Call-by-name, call-by-value, call-by-need, and the linear lambda calculus

John Maraist and Martin Odersky

Institut für Programmstrukturen, University of Karlsruhe, 76128 Karlsruhe, Germany. E-mail: `maraist,odersky@ira.uka.de`

David N. Turner and Philip Wadler

Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: `dnt,wadler@dcs.glasgow.ac.uk`

Abstract

Girard described two translations of intuitionistic logic into linear logic, one where $A \to B$ maps to $(!A) \multimap B$, and another where it maps to $!(A \multimap B)$. We detail the action of these translations on terms, and show that the first corresponds to a call-by-name calculus, while the second corresponds to call-by-value. We further show that if the target of the translation is taken to be an affine calculus, where ! controls contraction but weakening is allowed everywhere, then the second translation corresponds to a call-by-need calculus, as recently defined by Ariola, Felleisen, Maraist, Odersky and Wadler. Thus the different calling mechanisms can be explained in terms of logical translations, bringing them into the scope of the Curry-Howard isomorphism.

1 Introduction

Plotkin, in "Call-by-name, call-by-value and the λ-calculus" [25], demonstrated how two different calling mechanisms could be explained by two different translations into continuation passing style. At the time Plotkin wrote, the call-by-value translation was widely appreciated, but the call-by-name translation was less well known. In particular, the call-by-value translation was rediscovered several times (as related by Reynolds [27]), while the call-by-name translation appears to have been known only to Plotkin and Reynolds (the former credits it to the latter).

While we hesitate to compare our work to Plotkin's, our goal here is somewhat similar. We demonstrate how the two different calling mechanisms can

* Presented at *Mathematical Foundations of Programming Semantics (MFPS)*, New Orleans, Lousiana, March-April 1995.

be explained by two different translations into linear logic. At the time we are writing, the call-by-name translation is widely appreciated, but the call-by-value translation is less well known. Both translations can be found in the original paper of Girard [11], the first based on mapping $A \to B$ into $(!A) \multimap B$, and the second based on mapping it into $!(A \multimap B)$. He devotes several pages to the first translation and less than a paragraph to the second, stating that 'its interest is limited'. That the first translation corresponds to call-by-name appears to be widely known, while the knowledge that the second translation corresponds to call-by-value appears to be restricted to a narrower circle.

A number of different lambda calculi based on linear logic have been proposed, including work by Lafont [16], Holmström [14], Wadler [34–37], Abramsky [1], Mackie [19], Lincoln and Mitchell [18], Troelstra [32], Benton, Bierman, dePaiva, and Hyland [6–9], and della Rocca and Roversi [28]. Various embeddings of intuitionistic logic into linear logic have also been studied, including work by Girard [11], Troelstra [32], and Schellinx [29]. The linear lambda calculus used in this paper is a minor refinement of one previously presented by Wadler [36,37], which is based on Girard's successor to linear logic, the Logic of Unity [12]. A similar calculus has been devised by Plotkin [26].

Corresponding to Girard's first translation we define a mapping $\circ$ from the call-by-name calculus to the linear calculus, and show that this mapping is sound, in that $M \xrightarrow{\text{NAME}}\!\!\!\!\rightarrow N$ implies $M^\circ \xrightarrow{\text{LIN}}\!\!\!\!\rightarrow N^\circ$, and complete, in that the converse also holds. Corresponding to Girard's second translation we define a second mapping $*$ from the call-by-value calculus to the linear calculus, and show that this mapping is also sound and complete, in that $M \xrightarrow{\text{VAL}} N$ if and only if $M^* \xrightarrow{\text{LIN}} N^*$. We believe these soundness and completeness results to be new; showing soundness is straightforward, while completeness is somewhat trickier to establish.

Since writing the first draft of this paper, we have become aware of the work of Mackie [20], which also observes that the first translation corresponds to call-by-name and the second to call-by-value. (He also states that these observations are common in the literature; but he gives no references and we have been unable to locate any.) Mackie's work is complementary to our own. Our translations are into a linear lambda calculus, corresponding to intuitionistic linear logic, while Mackie's translation is into proof nets, corresponding to classical linear logic. We prove soundness and completeness for beta (but not eta); while Mackie proves soundness (but not completeness) for beta and eta. Mackie also says nothing about call-by-need, which we discuss below.

Gonthier, Abadi, and Levy [13] also have a translation based on taking $A \to B$ into $!(A \multimap B)$, but it is rather more complex than our translation at the term level, because it deals with a rather more complex notion of optimal reduction.

Our original motivation for studying these questions came from an interest not in call-by-name or in call-by-value, but instead in a call-by-need calculus, which was recently proposed by Ariola, Felleisen, Maraist, Odersky, and Wadler [2,3,24]. The call-by-name calculus is not entirely suitable for reasoning about functional programs in lazy languages, because the beta rule may copy the argument of a function any number of times. The call-by-need calculus uses a different notion of reduction, observationally equivalent to the call-by-name calculus. But call-by-need, like call-by-value, guarantees that the argument to a function is not copied before it is reduced to a value.

The emphasis on avoiding copying suggests that the 'resource conscious' approach of linear logic may be relevant. In the linear lambda calculus (as in linear logic), the '!' connective is used to control duplication (contraction) and discarding (weakening) of lambda terms (proofs). For call-by-need we wish to avoid duplication but not discarding, and so an appropriate target for our translation is an affine calculus, in which contraction is controlled by the '!' connective but weakening is allowed everywhere. The use of '!' to separately control contraction and weakening has been studied by Jacobs [15].

