A Reflection on Call-by-Value

AMR SABRY
University of Oregon
and
PHILIP WADLER
Bell Laboratories, Lucent Technologies

One way to model a sound and complete translation from a source calculus into a target calculus is with an *adjoint* or a *Galois connection*. In the special case of a *reflection*, one also has that the target calculus is isomorphic to a subset of the source. We show that three widely studied translations form reflections. We use as our source language Moggi's computational lambda calculus, which is an extension of Plotkin's call-by-value calculus. We show that Plotkin's CPS translation, Moggi's monad translation, and Girard's translation to linear logic can all be regarded as reflections from this source language, and we put forward the computational lambda calculus as a model of call-by-value computation that improves on the traditional call-by-value calculus. Our work strengthens Plotkin's and Moggi's original results and improves on recent work based on *equational correspondence*, which uses equations rather than reductions.

---

1. INTRODUCTION

Compiler correctness is a central concern of computing. It is often expressed in the form of a diagram.

$$
\begin{array}{ccc}
M & \xrightarrow{\ \ S\ \ } & N^{\#} \\
{\scriptstyle *}\big\downarrow & & \big\uparrow{\scriptstyle \#} \\
M^{*} & \xrightarrow{\ \ T\ \ } & N
\end{array}
$$

Here the upper arrow indicates reduction in the source calculus S; the lower arrow indicates reduction in the target calculus T; the downward arrow indicates the compiling map $*$; and the upward arrow indicates the decompiling map $\#$.

The diagram states that compiling, evaluating in the target, and decompiling is equivalent to evaluating in the source. There are two forms this equivalence may take.

—*Soundness*. Every reduction in the source is valid in the target: If $M \longrightarrow\!\!\!\twoheadrightarrow_S N^\#$ then $M^* \longrightarrow\!\!\!\twoheadrightarrow_T N$.

—*Completeness*. Every reduction in the target is valid in the source: If $M^* \longrightarrow\!\!\!\twoheadrightarrow_T N$ then $M \longrightarrow\!\!\!\twoheadrightarrow_S N^\#$.

If both properties are present, one has what theorists call an *adjoint* or a *Galois connection*.

Sometimes one has a further pleasant property, that compiling is a left inverse to decompiling, $N^{\#*} \equiv N$ for every target term N, where $\equiv$ is syntactic identity (up to renaming). This special form of an adjoint is called a *reflection*. Whenever a reflection exists, the target is isomorphic to a subset of the source, which we call the *kernel*.

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 the continuation passing style (CPS) translation, also of Plotkin [1975]. Our second translation is into the monadic metalanguage, λ_{ml}, also of Moggi [1988]. The CPS translation arises as a special case of the monad translation, and factors through it. Our third translation is into a calculus based on the linear logic of Girard [1987]. Our linear calculus is closely related to formulations devised independently by Wadler [1993b] and Barber [1995].

We do not merely shoehorn the classic CPS and monad translations into a new framework, we actually improve on them, taking results previously proved for equalities and extending them to reductions. Further, we show for the first time that Moggi's computational lambda calculus λ_c actually has computational content, and therefore might serve as an improvement over Plotkin's call-by-value lambda calculus λ_v.

Plotkin [1975] showed, among other things, that the call-by-value CPS translation was sound but *not* complete, in that it preserves but does not reflect equalities. Plotkin devised the call-by-value lambda calculus λ_v to model some operational properties of programming languages, notably procedure calls. It was a calculus with a reduction theory, from which one can derive an equational theory. As usual, we take equality to be the symmetric, transitive, and reflexive closure of reduction; it follows from the Church-Rosser theorem that two terms are equal if and only if they reduce to a common term. Plotkin's CPS translation (here written $*$) takes the call-by-value calculus λ_v (the source) into the traditional call-by-name lambda calculus λ_n (the target). Plotkin proved the translation is sound, in that if $M = N$ in the source then $M^* = N^*$ in the target, and showed it is not complete, in that the converse does not hold. He proved soundness by showing that the translation preserves reductions, so if $M \longrightarrow\!\!\!\twoheadrightarrow N$ in the source then $M^* \longrightarrow\!\!\!\twoheadrightarrow N^*$ in the target.

A baker's dozen of years later, Moggi [1988] showed, among other things, that the

call-by-value monad translation is both sound and complete, in that it preserves and reflects equalities. Moggi devised the monadic metalanguage λ_{ml} to model some denotational properties of programming languages, notably computational effects and their interaction with procedure calls; the theory included CPS as a special case. The original λ_{ml} had an equational theory only. Moggi's monad translation (here written $*$) takes the computational lambda calculus λ_c (the source) into the metalanguage λ_{ml} (the target). The translation is both sound and complete, in that $M = N$ in the source if and only if $M^* = N^*$ in the target. This is hardly a surprise, as λ_c was designed precisely so this result would hold.

Later still, Sabry and Felleisen [1993] strengthened Plotkin's result using Moggi's. Their variant of the CPS translation (here written $*$) takes the computational lambda calculus λ_c (the source) into the traditional call-by-name lambda calculus λ_n (the target). They showed that $M = N$ in the source if and only if $M^* = N^*$ in the target. Further, they introduced an inverse CPS translation (here written $\#$) such that $M = N$ in the target if and only if $M^\# = N^\#$ in the source, and $M = M^{*\#}$ in the source, and $N^{\#*} = N$ in the target. Taken together, these conditions amount to an *equational correspondence*. This result showed that monads in general and continuations in particular satisfy exactly the same equations.

Hatcliff and Danvy [1994] strengthened Moggi's result by showing that the monad translation is also an equational correspondence. This extension of Moggi's unsurprising result is equally unsurprising.

What is surprising is how little attention was paid to reductions. The original technical report [Moggi 1988] specified theories of reduction and of equality for λ_c; however, 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}, though 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}.

