} & \\
M^* & \equiv & M : [\,] \\
V : K & \equiv & K[V] \\
(PM) : K & \equiv & P : (\mathtt{let}\ x = [\,]\ \mathtt{in}\ ((xM) : K)) \\
(VQ) : K & \equiv & Q : (\mathtt{let}\ y = [\,]\ \mathtt{in}\ ((Vy) : K)) \\
(VW) : K & \equiv & K[V^\dagger W^\dagger] \\
(\mathtt{let}\ x = M\ \mathtt{in}\ N) : K & \equiv & M : (\mathtt{let}\ x = [\,]\ \mathtt{in}\ (N : K)) \\
x^\dagger & \equiv & x \\
(\lambda x.\, M)^\dagger & \equiv & \lambda x.\, M^*
\end{array}
$$

$\#_1 : \lambda_{c**} \to \lambda_c$

the trivial inclusion

Figure 21: Inclusion in λ_c of λ_{c**}

$$
\begin{array}{lll}
 & *_2 : \lambda_{c**} \to \lambda_{cps} & \\
(K[V])^* & \equiv & K^\ddagger V^\dagger \\
(K[VW])^* & \equiv & V^\dagger W^\dagger K^\ddagger \\
x^\dagger & \equiv & x \\
(\lambda x.\, M)^\dagger & \equiv & \lambda x.\, \lambda k.\, M^\dagger \\
[\,]^\ddagger & \equiv & k \\
(\mathtt{let}\ x = [\,]\ \mathtt{in}\ N)^\ddagger & \equiv & \lambda x.\, N^*
\end{array}
$$

$\#_2 : \lambda_{cps} \to \lambda_{c**}$

same as # from Figure 19

Figure 22: Isomorphism of λ_{c**} and λ_{cps}

where the right hand side is in (*assoc*)-normal form. The CPS language achieves this normalization using the meta-operation of substitution which traverses the CPS term to locate k and replace it by the continuation thus effectively "pushing" the continuation deep inside the term.

In order to properly match the behavior of CPS, we therefore add a corresponding meta-operation to λ_{c**}, $M : K$ shown in Figure 20. Using the new meta-operation we can extend the problematic source reduction $\beta.v$ with a built-in (*assoc*)-normalization action that mirrors the action of $\beta.v$ on CPS terms.

(An alternative to adding meta-notation for substitution may be to move to calculi that use explicit substitution, but we have not explored this possibility.)

There is one pitfall to avoid: since reductions apply to any subterm, one may be tempted to apply rule $(\beta.v)$ to the subterm $((\lambda y.\, \mathtt{let}\ z = ab\ \mathtt{in}\ c)\ d)$ in the counterexample by taking K to be $[\,]$ rather than $\mathtt{let}\ x = [\,]\ \mathtt{in}\ e$. But it is natural to insist that the reduction rule $(\beta.v)$ is only applied when K is determined according to the unique decomposition afforded by the grammar of λ_{c**}, and this steps neatly around the pitfall.

Again, as guaranteed by Proposition 3.3, the reflection $* : \lambda_c \to \lambda_{cps}$ factors into an inclusion $*_1 : \lambda_c \to \lambda_{c**}$ and an order isomorphism $*_2 : \lambda_{c**} \to \lambda_{cps}$, given in Figures 21 and 22. Again, these can be computed directly from the proposition, and again there is a bonus: $*_1$ has a simple interpretation as reducing a term of λ_c to its (*let*.1), (*let*.2), and (*assoc*) normal form.

In addition, the CPS translation factors through the monad translation. One may translate λ_{ml*} into λ_{cps} as follows.

$$
\begin{array}{lcl}
[V]^* & \equiv & \overline{\lambda} k.\, kV^\dagger \\
(VW)^* & \equiv & \overline{\lambda} k.\, V^\dagger W^\dagger k \\
(\mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ N)^* & \equiv & \overline{\lambda} k.\, M^*(\lambda x.\, N^* k) \\
x^\dagger & \equiv & x \\
(\lambda x.\, N)^\dagger & \equiv & \lambda x.\, N^*
\end{array}
$$

Regarding this as a two-stage translation with administrative reductions yields a reflection. As before, the reflection factors through a calculus λ_{ml**}, corresponding to the subset of λ_{ml*} consisting of terms in (*assoc*) normal form. The three calculi λ_{c**}, λ_{ml**}, and λ_{cps} are isomorphic.