We derive the call-by-need calculus from the call-by-value calculus in two steps. The first step adds 'let' terms, which enforce sharing, to the call-by-value calculus. The resulting call-by-let calculus is observationally equivalent to call-by-value; the $*$ translation, easily extended, is still sound and complete. We then add one further law, which allows a value bound by a 'let' to be discarded without first being computed if the value is not needed for the result. The resulting call-by-need calculus is observationally equivalent to call-by-name as opposed to call-by-value, and the $*$ translation remains sound and complete if its target is taken to be an affine calculus as opposed to the linear calculus.

As a result, the call-by-value and call-by-need calculi are brought into the scope of the Curry-Howard isomorphism, as the $*$ translation relates these to reductions of the linear calculus that have a clear logical explanation.

An additional contribution of this work is that we confirm that our linear calculus is confluent. Although many linear calculi have been described, relatively few possess claims of confluence, notable exceptions being the work of Benton [7], Bierman [9], and della Rocca and Roversi [28].

As an application of these results, we have devised a type system that can infer information about which variables are used linearly in a call-by-need or call-by-value lambda calculus. This is useful for program transformation: the reduction $((\lambda x.\, M)\; N) \rightarrow M[x := N]$ does not in general hold for a call-by-value or call-by-need calculus, but it does hold if x is used linearly. It may also useful for implementing call-by-need: normally a closure needs to be overwritten on evaluation, this step may be saved if the closure is bound to a variable that is used linearly. These applications are developed in a companion paper by Turner, Wadler, and Mossin [33].

The remainder of this paper is organised as follows. Section 2 introduces the linear lambda calculus. Sections 3–6 describe the call-by-name, call-by-value, call-by-let, and call-by-need calculi and their translations. For simplicity, we restrict our attention to function types only. Section 7 sketches how this work may be extended by adding products, by adding constants and primitive operations, or by removing types; and it remarks that adding sums or recursion is more problematic. Section 8 concludes.

Full versions of the proofs can be found in the technical report version of this paper [23].

2 Linear lambda calculus

We begin with a linear lambda calculus similar to one introduced previously by Wadler [36,37].

In many presentations of logic a key role is played by the structural rules: contraction provides the only way to duplicate an assumption, while weakening provides the only way to discard one. In linear logic [11], the presence of contraction or weakening is revealed in a formula by the presence of the 'of course' connective, written '!'. The Logic of Unity [12] takes this one step further by introducing two sorts of assumption, one that cannot be contracted or weakened, which we will write $x : A$, and one which can, which we will write $!x : !A$.

Figure 1 presents the details of the linear lambda calculus LIN. A type, corresponding to a formula of the logic, is either a base type, an 'of course' type, or a linear function; let Z range over base types, and A, B, C range over types. A term, corresponding to a proof in the logic, is either a variable, an 'of course' introducer, an 'of course' eliminator, a function abstraction, or a function application; let x, y, z range over variables, and L, M, N range over terms. We write $M[x := N]$ for the result of substituting term N for every free occurrence of the variable x in term M.

Contexts are sets containing the two forms of assumption, $x : A$ and $!x : !A$, where each variable x is distinct. Let Γ and Δ range over contexts. If Γ and Δ are contexts with distinct variables, then Γ, Δ denotes their union. If Γ is a context of the form $x_1 : A_1, \ldots, x_n : A_n$, then $!\Gamma$ is the context $!x_1 : !A_1, \ldots, !x_n : !A_n$.

A typing judgement $\Gamma \vdash M : A$ indicates that in context Γ term M has type A. If the judgement

$$x_1 : A_1, \ldots, x_m : A_m, !y_1 : !B_1, \ldots, !y_n : !B_n \vdash M : C$$

holds then the free variables of M will be drawn from $x_1, \ldots, x_n$, each of which occurs linearly, and $y_1, \ldots, y_n$, each of which occurs any number of times.

Those familiar with linear logic or the Logic of Unity may observe that the following three statements are equivalent:

- There exist variables $x_1, \ldots, x_m, y_1, \ldots, y_n$ and term M such that the judgement $x_1 : A_1, \ldots, x_m : A_m, !y_1 : !B_1, \ldots, !y_n : !B_n \vdash M : C$ holds in LIN.
- The judgement $A_1, \ldots, A_m, !B_1, \ldots, !B_n \vdash C$ holds in linear logic.
- The judgement $A_1, \ldots, A_m;\ B_1, \ldots, B_n \vdash C$ holds for the Logic of Unity (without polarities).

There are five rules concerned with the ! connective: Dereliction, Contraction, and Weakening are structural rules, while !-I introduces and !-E eliminates the ! connective. An assumption of the form $!x : !A$ can only be introduced by the !-E rule. The only thing that may be done with such an

Syntactic domains

Types $A, B, C \ ::= \ Z \mid {!A} \mid A \multimap B$
Terms $L, M, N \ ::= \ x \mid {!M} \mid \text{let } {!x} = M \text{ in } N \mid \lambda x.\, M \mid M\ N$

Typing judgements

$$\text{Id}\ \frac{}{x : A \vdash x : A} \qquad \text{Dereliction}\ \frac{\Gamma,\ x : A \vdash M : B}{\Gamma,\ !x : !A \vdash M : B}$$

$$\text{Contraction}\ \frac{\Gamma,\ !y : !A,\ !z : !A \vdash M : B}{\Gamma,\ !x : !A \vdash M[y := x, z := x] : B}$$

$$\text{Weakening}\ \frac{\Gamma \vdash M : B}{\Gamma,\ !x : !A \vdash M : B}$$

$$\text{!-I}\ \frac{!\Gamma \vdash M : A}{!\Gamma \vdash !M : !A} \qquad \text{!-E}\ \frac{\Gamma \vdash M : !A \qquad \Delta,\ !x : !A \vdash N : B}{\Gamma,\ \Delta \vdash \text{let } !x = M \text{ in } N : B}$$

$$\multimap\text{-I}\ \frac{\Gamma,\ x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \multimap B} \qquad \multimap\text{-E}\ \frac{\Gamma \vdash M : A \multimap B \qquad \Delta \vdash N : A}{\Gamma,\ \Delta \vdash M\ N : B}$$