Our results supersede all of the preceding results, replacing equalities with reductions. The existence of an adjoint is equivalent to stating that $M \longrightarrow N$ in the source implies $M^* \longrightarrow N^*$ in the target, and $M \longrightarrow N$ in the target implies $M^\# \longrightarrow N^\#$ in the source, and $M \longrightarrow M^{*\#}$ in the source, and $N^{\#*} \longrightarrow N$ in the target. These trivially imply the existence of an equational correspondence. Further, since we have a reflection, the mere equality $N^{\#*} = N$ of the equational correspondence is here replaced by syntactic identity $N^{\#*} \equiv N$.

Although Moggi introduced a confluent reduction theory for λ_c, it appears to be treated purely as a technical trick to aid equational reasoning. There is no evidence that Moggi considered it as a model of computation via reduction (and he later confirmed this in a personal communication). Further, λ_c has two rules, $(let.1)$ and $(let.2)$ that look more like expansions then reductions. The order Moggi picked for them was dictated purely by a technicality: the need for confluence.

Nonetheless, the results here show that λ_c relates closely to several models of computation, including ordinary lambda calculus via CPS and the monadic metalanguage. More precisely, there exists an isomorphism between the kernel of λ_c and suitably "shrunk" versions of each of the target calculi. It turns out that the dubious rules $(let.1)$ and $(let.2)$ have a natural interpretation as administrative reductions (that is, they can be viewed as occurring at compile time rather than

run time), which makes them far more intuitive: they correspond to the naming of subterms, which is considered one of the beneficial properties of CPS.

The process of "shrinking" the target languages T deserves some clarification. In general the goal is to find a sublanguage T_* of T whose set of terms is a subset of the set of T-terms and whose reductions are consistent with the T-reductions. The restricted set of T_*-terms is straightforward to specify: it should contain those terms (1) that are reachable from the source language via the translation to the target language and (2) that are closed under the T_*-reductions. The set of T_*-reductions is more problematic to specify. Ideally the restricted set of reductions would be the same as the set of T-reductions, only restricted to the smaller syntax. And indeed such a case arises when restricting λ_{ml} to λ_{ml*}. The other two cases are less than ideal. For the shrinking of λ_n to λ_{cps}, some of the reductions of λ_{cps} correspond to a *sequence* of two λ_n-reductions. For the shrinking of λ_{lin} to λ_{lin*}, one of the λ_{lin*} reductions corresponds to an *equality* of λ_{lin}. One way to fix this bad fit between λ_{lin} and λ_{lin*} might be to develop a new linear calculus with a more suitable computational model.

While the step from equalities to reductions seems natural in retrospect, we doubt we would have taken it without the guiding light of category theory. In particular, we were motivated by noting the close resemblance between the notion of equational correspondence in Sabry and Felleisen [1993] and the properties of an adjunction.

Related Work. Like the wheel, CPS is such an excellent idea that it was reinvented many times [Reynolds 1993]. Two formalizations are due to Fischer [1972] and Plotkin [1975]. It was proposed as a basis for compiler construction by Steele [1978] and subsequently exploited by Kranz et al. [1986], Appel [1992], and others.

One might hope that the variant CPS translation of Fischer [1972] is also a reflection. An earlier version of this article [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.

In applications of CPS to compiling, one first translates the source into the target via CPS and then optimizes the target. One claim of Sabry and Felleisen [1993] was that one could perform the same optimizations directly in the source, and this was further elaborated by Flanagan et al. [1993]. However, the claim was not properly justified: it was only shown that the same equations hold in the source and target. Here we show for the first time that the same reductions hold in source and target, which more properly corresponds to optimization.

Lawall and Danvy [1993] make much of relating forward and inverse translations via two Galois connections and an isomorphism. Their Galois connections are induced from the translations and are not defined a priori. Furthermore, their order is based on the syntactic structure of terms as opposed to their semantic properties under reduction, and it is not preserved by their isomorphism. Hence their three maps do not compose to give a single Galois connection between source and target, whereas here we show such a connection.

Administrative reductions play a central role in many CPS translations, from Plotkin's work onward. Sabry and Felleisen [1993] show a correspondence between what they dub the A-reductions in the source and administrative reductions in the target. Similar correspondences are studied by Flanagan et al. [1993] and Hatcliff

$$\lambda_v \quad \rightarrow \quad \lambda_{ml} \quad \rightarrow \quad \lambda_{lin} \quad \rightarrow \quad \lambda_n$$

$$\lambda_c$$
$$\cup$$
$$\lambda_{c*} \quad \cong \quad \lambda_{ml*} \quad \cong \quad \lambda_{lin*}$$
$$\cup \qquad\qquad \cup \qquad\qquad \cup$$
$$\lambda_{c**} \quad \cong \quad \lambda_{ml**} \quad \cong \quad \lambda_{lin**} \quad \cong \quad \lambda_{cps}$$

Fig. 1.   Summary of results.

and Danvy [1994]. Here we show that administrative reductions correspond precisely to the reductions $M \rightarrow M^{*\#}$ implied by the existence of the adjunction, and we show that the A-normal form corresponds to the kernel of the reflection, the isomorphic image of the target in the source.

The adjunction is a central abstraction of category theory [Mac Lane 1971] and has the Galois connection as a degenerate case [Davey and Priestley 1990]. Adjoints and Galois connections have long been the tool of choice for relating semantic models, but are less often used to relate operational models. A notable exception is the work of Melton et al. [1985], who discuss a notion of compiler correctness strikingly similar to ours. However, their Galois connections happen to be insertions (the map $*$ from source to target is injective), while ours are reflections (the map $\#$ from target to source is injective).

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. So far as we can see, everything works equally well for typed calculi of reductions, such as those considered by Hatcliff and Danvy [1994], since we conjecture that our translations preserve types.

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 article is structured as follows. Section 2 summarizes our results. Section 3 introduces Galois connections and reflections. 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. As before, making the translation into a reflection requires expanding λ_v into λ_c and shrinking λ_{lin} to λ_{lin*}; the reflection also guarantees that λ_{lin*} is isomorphic to the kernel computational calculus λ_{c*}.