In a similar way, the isomorphism between λ_{c*} and λ_{lin} of Figure 17 can be adapated to an isomorphism between λ_{c**} and a kernel of the linear calculus λ_{lin**} in (!!) normal form. Figure 1 diagrams the situation.

9 The Fischer translation

The CPS translation of Fischer (1972) differs from the CPS translation of Plotkin (1975) in that it swaps the order of function argument and continuation, allowing additional administrative reductions to be performed. Here is the Fischer translation, for comparison with the the Plotkin translation in Section 8.

$$\begin{array}{lcl} V^* & \equiv & \overline{\lambda}k.\, kV^\dagger \\ (LM)^* & \equiv & \overline{\lambda}k.\, L^*(\overline{\lambda}x.\, M^*(\overline{\lambda}y.\, xky)) \\ \\ x^\dagger & \equiv & x \\ (\lambda x.\, N)^\dagger & \equiv & \overline{\lambda}k.\, \lambda x.\, N^*k \end{array}$$

For example, the term $(\lambda x.\, x)y$ yields $(\lambda x.\, \lambda k.\, kx)yk$ under the Plotkin translation, and yields $(\lambda x.\, kx)y$ under the Fischer translation, where in each case one reduces all administrative $(\beta.v)$ redexes and strips the leading 'λk'.

An earlier version of this paper claimed that the Fischer CPS translation could be made a reflection, but gave no proof. Here we withdraw that claim. We show that the Fischer translation cannot be a reflection, and nor can it be a Galois connection with the source calculus λ_c. However, it may be a Galois connection with a different source calculus.

The Fischer translation cannot be a reflection. Thanks to administrative reduction, no term in the image of the Fischer translation will contain a 'λk' redex; but in order to be closed under reduction, the target language must contain such redexes. For example, take $M \equiv (\lambda f.\, fx)(\lambda y.\, y)$, so that $M^* \equiv (\lambda f.\, fkx)(\lambda k.\, \lambda y.\, ky)$. Then $M^* \longrightarrow P$, where $P \equiv (\lambda k.\, \lambda y.\, ky)kx$. But there is no source term with Fischer translation P, and so we cannot have $P^{\#*} \equiv P$. Note that this argument is independent of the reductions in the source language.

Further, the Fischer translation cannot be a Galois connection when the source calculus is λ_c. For example, take $M \equiv f((\lambda x.\, x)y)$ and $N \equiv (\lambda x.\, fx)y$. Both terms have the same Fischer translation, namely $M^* \equiv N^* \equiv P \equiv (\lambda x.\, fkx)y$. Since we have a Galois connection, we require that $M \twoheadrightarrow M^{*\#} \equiv P^\#$ and $N \twoheadrightarrow N^{*\#} \equiv P^\#$ in λ_c, and so fy is the only possible choice for $P^\#$. For a Galois connection we also require that $P^{\#*} \twoheadrightarrow P$, but this fails since $P^{\#*} \equiv fky \not\twoheadrightarrow (\lambda x.\, fkx)y \equiv P$.

We saw in Section 4 that a naïve translation with no administrative reductions fails to be a reflection. Here we see that the Fischer translation has too many administrative reductions to be a reflection. Just as Goldilock's porridge must be neither too hot nor too cold, administrative reductions must be neither too many nor too few.

Nonetheless, it may be possible to find a Galois connection based on the Fischer translation with a different choice of source calculus. We leave this as an open question.

10 Related Work

Plotkin (1975), among other contributions, formalizes the call-by-value CPS translation and shows that it preserves but does not reflect equalities: if $M = N$ in λ_v then $M^* = N^*$ in λ_v, but not conversely. Here and throughout, we write $*$ for the variant of the CPS translation under consideration, and hope this will lead to no confusion.

Sabry and Felleisen (1993) strengthen Plotkin's result by making the implication above reversible. They extend the call-by-value lambda calculus λ_v with a set of reductions X such that $M^* = N^*$ in $\lambda_v X$ if and only if $M^* = N^*$ in λ_v. As they note, $\lambda_v X$ and λ_c prove the same equalities; but they do not prove the same reductions.