Reduction relation

$(\beta\multimap)$	$(\lambda x.\, M)\ N$	$\to M[x := N]$
$(\beta!)$	$\text{let } !x = !M \text{ in } N$	$\to N[x := M]$
$(!\multimap)$	$(\text{let } !x = L \text{ in } M)\ N$	$\to \text{let } !x = L \text{ in } (M\ N)$
$(!!)$	$\text{let } !y = (\text{let } !x = L \text{ in } M) \text{ in } N$	$\to \text{let } !x = L \text{ in } (\text{let } !y = M \text{ in } N)$

Fig. 1. The linear lambda calculus LIN.

assumption is to duplicate it via Contraction, discard it via Weakening, or convert it to an assumption of the form $x : A$ via Dereliction. Furthermore, it is only assumptions of this form that can appear in the !-I rule, as indicated by writing $!\Gamma$ for the context. Analogous to the conclusion $x : A \vdash x : A$ of Id, the conclusion $!x : !A \vdash !x : !A$ may be derived by combining Id, Derelection, and !-I.

The reduction relation is specified by two beta rules, $(\beta\multimap)$ and $(\beta!)$, and two commuting rules, $(!\multimap)$ and $(!!)$. We take the reduction relation to be the compatible closure of the given rules, as we do for all reduction systems presented in the remainder of this paper. Also, in order to avoid capture, we assume free and bound variables of a term are distinct; for instance, in rules $(!\multimap)$ and $(!!)$, variable x cannot appear free in term N.

Some notation: We write $\twoheadrightarrow$ for the reflexive and transitive closure of $\to$. We write $=$ for the reflexive, symmetric, and transitive closure of $\to$, and we write $\equiv$ for syntactic identity. When necessary, we may write the name of a calculus above a symbol to disambiguate, as in $\overset{\text{LIN}}{\vdash}$ or $\overset{\text{LIN}}{\twoheadrightarrow}$. We may also write the name of a rule below an arrow to indicate which rule is applied, as in $\xrightarrow[(\beta\multimap)]{\text{LIN}}$.

Each of the reduction rules has a logical basis.

- Rule $(\beta\multimap)$ arises when a $\multimap$-I rule meets a $\multimap$-E rule, and the two rules annihilate.

- Rule ($\beta!$) arises when a !-I rule meets a !-E rule, and the two rules annihilate.
- Rule ($!\multimap$) arises when a !-E rule meets a $\multimap$-E rule, commuting one through the other.
- Rule (!!) arises when two !-E rules meet, commuting one through the other.

These reduction rules are compatible with an operational interpretation where one evaluates (let $!x = M$ in N) by first evaluating M to the form $!M'$ and then evaluating $N[x := M']$. Thus we may view the term associated with !-E as forcing evaluation, and the term associated with !-I as suspending evaluation.

It is important to verify that the substitutions respect the restrictions on variables. In rule ($\beta\multimap$), the variable x appears linearly in M, and so any free variable that appears linearly in N will still appear linearly in $M[x := N]$. Hence the substitution is well formed. In rule ($\beta!$) the variable x may appear any number of times in N, and so a free variable of M may be copied arbitrarily many times in $N[x := M]$. This is where distinguishing two sorts of assumptions is helpful: the constraint on the !-I rule guarantees that the term $!M$ may only contain free variables that can appear any number of times. Hence this substitution is also well formed.

Some terminology: A calculus satisfies the *subject reduction* property if whenever $\Gamma \vdash M : A$ and $M \to N$ then $\Gamma \vdash N : A$. A calculus is *confluent* if whenever $L \twoheadrightarrow M$ and $L \twoheadrightarrow M'$, there exists a term N such that $M \twoheadrightarrow N$ and $M' \twoheadrightarrow N$. All of the systems we study will possess both the subject reduction and confluence properties.

Proposition 2.1 LIN *satisfies subject reduction.*

Proposition 2.2 LIN *is confluent.*

Proof. The subject reduction result is straightforward: its essence is is the logical content of the reduction rules listed above. For confluence, the proof is similar to our proof of confluence for the call-by-need lambda calculus [24], which uses Barendregt's technique of marked and weighted redexes with a norm for weighted terms [5]. □

We conclude this section with a few words about the relation of our linear lambda calculus to other formulations.

The formulation given here is based on the Logic of Unity [12], but omitting the extra complication of polarities. In Girard's presentation, the rule !-E does not appear, but it can be derived by combining his ! elimination rule (the fourth to last rule on the right in his Figure 2) with one of his structural rules (the last rule on the right in his Figure 1). We chose our formulation because it yields a simpler ($\beta!$) rule.