Finally we consider the traditional CPS translation from λ_v into λ_n. Again, this translation is not a reflection; but expanding λ_v into λ_c, shrinking λ_n 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; 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 us review the standard results about Galois connections and reflections [Davey and Priestley 1990; Mac Lane 1971; Melton et al. 1985]. 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 antisymmetric).

We write $\longrightarrow$ for a single-step of reduction; $\longrightarrow\!\!\!\!\rightarrow$ 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 $\longrightarrow\!\!\!\!\rightarrow_S$ and a target calculus T with reduction relation $\longrightarrow\!\!\!\!\rightarrow_T$. Reductions are directed in such a way that they correspond to evaluation steps or optimizations. Let the maps $* : S \rightarrow T$ and $\# : T \rightarrow S$ correspond to compiling and decompiling, respectively. Finally, let M_S, N_S range over terms of S, and let M_T, N_T range over terms of T.

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

$$M_S \longrightarrow\!\!\!\!\rightarrow_S N_T{}^\# \text{ if and only if } M_S{}^* \longrightarrow\!\!\!\!\rightarrow_T N_T.$$

There is an alternative characterization of a Galois connection.

PROPOSITION 3.2. *Maps $*$ and $\#$ form a Galois connection from S to T if and only if the following four conditions hold.*

(1) $M_S \longrightarrow\!\!\!\!\rightarrow_S M_S{}^{*\#}$,

(2) $N_T{}^{\#*} \longrightarrow\!\!\!\!\rightarrow_T N_T$,

(3) $M_S \longrightarrow\!\!\!\!\rightarrow_S N_S$ *implies* $M_S{}^* \longrightarrow\!\!\!\!\rightarrow_T N_S{}^*$,

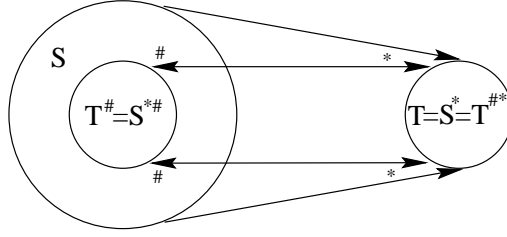*(4)* $M_T \longrightarrow\!\!\!\!\rightarrow_T N_T$ *implies* $N_T{}^\# \longrightarrow\!\!\!\!\rightarrow_S N_T{}^\#$.

Fig. 2.   A reflection and its kernel isomorphism.

If the same four conditions hold with $\longrightarrow\!\!\!\rightarrow_S$ and $\longrightarrow\!\!\!\rightarrow_T$ 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.

   If the compiling map $*$ is left inverse to the decompiling map $\#$, then one has a reflection.

   Definition 3.3.   Maps $*$ and $\#$ form a *reflection* in S of T if they form a Galois connection and $N_T \equiv N_T{}^{\#*}$.

   For a reflection, $\#$ is necessarily injective.   Galois connections and reflections compose.

   PROPOSITION 3.4.   *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 3.5.   *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.*

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

The proposition is illustrated in Figure 2, 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 antisymmetry and hence do not apply here. (As a counterexample to antisymmetry, we have $YI \longrightarrow I(YI) \longrightarrow YI$,

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

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

Fig. 3.   The call-by-value calculus, λ_v.

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

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

Fig. 4.   The monadic calculus, λ_{ml}.

but $YI \not\equiv I(YI)$; I is the identity, and Y is the fixpoint combinator. Note that antisymmetry 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 us 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 3. 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 with which we deal is that it is confluent. Confluence ensures that optimizations (reductions) can be combined in any order.

PROPOSITION 4.1.   *The reductions of λ_v are confluent.*

Moggi's monadic metalanguage λ_{ml} is summarized in Figure 4. 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 values to computations: the term $[M]$ denotes the computation that does nothing save return the value M; and the term $\mathtt{let}\ x \Leftarrow L\ \mathtt{in}\ N$ denotes the computation that composes computation L and N by first performing L, binding its result to x, and then performing N. The interaction of these terms is described by the three rules $(\beta.let)$, $(\eta.let)$, and $(assoc)$. The rule $(assoc)$ is only valid if the obvious scoping restrictions are satisfied (y is not bound in L, and x is not bound in N). We omit such restrictions in the remainder of the article.

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

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

$$
\begin{aligned}
V^* &\equiv [V^\dagger] \\
(LM)^* &\equiv \texttt{let } x \Leftarrow L^* \texttt{ in let } y \Leftarrow M^* \texttt{ in } xy \\[4pt]
(x)^\dagger &\equiv x \\
(\lambda x.\, N)^\dagger &\equiv \lambda x.\, N^*
\end{aligned}
$$

Fig. 5.   Translation of λ_v into λ_{ml}.

A translation $*$ from λ_v to λ_{ml} is described in Figure 5. In the case of applications, the translation introduces two variables x and y that occur free in neither L nor M. 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 $M \longrightarrow\!\!\!\to_v N$ implies $M^* \longrightarrow\!\!\!\to_{ml} N^*$. This is easily checked by looking at the translation of $(\beta.v)$,

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

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

$$((\lambda x.\, xM)L)^* \; =_{ml} \; (LM)^* \quad \text{if } x \notin \mathit{fv}(M)$$

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 6. The new calculus corresponds to Moggi's untyped computational λ-calculus as it appeared in his original Edinburgh LFCS Technical Report [Moggi 1988, Def. 6.2]. It was carefully designed 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 nonvalues. Rules $(let.1)$ and $(let.2)$ are restricted to act on nonvalues, since values yield a reduction in the opposite direction via $(\beta.let)$.

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

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

Fig. 6.   The computational calculus, λ_c.

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

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

Fig. 7.   The simplified monadic calculus, λ_{ml*}.

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 explicitly captures the evaluation order of the subexpressions of an application. To evaluate an application, we evaluate the function position to a value $(let.1)$, and then evaluate the argument to a value $(let.2)$. Furthermore, reversing the rules would yield a nonconfluent system. 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. 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 7. The grammar of λ_{ml} is the smallest one that contains all terms in the image of the translation $* : \lambda_v \to \lambda_{ml}$ of the previous section and which is closed under the reductions of λ_{ml}. The new grammar differs from the old in two key ways: applications MN

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

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

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

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

Fig. 8. Reflection in λ_c of λ_{ml*}.