Sabry and Felleisen introduce the notion of *equational correspondence* described in Section 3, and they prove that their translation constitutes such a correspondence. In fact, they prove something stronger, making their translation almost, but not quite, a Galois connection: their translations satisfies all four conditions of Proposition 3.1, except that condition (ii), $P^{\#*} \twoheadrightarrow P$, is replaced by the weaker $P^{\#*} = P$. (Compare this to the stronger $P^{\#*} \equiv P$ required of a reflection.)

They also single out a subset A of X, and observe that these A reductions correspond directly to the administrative reductions on CPS terms. Their terms in A normal form roughly correspond to our kernel calculus λ_{c**}, of terms in $(let.1)$, $(let.2)$, and $(assoc)$ normal form.

However, Sabry and Felleisen use Fischer's CPS translation, which we have seen cannot be a reflection. It remains an open question whether there is a variant of Sabry and Felleisen's work which yields a Galois connection.

Flanagan, Sabry, Duba, and Felleisen (1993) apply the results of Sabry and Felleisen. They suggest that CPS translation may not be so beneficial after all: it may be better to work directly in the source calculus. They show that terms in A normal form behave similarly to CPS terms, demonstrating this via a sequence of abstract machines. They also briefly sketch possible applications of the full set of reductions X. But they fail to observe our central point: that for optimization purposes one wants a result showing correspondence of reductions rather than correspondence of equations.

Lawall and Danvy (1993) give a factoring of CPS similar to the one described here, and also refer to Galois connections. They relate four languages: a source language, a kernel source language, a kernel target language, and a target language. The first two relate via a Galois connection, the middle two are isomorphic, and the last two again relate via a Galois connection. The first two parts of their factorization are similar to our inclusion in λ_c of λ_{c**}, and our order isomorphism from λ_{c**} to λ_{cps}. Their third step schedules evaluation, determining for each application whether the function or argument evaluates first. Here we use a language where the function always evaluates before the argument, and so we need no counterpart of their third step.

The Galois connections of Lawall and Danvy are based on an artificial ordering induced directly from the translations. One might argue that they are misusing the notion of Galois connection, and are instead dealing with the somewhat weaker notion of a pair of translations $*$ and $\#$ satisfying $M^{*\#*} \equiv_T M^*$ and $P^{\#*\#} \equiv_S P^\#$. As they note, their artificial ordering is unsatisfactory, because their middle isomorphism between the source kernel and target kernel does not respect this ordering: it is not an order isomorphism, and hence not a Galois connection. Thus, their factorization cannot be viewed as a composition of Galois connections. In contrast, we use a natural ordering relation, and our Galois connections do compose. Whereas their isomorphism *violates* the ordering of their Galois connection, our isomorphism arises as a *consequence* of our Galois connection, as shown by Proposition 3.3.

Hatcliff and Danvy (1994) consider translations analogous to our translations from λ_v to λ_{ml}, and from λ_{ml} to λ_{cps}. They also look at translations from other source languages into λ_{ml}, an issue we ignore. They show the translation from λ_v to λ_{ml} is sound; we give the stronger result that the translation from λ_c to λ_{ml} is a reflection. They also show that the translation from λ_{ml} to λ_{cps} is an equational correspondence; again, we give the stronger result that it is a reflection.

Maraist, Turner, Odersky, and Wadler (1995) consider translations into a linear lambda calculus. That paper extends the call-by-value calculus VAL to the calculus LET, this paper extends the call-by-value calculus λ_v to the calculus λ_c. The earlier paper stressed the anology of the linear translation with the CPS translation; as can be seen from the work here, an analogy of the linear translation with the monad translation is even more apposite.

11 Conclusion

This section describes a number of possible extensions of our results, and draws a conclusion about λ_c as a model of call-by-value.

Standard reduction. Most lambda calculi possess a notion of *standard reduction*, characterized by two properties. First, at most one standard reduction applies to a term. Second, if any sequence of reductions reduces a term to an answer, then the sequence of standard reductions will also do so. Hence standard reductions capture the behavior of an evaluator. Plotkin (1975) specified standard reductions for λ_v, and his results for CPS demonstrate not only that reductions are preserved, but also that standard reductions are preserved. (He expresses this in a different but equivalent form by saying that evaluation is preserved.) Hatcliff and Danvy (1994) give similar results for translation from Moggi's λ_{ml} into CPS. It appears straightforward to extend the work here by specifying a suitable notion of standard reduction for each of the calculi involved, and to show that the given translations are still reflections if one replaces reductions by standard reductions.