Our system follows the Logic of Unity, but differs from most other linear lambda calculi, in that we use two forms of assumption, which enables the subject reduction property to be established in a simple way. The other systems listed in the introduction either lack this property altogether, or satisfy

Syntactic domains

Types	A, B, C	$::=$	$Z \mid A \to B$
Terms	L, M, N	$::=$	$V \mid M\ N$
Values	V, W	$::=$	$x \mid \lambda x.\, M$

Typing judgements

$$\text{Id}\ \frac{}{x : A \vdash x : A}$$

$$\text{Contraction}\ \frac{\Gamma,\ y : A,\ z : A \vdash M : B}{\Gamma,\ x : A \vdash M[y := x, z := x] : B}$$

$$\text{Weakening}\ \frac{\Gamma \vdash M : B}{\Gamma,\ x : A \vdash M : B}$$

$$\to\text{-I}\ \frac{\Gamma,\ x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B} \qquad \to\text{-E}\ \frac{\Gamma \vdash M : A \to B \qquad \Delta \vdash N : A}{\Gamma,\ \Delta \vdash M\ N : B}$$

Reduction relation

$$(\beta\!\to) \qquad (\lambda x.\, M)\ N \to M[x := N]$$

Translation

$$\begin{array}{ll} Z^\circ & \equiv Z \\ (A \to B)^\circ & \equiv (!A^\circ) \multimap B^\circ \\ x^\circ & \equiv x \\ (\lambda x.\, M)^\circ & \equiv \lambda y.\, \text{let } !x = y \text{ in } M^\circ \\ (M\ N)^\circ & \equiv M^\circ\ !N^\circ \\ (x_1 : A_1,\ \ldots,\ x_n : A_n)^\circ & \equiv !x_1 : !A_1^\circ,\ \ldots,\ !x_n : !A_n^\circ \end{array}$$

Fig. 2. The call-by-name lambda calculus NAME

only a restricted version, or else possess full subject reduction but have a more complex syntax for ! introduction.

Also, most other systems treat Weakening and Contraction as logical rules with associated term forms. Our system treats Weakening and Contraction as structural rules, without the clutter of term forms. The result is more compact, and arguably more suitable as the basis of a programming language.

Some elaboration of the above points can be found in our previous work [36,37]. Every term of our language is also a term of the language in [36], from which it is easy to see how to give a semantics to this language in a categorical model in the style of Seely [30] as ammended by Bierman [8,9].

3 Call-by-name

Figure 2 reviews the call-by-name lambda calculus NAME and presents its translation into the linear lambda calculus. Types, terms, and values are standard: a type is a base type or function type, a term is a value or a function application, and a value is a variable or a function abstraction.

Contexts are sets of assumptions of the form $x : A$, where each variable x

is distinct. A typing judgement $\Gamma \vdash M : A$ indicates that in context Γ term M has type A. The type rules are standard. We chose a formulation with Weakening and Contraction to stress the connection with the linear type system, the key difference being that now the use of Contraction and Weakening is unconstrained.

There is a single reduction rule, $(\beta{\rightarrow})$. This calculus satisfies the usual subject reduction and confluence results.

Proposition 3.1 NAME *satisfies subject reduction.* □

Proposition 3.2 NAME *is confluent.* □

Translation $\circ$ takes types, terms, and contexts A, M, Γ of the call-by-name lambda calculus NAME to types, terms, and contexts A°, M°, Γ° of the linear lambda calculus LIN. In the translation of abstractions, y is a fresh variable, not appearing in M.

The idea behind this translation is that ! is added to the left of $\vdash$ and $\multimap$, but not to the right. Every function argument is surrounded by !, which can be thought of as suspending evaluation, corresponding to the call-by-name discipline.

In particular, corresponding to $(\beta \rightarrow)$ we have

$$\begin{array}{ll} & ((\lambda x.\, M)\, N)^\circ \\ \equiv & (\lambda y.\, \text{let } !x = y \text{ in } M^\circ)\, !N^\circ \\ \xrightarrow[(\beta\multimap)]{\text{LIN}} & \text{let } !x = !N^\circ \text{ in } M^\circ \\ \xrightarrow[(\beta!)]{\text{LIN}} & M^\circ[x := N^\circ] \\ \equiv & (M[x := N])^\circ \end{array}$$

which shows that the translation is sound.

Proposition 3.3 (Call-by-name translation) *The translation $\circ$ from* NAME *to* LIN *preserves substitution, types, and reductions:*

(i) $(M[x := N])^\circ \equiv M^\circ[x := N^\circ]$.

(ii) $\Gamma \overset{\text{NAME}}{\vdash} M : A$ *if and only if* $\Gamma^\circ \overset{\text{LIN}}{\vdash} M^\circ : A^\circ$.

(iii) $M \overset{\text{NAME}}{\twoheadrightarrow} N$ *if and only if* $M^\circ \overset{\text{LIN}}{\twoheadrightarrow} N^\circ$.

Proof. We prove (i) by an easy structural induction over terms of NAME, and prove (ii) by an easy structural induction over type derivations in NAME. The proof of (iii) in the forward direction is given above. For the backward direction, we consider the grammar

$$S, T ::= x \mid \lambda y.\, \text{let } !x = y \text{ in } S \mid S\, !T \mid \text{let } !x = !S \text{ in } T \ ,$$

which defines the set of LIN terms reachable from translations of terms in NAME: $M^\circ \overset{\text{LIN}}{\twoheadrightarrow} S$. We then define an erasure $\dagger$ that takes this set back into NAME:

$$\begin{array}{ll} x^\dagger & \equiv x \\ (\lambda y.\, \text{let } !x = y \text{ in } S)^\dagger & \equiv \lambda x.\, S^\dagger \\ (S\, !T)^\dagger & \equiv S^\dagger\, T^\dagger \\ (\text{let } !x = !S \text{ in } T)^\dagger & \equiv T^\dagger[x := S^\dagger] \ . \end{array}$$

Syntactic domains As for NAME.

Typing judgements As for NAME.