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

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

Here $\dagger$ is as in Figure 5. 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.

$$
\begin{aligned}
(VQ)^* \equiv \quad & \texttt{let } x \Leftarrow [V^\dagger] \texttt{ in let } y \Leftarrow Q^* \texttt{ in } xy \\
\longrightarrow_{\beta.let} \ & \texttt{let } y \Leftarrow Q^* \texttt{ in } V^\dagger y \\
\longleftarrow_{\beta.let} \ & \texttt{let } y \Leftarrow Q^* \texttt{ in let } x' \Leftarrow [V^\dagger] \texttt{ in let } y' \Leftarrow [y] \texttt{ in } x'y' \\
\equiv \quad & (\texttt{let } y = Q \texttt{ in } Vy)^*
\end{aligned}
$$

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 8. 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 $\#$ (see Figure 8) form a reflection in λ_c of λ_{ml*}.*

PROOF. To prove that the translations form a reflection, we must establish the following (see Proposition 3.2 and Definition 3.3):

(1) For $M \in \lambda_c$, $M \longrightarrow\!\!\!\rightarrow M^{*\#}$,
(2) For $M \in \lambda_{ml*}$, $M^{\#*} \equiv M$,
(3) For $M, N \in \lambda_c$, $M \longrightarrow\!\!\!\rightarrow N$ implies $M^* \longrightarrow\!\!\!\rightarrow N^*$,
(4) For $M, N \in \lambda_{ml*}$, $M \longrightarrow\!\!\!\rightarrow N$ implies $M^\# \longrightarrow\!\!\!\rightarrow N^\#$.

Parts (1) and (2) are verified by induction over the structure of terms, and parts (3) and (4) are verified by induction over the length of reduction sequences. We show the proof of case (3) in detail. The main part of the proof is to show that every reduction between terms in λ_c corresponds to a sequence of reductions between the translated terms. We proceed by cases:

Case $\beta.v$. We have

$$((\lambda x.N)\ V)^* \equiv ((\lambda x.N^*)\ V^\dagger) \longrightarrow N^*[x := V^\dagger] \equiv (N[x := V])^*,$$

where the last step follows by Substitution Lemma 5.4.

Case $\eta.v$. We have

$$(\lambda x.Vx)^* \equiv [\lambda x.V^\dagger x] \longrightarrow [V^\dagger] \equiv V^*.$$

Case $\beta.let$. We have

$$(\texttt{let}\ x = V\ \texttt{in}\ N)^* \equiv \texttt{let}\ x \Leftarrow [V^\dagger]\ \texttt{in}\ N^* \longrightarrow N^*[x := V^\dagger] \equiv (N[x := V])^*,$$

where the last step follows by Substitution Lemma 5.4.

Case $\eta.let$. We have

$$(\texttt{let}\ x = M\ \texttt{in}\ x)^* \equiv \texttt{let}\ x \Leftarrow M^*\ \texttt{in}\ [x] \longrightarrow M^*.$$

Case *assoc*. We have

$$
\begin{aligned}
(\texttt{let}\ y = (\texttt{let}\ x = L\ \texttt{in}\ M)\ \texttt{in}\ N)^* \ &\equiv\ \texttt{let}\ y \Leftarrow (\texttt{let}\ x \Leftarrow L^*\ \texttt{in}\ M^*)\ \texttt{in}\ N^* \\
&\longrightarrow\ \texttt{let}\ x \Leftarrow L^*\ \texttt{in}\ \texttt{let}\ y \Leftarrow M^*\ \texttt{in}\ N^* \\
&\equiv\ (\texttt{let}\ x = L\ \texttt{in}\ (\texttt{let}\ y = M\ \texttt{in}\ N))^*.
\end{aligned}
$$

Case *let*.1. We have

$$(PM)^* \equiv \texttt{let}\ x \Leftarrow P^*\ \texttt{in}\ (xM)^* \equiv (\texttt{let}\ x = P\ \texttt{in}\ xM)^*.$$

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

Reductions $(\beta.v), (\eta.v), (\beta.let), (\eta.let), (assoc)$, as in Figure 6

Fig. 9.   The kernel computational calculus, λ_{c*}.

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

$$
\begin{array}{lll}
V^* & \equiv V^\dagger & \qquad\qquad \text{the trivial inclusion} \\
(PM)^* & \equiv \texttt{let } x = P^* \texttt{ in } (xM)^* & \\
(VQ)^* & \equiv \texttt{let } y = Q^* \texttt{ in } (Vy)^* & \\
(VW)^* & \equiv V^\dagger W^\dagger & \\
(\texttt{let } x = M \texttt{ in } N)^* & \equiv \texttt{let } x = M^* \texttt{ in } N^* & \\
\\
x^\dagger & \equiv x & \\
(\lambda x.\, M)^\dagger & \equiv \lambda x.\, M^* &
\end{array}
$$

Fig. 10.   Inclusion in λ_c of λ_{c*}.

Case $let.2$. We have $(VQ)^* \equiv \texttt{let } y \Leftarrow Q^* \texttt{ in } (Vy)^* \equiv (\texttt{let } y = Q \texttt{ in } Vy)^*$. $\qquad\square$

The proofs of cases (3) and (4) above rely on the following substitution lemma.

LEMMA 5.4. *Let M, V be terms of the appropriate sort in λ_c and N, W be terms of the appropriate sort in λ_{ml*}; then we have*

$$
\begin{array}{ll}
M^*[x := V^\dagger] & \equiv (M[x := V])^* \\
N^\#[x := W^\dagger] & \equiv (N[x := W])^\#.
\end{array}
$$

6.   MONADS: THE FACTORIZATION

Proposition 3.5 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 9. 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*}.

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 3.5, 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 10 and 11. 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*}.