Call-by-value and call-by-need. Maraist, Turner, Odersky, and Wadler (1995) study a call-by-let calculus that is closely related to λ_c and that translates into linear logic. By extending the call-by-let calculus with just one law,

$$\texttt{let}\ x = M\ \texttt{in}\ N \quad \longrightarrow \quad N, \qquad \text{if } x \notin fv(N),$$

the authors derive a call-by-need calculus (Ariola *et al.* 1995) that translates into affine logic. We conjecture that when augmented with the above law, λ_c also yields a model of call-by-need.

A reflection on call-by-value. Plotkin's original paper on λ_v layed out two key properties of this calculus: first, it is adequate to describe evaluation, and second, it is inadequate to prove some equalities that we might reasonably expect to hold between terms. The first was demonstrated by a correspondence between λ_v and SECD machine of Landin (1964). The second was demonstrated by observing that there are terms that are not provably equal in λ_v, but whose translations into CPS are provably equal.

Moggi defined λ_c as an extension of λ_v that is sound and complete for all monad models, and hence proves a reasonably large set of equalities. He picked a confluent calculus to ease symbolic manipulation, but made no claims that λ_c was itself a reasonable model of computation. Sabry and Felleisen showed that λ_c proves two terms equal exactly when their CPS translations are equal. This reinforces the claim that λ_c yields a good theory of equality, but because they dealt only with equational correspondence, again says nothing about λ_v as a model of computation. Our results here relate λ_c reductions to reductions in λ_{ml} and λ_{cps}, both widely accepted as models of computation. We hereby put forward λ_c as a model of call-by-value computation that improves on λ_v.

References

[1] Appel, A. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Ariola, Z., Felleisen, M., Maraist, J., Odersky, M., and Wadler, P. A call-by-need lambda calculus. In *ACM Symposium on Principles of Programming Languages* (1995), pp. 233–246.

[3] Davey, B. A., and Priestley, H. A. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[4] Fischer, M. Lambda calculus schemata. In *ACM Conference on Proving Assertions about Programs* (1972), SIGPLAN Notices, bf 7, 1, pp. 104–109. Revised version in *Lisp and Symbolic Computation*, **6**, 3/4, (1993) 259-287.

[5] Flanagan, C., Sabry, A., Duba, B., and Felleisen, M. The essence of compiling with continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1993), pp. 237–247.

[6] Hatcliff, J., and Danvy, O. A generic account of continuation-passing styles. In *Proceedings of the 21th ACM Symposium on Principles of Programming Languages* (1994), pp. 458–471.

[7] Kranz, D., et al. Orbit: An optimizing compiler for Scheme. In *ACM SIGPLAN Symposium on Compiler Construction* (1986), SIGPLAN Notices, 21, 7, pp. 219–233.

[8] Landin, P. The mechanical evaluation of expressions. *Computer Journal 6*, 4 (1964), 308–320.

[9] Lawall, J., and Danvy, O. Separating stages in the continuation-passing transform. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages* (1993), pp. 124–136.

[10] Mac Lane, S. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[11] Maraist, J., et al. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. In *Conference on Mathematical Foundations of Programming Semantics* (1995).

[12] Moggi, E. Computational lambda-calculus and monads. Tech. Rep. ECS-LFCS-88-86, University of Edinburgh, 1988.

[13] Moggi, E. Computational lambda-calculus and monads. In *Proceedings of the Symposium on Logic in Computer Science* (1989), pp. 14–23.

[14] Moggi, E. Notions of computation and monads. *Information and Computation 93* (1991), 55–92.

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

[16] Reynolds, J. C. The discoveries of continuations. *Lisp and Symbolic Computation 6*, 3/4 (1993), 233–247.

[17] Sabry, A., and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation 6*, 3/4 (1993), 289–360.

[18] Sabry, A., and Wadler, P. A reflection on call-by-value. In *ACM SIGPLAN International Conference on Functional Programming* (1996), p. ?

[19] Steele, G. L. Rabbit: A compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, 1978.

[20] Wadler, P. A syntax for linear logic. In *Conference on Mathematical Foundations of Programming Semantics* (1993), Lecture Notes in Computer Science, 802, Springer Verlag.

[21] Wadler, P. A taste of linear logic. In *Mathematical Foundations of Computer Science* (1993), Lecture Notes in Computer Science, 711, Springer Verlag.