Reduction relation

$(\beta{\rightarrow_v}) \qquad (\lambda x.\, M)\, V \;\rightarrow\; M[x := V]$

Translation

$$
\begin{array}{ll}
A^* & \equiv\; !A^+ \\
Z^+ & \equiv\; Z \\
(A \rightarrow B)^+ & \equiv\; (A^* \multimap B^*) \\
V^* & \equiv\; !V^+ \\
(M\; N)^* & \equiv\; (\text{let } !z = M^* \text{ in } z)\; N^* \\
x^+ & \equiv\; x \\
(\lambda x.\, M)^+ & \equiv\; \lambda y.\, \text{let } !x = y \text{ in } M^* \\
(x_1 : A_1, \ldots, x_n : A_n)^* & \equiv\; !x_1 : !A_1^+, \ldots, !x_n : !A_n^+
\end{array}
$$

Fig. 3. The call-by-value lambda calculus VAL.

It is straighforward to show:

(a) The mapping † is a right-inverse of ∘: $M^{\circ\dagger} \equiv M$, for all M in NAME.

(b) The mapping † sends LIN-reduction sequences to NAME-reduction sequences: if $S \xrightarrow{\text{LIN}} T$ then $S^\dagger \xrightarrow{\text{NAME}}\!\!\!\!\rightarrow T^\dagger$.

Now, $M^\circ \xrightarrow{\text{LIN}}\!\!\!\!\rightarrow N^\circ$ implies by (b) that $M^{\circ\dagger} \xrightarrow{\text{NAME}}\!\!\!\!\rightarrow N^{\circ\dagger}$, which implies by ($a$) that $M \xrightarrow{\text{NAME}}\!\!\!\!\rightarrow N$. □

4 Call-by-value

Figure 3 reviews the call-by-value lambda calculus VAL and presents its translation into the linear lambda calculus. Types, terms, and values are as in the call-by-name calculus.

There is a single reduction rule, $(\beta{\rightarrow_v})$, which is a restriction of $(beta{\rightarrow})$ to the case when the function argument is a value. Again, the usual subject reduction and confluence results hold.

Proposition 4.1 VAL *satisfies subject reduction.* □

Proposition 4.2 VAL *is confluent.* □

Translation $*$ takes types, terms, and contexts A, M, Γ of VAL to types, terms, and contexts A^*, M^*, Γ^* of LIN. There is also an auxiliary mapping $+$ from types and values A, V of VAL to types and terms A^+, V^+ of LIN; this mapping omits the outermost '!'. As before, in the translation of applications and abstractions y and z are fresh variables not appearing in M.

The idea behind this translation is that ! is added on both the left and right of $\vdash$ and $\multimap$. Function arguments are no longer surrounded by !, so the argument will be reduced until a ! is encountered before evaluation of the function body proceeds. Under this translation, the only terms beginning with

! are those of the form $V^* \equiv !V^+$, so function arguments are forced to reduce to values.

In particular, corresponding to the call-by-name $(\beta \to)$ rule we have

$$
\begin{array}{ll}
 & ((\lambda x.\, M)\; N)^* \\
\equiv & (\text{let } !z = !(\lambda y.\, \text{let } !x = y \text{ in } M^*) \text{ in } z)\; N^* \\
\xrightarrow[(\beta !)]{\text{LIN}} & (\lambda y.\, \text{let } !x = y \text{ in } M^*)\; N^* \\
\xrightarrow[(\beta \multimap)]{\text{LIN}} & \text{let } !x = N^* \text{ in } M^* \;\; ,
\end{array}
$$

and the reduction cannot necessarily proceed further. But if we replace the argument term N by a value V, then corresponding to the call-by-value $(\beta \to_v)$ rule we have

$$
\begin{array}{ll}
 & ((\lambda x.\, M)\; V)^* \\
\xrightarrow{\text{LIN}} & \text{let } !x = !V^+ \text{ in } M^* \\
\xrightarrow[(\beta !)]{\text{LIN}} & M^*[x := V^+] \\
\equiv & (M[x := V])^*
\end{array}
$$

which shows that the translation is sound.

Proposition 4.3 (Call-by-value translation) *The translation $*$ from* VAL *to* LIN *preserves substitution of values, preserves types, and preserves reductions.*

(i) $(M[x := V])^* \equiv M^*[x := V^+]$.

(ii) $\Gamma \overset{\text{VAL}}{\vdash} M : A$ *if and only if* $\Gamma^* \overset{\text{LIN}}{\vdash} M^* : A^*$, *and* $\Gamma \overset{\text{VAL}}{\vdash} V : A$ *if and only if* $\Gamma^* \overset{\text{LIN}}{\vdash} V^+ : A^+$.

(iii) $M \overset{\text{VAL}}{\twoheadrightarrow} N$ *if and only if* $M^* \overset{\text{LIN}}{\twoheadrightarrow} N^*$.

Proof. Prove (i) and (ii) similarly to NAME. The proof of (iii) in the forward direction is given above. The backward direction follows from two results stated in the next section: that the implication holds in an enriched call-by-value calculus LET, and that reduction in LET conservatively extends reduction in VAL. □

5 Call-by-let

Soundness is reasonably robust under variation of the translations, while completeness is fairly fragile. For instance, one might alter the translation of application to

$$(M\; N)^* \equiv \text{let } !z = M^* \text{ in } (z\; N^*)$$

and this turns out to be sound but no longer complete.

Or say we wish to extend VAL with a 'let' construct, where 'let $x = M$ in N' has the same semantics as $((\lambda x.\, N)\; M)$. We previously observed that

$$((\lambda x.\, N)\; M)^* \overset{\text{LIN}}{\twoheadrightarrow} \text{let } !x = M^* \text{ in } N^* \;\; .$$

So it seems eminently sensible to define

$$(\text{let } x = M \text{ in } N)^* \equiv \text{let } !x = M^* \text{ in } N^* \;\; .$$

This is clearly sound, but again completeness has been lost.