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

$$
\begin{aligned}
V^* &\equiv [V^\dagger] &&\text{same as $\#$ from Figure 8}\\
(VW)^* &\equiv V^\dagger W^\dagger\\
(\texttt{let } x = M \texttt{ in } N)^* &\equiv \texttt{let } x \Leftarrow M^* \texttt{ in } N^*\\[4pt]
x^\dagger &\equiv x\\
(\lambda x.\, M)^\dagger &\equiv \lambda x.\, M^*
\end{aligned}
$$

Fig. 11.   Isomorphism of λ_{c*} and λ_{ml*}.

$$
\begin{aligned}
\text{terms} &\quad L, M, N &&::= V \mid P\\
\text{values} &\quad V &&::= x \mid \lambda a.\, M\\
\text{nonvalues} &\quad P, Q &&::= a \mid LM \mid (!M) \mid \texttt{let } !x = L \texttt{ in } N
\end{aligned}
$$

$$
\begin{aligned}
(\beta.\!\multimap) &\quad (\lambda a.\, N)M &&\longrightarrow N[a := M]\\
(\eta.\!\multimap) &\quad \lambda a.\, Ma &&\longrightarrow M\\
(\beta.!) &\quad \texttt{let } !x =\, !M \texttt{ in } N &&\longrightarrow N[x := M]\\
(\eta.!) &\quad \texttt{let } !x = M \texttt{ in } !x &&\longrightarrow M, \quad \text{if } x \notin \mathit{fv}(M)\\
(!\!\multimap) &\quad (\texttt{let } !x = L \texttt{ in } M)N &&\longrightarrow \texttt{let } !x = L \texttt{ in } (MN)\\
(\multimap!) &\quad V(\texttt{let } !x = L \texttt{ in } M) &&\longrightarrow \texttt{let } !x = L \texttt{ in } (VM)\\
(!!) &\quad \texttt{let } !y = (\texttt{let } !x = L \texttt{ in } M) \texttt{ in } N &&\longrightarrow \texttt{let } !x = L \texttt{ in } (\texttt{let } !y = M \texttt{ in } N)
\end{aligned}
$$

Fig. 12.   The linear calculus, λ_{lin}.

7.   LINEAR CALCULUS

Figure 12 summarizes a linear lambda calculus similar to the ones studied by Wadler [1993a; 1993b] and Maraist et al. [1995]. 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. These conventions are the ones used by Benton [1995] and Benton and Wadler [1996]. 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 $\texttt{let } !x = L \texttt{ in } N$.

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 $!M$. Informally, evaluation of the term $\texttt{let } !x = L \texttt{ in } N$ 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 $\texttt{let } !x = L \texttt{ in } N$ 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.

All the rules of the calculus correspond to simplification of proofs in linear logic. The rules $(\beta\multimap)$ and $(\beta.!)$ simplify a logical introduction followed by the corresponding logical elimination. The rules $(\eta.\multimap)$ and $(\eta.!)$ simplify a logical elimination followed by the corresponding logical introduction. Note that there is no need for

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

$$
\begin{aligned}
V^* &\equiv\ !V^\dagger \\
(LM)^* &\equiv\ (\texttt{let }!x = L^* \texttt{ in } x)M^* \\[6pt]
x^\dagger &\equiv\ x \\
(\lambda x.\, N)^\dagger &\equiv\ \lambda a.\, \texttt{let }!x = a \texttt{ in } N^*
\end{aligned}
$$

<p align="center">Fig. 13. Translation of λ_v into λ_{lin}.</p>

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

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

<p align="center">Fig. 14. The restricted linear calculus, λ_{lin*}.</p>

a side condition in $(\eta.\multimap)$, since linear variables can only occur once in a term. The rules $(!\multimap)$, $(\multimap!)$, and $(!!)$ correspond to commuting conversions. All of the rules, except for $(\multimap!)$, were discussed by Maraist et al. [1995]. We explain the significance of the additional $(\multimap!)$ rule in Section 7.2 after studying the translation from λ_v to λ_{lin}.

7.1 Reflection of Linear Calculus in Call-by-Value

Our starting point is the translation from λ_v to λ_{lin} of Maraist et al. [1995], which is described in Figure 13. This translation maps $(\beta.v)$ reductions to reductions in the linear calculus:

$$
\begin{aligned}
((\lambda x.\, N)V)^* &\equiv & (\texttt{let }!z =\, !(\lambda a.\, \texttt{let }!x = a \texttt{ in } N^*) \texttt{ in } z)\,(!V^\dagger) \\
&\longrightarrow\!\!\!\!\longrightarrow_{(\beta.!)} & (\lambda a.\, \texttt{let }!x = a \texttt{ in } N^*)\,(!V^\dagger) \\
&\longrightarrow\!\!\!\!\longrightarrow_{(\beta.\multimap)} & \texttt{let }!x =\, !V^\dagger \texttt{ in } N^* \\
&\longrightarrow\!\!\!\!\longrightarrow_{(\beta.!)} & N^*[x := V^\dagger] \\
&\equiv & N[x := V]^*
\end{aligned}
$$

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

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

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 restrict the target calculus from λ_{lin} to λ_{lin*}, and to fine-tune the corresponding translation. Instead of constructing these calculi and translations from scratch, we exploit the existing isomorphism between λ_{c*} and λ_{ml*} to guide

$$*_2 : \lambda_{c*} \rightarrow \lambda_{lin*}$$

$$V^* \equiv !V^\dagger$$
$$(VW)^* \equiv V^\dagger(!W^\dagger)$$
$$(\texttt{let } x = M \texttt{ in } N)^* \equiv \texttt{let } !x = M^* \texttt{ in } N^*$$

$$x^\dagger \equiv x$$
$$(\lambda x.\, M)^\dagger \equiv \lambda a.\, \texttt{let } !x = a \texttt{ in } M^*$$

$$\#_2 : \lambda_{lin*} \rightarrow \lambda_{c*}$$

$$x^\# \equiv x$$
$$(\lambda a.\, \texttt{let } !x = a \texttt{ in } M)^\# \equiv \lambda x.\, M^\#$$
$$(V(!W))^\# \equiv V^\# W^\#$$
$$(!V)^\# \equiv V^\#$$
$$(\texttt{let } !x = M \texttt{ in } N)^\# \equiv \texttt{let } x = M^\# \texttt{ in } N^\#$$

Fig. 15.   Isomorphism of λ_{c*} and λ_{lin*}.

the design of λ_{lin*}. The idea is to adapt the optimized translation from λ_{c*} to λ_{ml*} to a translation from λ_{c*} to the linear calculus. The target of this translation would be the kernel linear calculus λ_{lin*}.