Syntactic domains As for VAL, plus the following:

Terms $\quad L, M, N ::= \cdots \mid \text{let } x = M \text{ in } N$

Typing judgements As for VAL, plus the following:

$$\text{Let } \frac{\Gamma \vdash M : A \qquad \Delta, x : A \vdash N : B}{\Gamma, \Delta \vdash \text{let } x = M \text{ in } N : B}$$

Reduction relation

(I)	$(\lambda x. M)\, N$	$\to \text{let } x = N \text{ in } M$
(V)	$\text{let } x = V \text{ in } M$	$\to M[x := V]$
(C)	$(\text{let } x = L \text{ in } M)\, N$	$\to \text{let } x = L \text{ in } (M\, N)$
(A)	$\text{let } y = (\text{let } x = L \text{ in } M) \text{ in } N$	$\to \text{let } x = L \text{ in } (\text{let } y = M \text{ in } N)$

Translation As for VAL, plus the following:

$(\text{let } x = M \text{ in } N)^* \equiv \text{let } !x = M^* \text{ in } N^*$

Fig. 4. The call-by-let calculus.

To restore completeness in the latter case we add more laws to VAL, yielding a system we call LET. The new laws are similar to the commuting conversions of LIN, and also have some similarity to Moggi's computational lambda calculus [21,22], as discussed in Section 8.3.

Figure 4 defines the call-by-let lambda calculus LET and presents its translation into the linear lambda calculus. The terms are the same as previously, plus a 'let' contruct, and the values remain unchanged. The types and type rules are also the same, with the addition of the obvious rule for 'let'.

Reduction for LET is defined by the four rules (I), (V), (C), (A), which stand for 'Introduce', 'Value', 'Commute', and 'Associate'. As before, subject reduction and confluence both hold.

Proposition 5.1 LET *satisfies subject reduction.*

Proposition 5.2 LET *is confluent.*

Proof. Subject reduction can be verified straightforwardly. Confluence is easily shown by modifying the our previous results for call-by-need [24]. □

The translation $*$ is extended by adding a clause for 'let'.

Proposition 5.3 (Call-by-let translation) *The translation $*$ from* LET *to* LIN *preserves substitution of values, preserves types, and preserves reductions.*

(i) $(M[x := V])^* \equiv M^*[x := V^+]$.

(ii) $\Gamma \overset{\text{LET}}{\vdash} M : A$ *if and only if* $\Gamma^* \overset{\text{LIN}}{\vdash} M^* : A^*$, *and*
$\Gamma \overset{\text{LET}}{\vdash} V : A$ *if and only if* $\Gamma^* \overset{\text{LIN}}{\vdash} V^+ : A^+$.

(iii) $M \xrightarrow{\text{LET}}\!\!\!\!\!\!\rightarrow N$ *if and only if* $M^* \xrightarrow{\text{LIN}}\!\!\!\!\!\!\rightarrow N^*$.

Proof. The proof of (i) and (ii) is similar to that for VAL. To prove (iii) in the forward direction, consider each possible reduction in LET. Reduction by rule (I) translates to

$$\begin{array}{ll} & (\text{let } z = !(\lambda y.\,\text{let } !x = y \text{ in } M^*) \text{ in } z)\ N^* \\ \xrightarrow[(\beta!)]{\text{LIN}} & (\lambda y.\,\text{let } !x = y \text{ in } M^*)\ N^* \\ \xrightarrow[(\beta\multimap)]{\text{LIN}} & \text{let } !x = N^* \text{ in } M^* \ . \end{array}$$

Reduction by rule (V) translates to

$$\begin{array}{ll} & \text{let } !x = !V^+ \text{ in } M^* \\ \xrightarrow[(\beta!)]{\text{LIN}} & M^*[x := V^+] \ . \end{array}$$

Reduction by rule (C) translates to

$$\begin{array}{ll} & (\text{let } !z = (\text{let } !x = L^* \text{ in } M^*) \text{ in } z)\ N^* \\ \xrightarrow[(!!)]{\text{LIN}} & (\text{let } !x = L^* \text{ in } (\text{let } !z = M^* \text{ in } z))\ N^* \\ \xrightarrow[(!\multimap)]{\text{LIN}} & \text{let } !x = L^* \text{ in } ((\text{let } !z = M^* \text{ in } z)\ N^*) \ . \end{array}$$

And reduction by rule (A) translates to (!!) in LIN.

The proof of the backward direction is similar to the corresponding proof for call-by-name. One shows that the set of LIN terms reachable from translations of terms in LET is produced by the grammar:

$$\begin{array}{lll} \text{Term image} & S, T & ::= !U \mid P\ S \mid \text{let } !x = S \text{ in } T \\ \text{Value image} & U & ::= x \mid \lambda y.\text{let } !x = y \text{ in } S \\ \text{Prefix} & P & ::= U \mid \text{let } !x = S \text{ in } P \ . \end{array}$$