The isomorphism is spelled out by the maps $*_2$ and $\#_2$ between λ_{c*} and λ_{lin}, shown in Figure 15. Note the similarity between these two maps and the corresponding maps between λ_{c*} and λ_{ml*} in Figure 11. The terms and reduction rules of the restricted linear calculus in Figure 14 are derived from the terms and reduction rules of λ_{c*} using the map $*_2$ of Figure 15.

PROPOSITION 7.1.1. *The terms and reductions of the calculi λ_{c*} and λ_{lin*} are isomorphic. In other words, for all $M, N \in \lambda_{c*}$ and $P, Q \in \lambda_{lin*}$, we have*

—$M^{*\#} \equiv M$ *and* $P^{\#*} \equiv P$,

—*If* $M \longrightarrow N$ *then* $M^* \longrightarrow N^*$, *and*

—*If* $P \longrightarrow Q$ *then* $P^\# \longrightarrow Q^\#$.

The existence of the isomorphism immediately implies several properties of λ_{lin*}.

PROPOSITION 7.1.2. *The calculus λ_{lin*} is confluent and closed under reduction.*

For this approach to make sense, it remains to verify that the calculus λ_{lin*} is really a restriction of λ_{lin} in the sense that all reductions rules in λ_{lin*} must be provable in λ_{lin}. For most of the rules, this is straightforward to verify. The only nonobvious rule is the λ_{lin*} version of $(\eta.\multimap)$, which can be proved in λ_{lin} as follows:

$$\begin{aligned} \lambda a.\, \texttt{let } !x = a \texttt{ in } V(!x) &= \lambda a.\, (V(\texttt{let } !x = a \texttt{ in } !x)) & (\multimap!) \\ &= \lambda a.\, (V a) & (\eta.!) \\ &= V & (\eta.\multimap) \end{aligned}$$

Note that the proof uses equalities rather than reductions, indicating the bad fit between λ_{lin} and λ_{lin*}.

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

$$
\begin{aligned}
V^* &\equiv\ !V^\dagger && \text{same as } \#_2 \text{ from Figure 15}\\
(PM)^* &\equiv \texttt{let}\ !x = P^*\ \texttt{in}\ (xM)^*\\
(VQ)^* &\equiv \texttt{let}\ !y = Q^*\ \texttt{in}\ (Vy)^*\\
(VW)^* &\equiv V^\dagger(!W^\dagger)\\
(\texttt{let}\ x = M\ \texttt{in}\ N)^* &\equiv \texttt{let}\ !x = M^*\ \texttt{in}\ N^*\\[4pt]
x^\dagger &\equiv x\\
(\lambda x.\,M)^\dagger &\equiv \lambda a.\,\texttt{let}\ !x = a\ \texttt{in}\ M^*
\end{aligned}
$$

Fig. 16. Reflection in λ_c of λ_{lin*}.

Putting together the inclusion in λ_c of λ_{c*} of Figure 10 and the isomorphism between λ_{c*} and λ_{lin*} of Figure 15, we can immediately construct a reflection in λ_c of λ_{lin*}.

PROPOSITION 7.1.3. *Translations $*$ and $\#$ (see Figure 16) form a reflection in λ_c of λ_{lin*}.*

7.2 Eta-Rules for Linear Logic

Maraist et al. [1995] were concerned with a similar problem: finding translations between call-by-value calculi and linear calculi that are both sound and complete. They adopt a similar solution: expanding the source calculus from VAL to LET. One significant difference is that their source calculus does not contain an $(\eta.v)$ rule, and their target calculus does not contain $(\eta.\multimap)$, $(\eta.!)$, and $(\multimap!)$.

Furthermore, when Maraist et al. [1995] attempted to add the $(\eta.v)$ rule to the call-by-value calculus, and the $(\eta.\multimap)$ and $(\eta.!)$ rules to the linear lambda calculus, they noted that the translation from the call-by-value calculus to the linear calculus does not preserve $(\eta.v)$. They suggested that perhaps more reductions rules should be added to the linear calculus if such reductions could be justified logically. We have indeed followed their suggestion to solve the problem by adding the rule $(\multimap!)$, which is the symmetric commuting conversion from $(!\multimap)$. (The symmetry is not complete, since we are evaluating the subexpressions of an application from left-to-right.) With the addition of the $(\multimap!)$ rule, the translation from the call-by-value calculus to the linear calculus preserves the $(\eta.v)$ rule as we have shown.

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_n via the traditional call-by-value CPS translation:

$$
\begin{aligned}
V^* &\equiv \lambda k.\,kV^\dagger\\
(LM)^* &\equiv \lambda k.\,L^*(\lambda x.\,M^*(\lambda y.\,xyk))\\[4pt]
x^\dagger &\equiv x\\
(\lambda x.\,N)^\dagger &\equiv \lambda x.\,N^*
\end{aligned}
$$

A remarkable property of this translation is that one may always choose k to

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

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

Fig. 17. Continuation-passing style calculus λ_{cps}.

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

$$
\begin{array}{lll}
M^* & \equiv & M : k \\[4pt]
V : K & \equiv & KV^\dagger \\
(PM) : K & \equiv & P : (\lambda x.\, ((xM) : K)) \\
(VQ) : K & \equiv & Q : (\lambda y.\, ((Vy) : K)) \\
(VW) : K & \equiv & V^\dagger W^\dagger K \\
(\texttt{let } x = M \texttt{ in } N) : K & \equiv & M : (\lambda x.\, (N : K)) \\[4pt]
x^\dagger & \equiv & x \\
(\lambda x.\, M)^\dagger & \equiv & \lambda x.\, \lambda k.\, M^*
\end{array}
$$

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

$$
\begin{array}{lll}
(KV)^\# & \equiv & K^\flat[V^\natural] \\
(VWK)^\# & \equiv & K^\flat[V^\natural W^\natural] \\[4pt]
x^\natural & \equiv & x \\
(\lambda x.\, \lambda k.\, M)^\natural & \equiv & \lambda x.\, M^\# \\[4pt]
k^\flat & \equiv & [\,] \\
(\lambda x.\, N)^\flat & \equiv & \texttt{let } x = [\,] \texttt{ in } N^\#
\end{array}
$$

Fig. 18. Reflection in λ_c of λ_{cps}.

be exactly the same name, without fear of name clash. Again, this translation is sound but not complete.

To refine the translation $* : \lambda_v \to \lambda_n$ into a reflection also requires three steps: first, grow the source λ_v into λ_c; second, shrink the target λ_n into λ_{cps}; third, fine-tune the translation $*$. Step one was already accomplished as part of the monad development. Like for the linear calculus, 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 $* : \lambda_v \to \lambda_n$, it is suitably extended for $\texttt{let}$ and has

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

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

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

Fig. 19.   The kernel computational calculus λ_{c**}.

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

Fig. 20.   Inclusion in λ_c of λ_{c**}.

some reductions labeled as administrative.

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

The translation now takes place in three stages. Stage one applies the translation proper. Stage two reduces any administrative (β) 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 18. The old translation relates to the new as follows: $M^{*\mathrm{old}} \equiv \lambda k.\, M^{*\mathrm{new}}$, 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_n into λ_{cps}, which is summarized in Figure 17. The grammar is the smallest one that is in the image of the refined translation

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

$$
\begin{aligned}
(K[V])^* &\equiv K^{\ddagger} V^{\dagger} &\qquad \text{same as } \# \text{ from Figure 18}\\
(K[VW])^* &\equiv V^{\dagger} W^{\dagger} K^{\ddagger}\\[4pt]
x^{\dagger} &\equiv x\\
(\lambda x.\, M)^{\dagger} &\equiv \lambda x.\, \lambda k.\, M^{\dagger}\\[4pt]
[\,]^{\ddagger} &\equiv k\\
(\mathtt{let}\ x = [\,]\ \mathtt{in}\ N)^{\ddagger} &\equiv \lambda x.\, N^*
\end{aligned}
$$

Fig. 21.   Isomorphism of λ_{c**} and λ_{cps}.

$*$. An additional class of nonterminals, 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 (β) and (η) 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.

PROPOSITION 8.1. *The grammar of λ_{cps} is closed under the reductions of λ_{cps}.*

The inverse translation $\#$ of Figure 18 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 holes of our evaluation contexts are never in the scope of a bound variable.

We can now verify that the maps of Figure 18 constitute a reflection.

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

PROOF. We prove each of the four parts of Proposition 3.2 separately. The proof of part (1)

$$M \quad \longrightarrow\!\!\!\rightarrow \quad M^{*\#}$$

requires a stronger inductive hypothesis,

$$K^{\flat}[M] \quad \longrightarrow\!\!\!\rightarrow \quad (M : K)^{\#},$$

but is otherwise straightforward. As usual the proofs rely on the relevant substitution lemmas.   □

LEMMA 8.3. *Let M, V be terms of the appropriate sorts in λ_c, and let K be in λ_{cps}; then $(M : K)[x := V^{\dagger}] \equiv (M[x := V]) : K$.*

LEMMA 8.4. *Let M, V, K be terms of the appropriate sort in λ_{cps}; then*

$$K^{\flat}[M^{\#}[x := V^{\natural}]] \quad \longrightarrow\!\!\!\rightarrow \quad (M[x := V][k := K])^{\#}.$$

This completes the parallel development of the reflection; there is also a parallel development of the factorization. Proposition 3.5 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 19. 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

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

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

$$((\lambda y.\,\lambda k.\,(a\ b\ (\lambda z.\,kc)))\ d\ (\lambda x.\,ke))$$
$$\longrightarrow (a\ b\ (\lambda z.\,(\lambda x.\,ke)c))$$

The image of this in λ_{c**} is

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

where the right-hand side is in $(assoc)$-normal form. The CPS language achieves this normalization using the metaoperation 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 metaoperation to λ_{c**}, $M : K$ shown in Figure 19. Using the new metaoperation we can extend the problematic source reduction $(\beta.v)$ with a built-in $(assoc)$-normalization action that mirrors the action of (β) on CPS terms. (An alternative to adding metanotation 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 $((\lambda y.\,\texttt{let } z = ab \texttt{ in } c)\ \ d)$ in the counterexample by taking K to be $[\,]$ rather than $\texttt{let } x = [\,] \texttt{ in } e$. To step around this pitfall, we use a nonstandard convention that pattern-matching a term against the left-hand side of $(\beta.v)$ should select the maximal K.