An erasure ‡ that takes this set back into LET is given by:

$$\begin{array}{lll} x^\ddagger & \equiv x & \\ (\lambda y.\text{let } !x = y \text{ in } S)^\ddagger & \equiv \lambda x.S^\ddagger & \\ (!U)^\ddagger & \equiv U^\ddagger & \\ (P\ S)^\ddagger & \equiv P^\ddagger\ S^\ddagger & \\ (\text{let } !x = S \text{ in } D[x])^\ddagger & \equiv (D[S])^\ddagger & \\ (\text{let } !x = S \text{ in } T)^\ddagger & \equiv \text{let } x = S^\ddagger \text{ in } T^\ddagger & \text{if } T \not\equiv D[x] \\ (\text{let } !x = S \text{ in } P)^\ddagger & \equiv \text{let } x = S^\ddagger \text{ in } P^\ddagger & \text{if } P \not\equiv D[x] \end{array}$$

where D ranges over contexts in the language

$$D ::= [\,] \mid D\ S \mid \text{let } !y = D \text{ in } S \ .$$

It is straightforward to show that ‡ is a right inverse of $*$. Furthermore, an analysis of LIN-reductions shows that ‡ sends LIN-reduction sequences to LET-reduction sequences. The result then follows as in the call-by-name case. □

What is the relationship between VAL and LET? Clearly, every $\beta{\rightarrow}_v$-reduction in VAL can be simulated in LET by a pair of (I) and (V) reductions. That is, LET-reduction extends VAL-reduction; and it is not hard to show that this extension is conservative.

Proposition 5.4 LET *conservatively extends* VAL*: For all terms M and N in* VAL*, $M \xrightarrow{\text{VAL}}\!\!\!\!\twoheadrightarrow N$ if and only if $M \xrightarrow{\text{LET}}\!\!\!\!\twoheadrightarrow N$.* □

Syntactic domains As for LIN.

Typing judgements As for LIN, but changing Weakening as follows:

$$\text{Weakening}\ \frac{\Gamma \vdash M : B}{\Gamma,\ x : A \vdash M : B}$$

Reduction relation As for LIN, plus the following:

(!Weakening) $\quad$ let $!x = M$ in $N \to N$, $\quad$ if x is not free in N

Fig. 5. The affine lambda calculus AFF.

Syntactic domains As for LET.

Typing judgements As for LET.

Reduction relation As for LET, plus the following:

(G) $\quad$ let $x = M$ in $N \to N$, $\quad$ if x is not free in N

Translation As for LET, except into AFF rather than LIN.

Fig. 6. The call-by-need lambda calculus NEED.

6 Call-by-need

In the call-by-name translation, every function argument was surrounded by !, and so could be freely duplicated or discarded. In the call-by-value and call-by-let translations, only values are surrounded by !, and so non-values cannot be duplicated, but also cannot be discarded. The call-by-need calculus differs from these in that any term may be discarded if it is not needed, but a term should not be duplicated unless and until it has been reduced to a value. Thus, we wish to shift to a calculus where discarding (Weakening) is always allowed, but duplication (Contraction) remains under the strict control of the '!' connective.

Figure 5 presents the details of the resulting affine lambda calculus AFF. The types, terms, and contexts are the same as for the linear lambda calculus LIN. The type rules are also identical to those for LIN, with the exception that the rule Weakening of LIN, which allows weakening only on assumptions of the form $!x : !A$, is replaced by a rule which allows weakening on assumptions of the form $x : A$. This is strictly stronger than the previous rule, as the previous rule can be derived by combining this Weakening with Dereliction.

Reduction is defined by the rules $(\beta\multimap)$, $(\beta!)$, $(!\multimap)$, $(!!)$ together with a new rule (!Weakening). Again, this rule has a logical basis.

- Rule (!Weakening) arises when a !-E rule meets a Weakening rule, commuting one through the other.

The (!Weakening) rule cannot be valid in LIN, as it does not satisfy subject reduction. For instance, $y : !A, z : B \vdash$ let $!x = y$ in $z : B$, which is a valid judgement in both LIN and AFF, reduces to $y : !A, z : B \vdash z : B$, which is valid in AFF but not in LIN.

What is the operational impact of the switch to an affine calculus? Recall that in the linear calculus, one evaluates (let $!x = M$ in N) by first evaluating M to the form $!M'$ and then evaluating $N[x := M']$. This notion of evaluation is not suitable for the affine calculus, as it would violate the (!Weakening) rule. Instead, one must evaluate (let $!x = M$ in N) by binding M to a closure which is evaluated only if x is required during the evaluation of N. Thus, much of the call-by-need machinery is implicit in the (!Weakening) rule.

As before, the logical origin of the rules ensures the subject reduction property. Furthermore, confluence still holds.

Proposition 6.1 AFF *satisfies subject reduction.*

Proposition 6.2 AFF *is confluent.*

Proof. These are easily demonstrated by modifying the equivalent proofs for LIN. □

Figure 6 defines the call-by-need calculus NEED, which can be derived as a slight modification of the call-by-let calculus LET. The only change is the addition of a new reduction rule (G), which can be thought of as the surface manifestation of the underlying reduction (!Weakening). The name (G) stands for 'Garbage collection'.

Proposition 6.3 NEED *satisfies subject reduction.*

Proposition 6.4 NEED *is confluent.*

Proof. Subject reduction is easy, requiring just one more case than LET. We have previously published a proof of confluence for call-by-need [24]. □

By design, if $*$ is now viewed as taking the call-by-need calculus NEED into the affine calculus AFF, then the transformation result still holds.

Proposition 6.5 (Call-by-need translation) *The translation $*$ from* NEED *to* AFF *preserves substitution of values, preserves types, and preserves reductions.*

(i) $(M[x := V])^* \equiv M^*[x := V^+]$.

(ii) $\Gamma \overset{\text{NEED}}{\vdash} M : A$ *if and only if* $\Gamma^* \overset{\text{AFF}}{\vdash} M^* : A^*$, *and*
$\Gamma \overset{\text{NEED}}{\vdash} V : A$ *if and only if* $\Gamma^* \overset{\text{AFF}}{\vdash} V^+ : A^+$.