Again, as guaranteed by Proposition 3.5, 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 20 and 21. 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:

$$
\begin{aligned}
[V]^* &\equiv \overline{\lambda}k.\, kV^\dagger \\
(VW)^* &\equiv \overline{\lambda}k.\, V^\dagger W^\dagger k \\
(\texttt{let } x \Leftarrow M \texttt{ in } N)^* &\equiv \overline{\lambda}k.\, M^*(\lambda x.\, N^*k) \\[4pt]
x^\dagger &\equiv x \\
(\lambda x.\, N)^\dagger &\equiv \lambda x.\, N^*
\end{aligned}
$$

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 15 can be adapted 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 Plotkin translation in Section 8.

$$
\begin{aligned}
V^* &\equiv \overline{\lambda}k.\, kV^\dagger \\
(LM)^* &\equiv \overline{\lambda}k.\, L^*(\overline{\lambda}x.\, M^*(\overline{\lambda}y.\, xky)) \\[4pt]
x^\dagger &\equiv x \\
(\lambda x.\, N)^\dagger &\equiv \overline{\lambda}k.\, \lambda x.\, N^*k
\end{aligned}
$$

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

An earlier version of this article 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, 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, 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 $M \longrightarrow M^{*\#} \equiv P^\#$ and $N \longrightarrow N^{*\#} \equiv P^\#$ in λ_c, and so fy is the only possible choice for $P^\#$. For a Galois connection we also require that $P^{\#*} \longrightarrow P$, but this fails, since $P^{\#*} \equiv fky \not\longrightarrow (\lambda x.\, fkx)y \equiv P$.

We saw in Section 4 that a naive 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_n, 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_n. As they note, $\lambda_v X$ and λ_c prove the same equalities; but they do not prove the same reductions.

Sabry and Felleisen [1993] 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 translation satisfies all four conditions of Proposition 3.2, except that condition (2), $P^{\#*} \longrightarrow\!\!\!\rightarrow P$, 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 et al. [1993] apply the results of Sabry and Felleisen [1993]. 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 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, 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 3.5.

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 et al. [1995] consider translations into a linear lambda calculus. That paper extends the call-by-value calculus VAL to the calculus LET; this article extends the call-by-value calculus λ_v to the calculus λ_c. The earlier paper stressed the analogy 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 (1) to extend the work here by specifying a suitable notion of standard reduction for each of the calculi involved and (2) to show that the given translations are still reflections if one replaces reductions by standard reductions.

Call-by-Value and Call-by-Need. Maraist et al. [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,

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

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 laid 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 the 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_c 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

APPEL, A. W. 1992. *Compiling with Continuations.* Cambridge University Press, Cambridge, Mass.

ARIOLA, Z. M., FELLEISEN, M., MARAIST, J., ODERSKY, M., AND WADLER, P. 1995. A call-by-need lambda calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, New York, 233–246.

BARBER, A. 1995. Dual intuitionistic linear logic. Draft, Dept. of Computer Science, Univ. of Edinburgh, Edinburgh, U.K.

BENTON, N. AND WADLER, P. 1996. Linear logic, monads, and the lambda calculus. In the *Symposium on Logic in Computer Science.* IEEE Computer Society Press, Los Alamitos, Calif.

BENTON, P. N. 1995. A mixed linear and non-linear logic: Proofs, terms, and models. In *Proceedings of Computer Science Logic* (Kazimierz, Poland, 1994). Lecture Notes in Computer Science, vol. 933. Springer-Verlag, Berlin. Full version available as Tech. Rep. 352, Computer Laboratory, Univ. of Cambridge, October 1994.

DAVEY, B. A. AND PRIESTLEY, H. A. 1990. *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, Mass.

FISCHER, M. 1972. Lambda calculus schemata. In *Conference on Proving Assertions about Programs. SIGPLAN Not. 7*, 1, 104-109. Revised version in *Lisp and Symbolic Computation 6*, 3/4 (1993), 259-287.

FLANAGAN, C., SABRY, A., DUBA, B., AND FELLEISEN, M. 1993. The essence of compiling with continuations. In the *ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, New York, 237–247.

GIRARD, J.-Y. 1987. Linear logic. *Theoret. Comput. Sci. 50*, 1–102.

HATCLIFF, J. AND DANVY, O. 1994. A generic account of continuation-passing styles. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, New York, 458–471.

KRANZ, D., KELSEY, R., REES, J., HUDAK, P., PHILBIN, J., AND ADAMS, N. 1986. Orbit: An optimizing compiler for Scheme. In the *ACM SIGPLAN Symposium on Compiler Construction. SIGPLAN Not. 21*, 7, 219-233.

LANDIN, P. 1964. The mechanical evaluation of expressions. *Comput. J. 6,* 4, 308–320.

LAWALL, J. AND DANVY, O. 1993. Separating stages in the continuation-passing transform. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 124–136.

MAC LANE, S. 1971. *Categories for the Working Mathematician*. Springer-Verlag, Berlin.

MARAIST, J., ODERSKY, M., TURNER, D. N., AND WADLER, P. 1995. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. In the *International Conference on the Mathematical Foundations of Programming Semantics*.

MELTON, A., SCHMIDT, D. A., AND STRECKER, G. 1985. Galois connections and computer science applications. In *Category Theory and Computer Programming*. Lecture Notes in Computer Science, vol. 240. Springer-Verlag, Berlin, 299–312.

MOGGI, E. 1988. Computational lambda-calculus and monads. Tech. Rep. ECS-LFCS-88-86, Univ. of Edinburgh, Edinburgh, U.K.

MOGGI, E. 1989. Computational lambda-calculus and monads. In the *Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 14–23.

MOGGI, E. 1991. Notions of computation and monads. *Inf. Comput. 93*, 55–92.

PLOTKIN, G. 1975. Call-by-name, call-by-value, and the λ-calculus. *Theoret. Comput. Sci. 1*, 125–159.

REYNOLDS, J. C. 1993. The discoveries of continuations. *Lisp Symbol. Comput. 6,* 3/4, 233–247.

SABRY, A. AND FELLEISEN, M. 1993. Reasoning about programs in continuation-passing style. *Lisp Symbol. Comput. 6,* 3/4, 289–360.

SABRY, A. AND WADLER, P. 1996. A reflection on call-by-value. In *SIGPLAN International Conference on Functional Programming*. ACM Press, New York, 13–24.

STEELE, G. L. 1978. Rabbit: A compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass.

WADLER, P. 1993a. A syntax for linear logic. In the *International Conference on the Mathematical Foundations of Programming Semantics*. Lecture Notes in Computer Science, vol. 802. Springer-Verlag, Berlin.

WADLER, P. 1993b. A taste of linear logic (Invited talk). In *Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, vol. 711. Springer-Verlag, Berlin.