(iii) $M \xrightarrow{\text{NEED}}\!\!\!\!\rightarrow N$ *if and only if* $M^* \xrightarrow{\text{AFF}}\!\!\!\!\rightarrow N^*$.

Proof. The proof is as before, noting that the reduction (G) in NEED translates to (!Weakening) in AFF. □

We conclude this section with some results on the observational behavior of the calculi we have studied. Some general terminology: Assume two calculi, say X and Y, extended with constants and primitives as outlined in Section 7.3. Assume that calculus X is bigger than calculus Y, that is, that every term of X is also a term of Y, but not necessarily conversely. Also assume that both systems have the same constants and primitives, and the same delta rules defining the value of these. We say that calculus X is *observationally equivalent* to calculus Y if for every term M in calculus Y, and for every constant c, we have $M \xrightarrow{X}\!\!\!\!\rightarrow c$ if and only if $M \xrightarrow{Y}\!\!\!\!\rightarrow c$. The previous section's result on conservative extension of LET over VAL has the following corollary.

Proposition 6.6 LET *is observationally equivalent to* VAL. □

Now, the addition of the single law (G) changes the situation significantly.

Proposition 6.7 NEED *is observationally equivalent to* NAME.

Proof. We showed in previous work [24], that if M is a term and V is a value (both in NAME) such that $M \xrightarrow{\text{NAME}} V$, then there exist variables $x_1, \ldots x_k$, terms $M_1, \ldots M_k$ and a value V' such that

$$M \xrightarrow{\text{NEED}} \text{let } x_1 = M_1 \text{ in } \cdots \text{let } x_k = M_k \text{ in } V'.$$

The proof can be modified to show that if $V \equiv c$ then $V' \equiv c$. By repeatedly applying (G), we get that $M \xrightarrow{\text{NAME}}\!\!\!\!\rightarrow c$ implies $M \xrightarrow{\text{NEED}}\!\!\!\!\rightarrow c$. The reverse implication is easy. □

7 Extensions

In this section we discuss various extensions of our results. It is straightforward to extend the translations for products (Section 7.1) and for constants and primitives (Section 7.3), but an appropriate translation of sums is less clear (Section 7.2). Recursion also presents problems (Section 7.4). Finally, our results are easily transferred to an untyped framework (Section 7.5).

7.1 Products

The extensions for products are straightforward.

Call-by-name

$$(A \times B)^\circ \equiv A^\circ \;\&\; B^\circ$$

$$(M, N)^\circ \equiv (M^\circ, N^\circ)$$

$$\begin{aligned}(\text{fst } M)^\circ &\equiv \text{fst } M^\circ \\ (\text{snd } M)^\circ &\equiv \text{snd } M^\circ\end{aligned}$$

Call-by-value

$$(A \times B)^+ \equiv A^+ \;\&\; B^+$$

$$(V, W)^+ \equiv (V^+, W^+)$$

$$\begin{aligned}(\text{fst } M)^* &\equiv \text{fst}\,(\text{let } !z = M \text{ in } z) \\ (\text{snd } M)^* &\equiv \text{snd}\,(\text{let } !z = M \text{ in } z)\end{aligned}$$

In the call-by-name translation, & is the additive product of linear logic; we use the notations (M, N), $(\text{fst } M)$, and $(\text{snd } M)$ to stand both for $\times$ introduction and elimination in the lambda calculus, and for & introduction and elimination in the linear lambda calculus. The call-by-value translation also uses this additive product. In contrast, Girard's version of the latter translation [11] defines

$$(A \times B)^* \equiv A^* \otimes B^* \ ,$$

which uses $\otimes$, the multiplicative (or tensor) product of linear logic. These two products are related by the isomorphism $!(A \mathbin{\&} B) \simeq (!A) \otimes (!B)$. Thus our translation is isomorphic to Girard's, since

$$A^* \otimes B^* \equiv (!A^+) \otimes (!B^+) \simeq !(A^+ \mathbin{\&} B^+) \equiv !(A \times B)^+ \equiv (A \times B)^* \ .$$

In call-by-value, the components of a pair are restricted to values. The more general construct (M, N) may be added to the call-by-value language by defining it as an abbreviation for $(\lambda x.\, \lambda y.\, (x, y))\; M\; N$. Restricting pairing to values makes the translation easier to define, and corresponds to a restriction on pairs that arises naturally in call-by-need calculi [2,17].

7.2 Sums

Extending the call-by-name translation to sums is straightforward. At the type level, following Girard, the translation is defined by:

$$(A + B)^\circ \equiv (!A^\circ) \oplus (!B^\circ) \ .$$

Here $\oplus$ is the additive (or direct) sum of linear logic; the term translation follows immediately.

Extending the call-by-value translation to sums is more problematic. At the type level, again following Girard, we would expect a definition satisfying the isomorphism

$$(A + B)^* \simeq A^* \oplus B^* \ .$$

As with products it is desirable to re-express the left hand side in the form $!(A + B)^+$. Unfortunately, it is not clear how to choose a C such that $(!A) \oplus (!B) \simeq !C$, and as a result the treatment of sums is less clear.

7.3 Constants and primitives

All of the lambda calculi discussed may be straightforwardly extended by the addition of constants c and primitive applications $p\, M_1 \cdots M_k$ of arity k. We also add a suitable reduction rule for each primitive to each lambda calculus.

$$(\delta) \qquad p\, c_1 \cdots c_k \rightarrow \mathrm{apply}(p, c_1, \ldots, c_k)$$

Following Plotkin [25], 'apply' is a function that yields a closed term for a given primitive and constants.

We extend the translations as follows.

Call-by-name

$$\begin{aligned} c^\circ &\equiv c \\ (p\, M_1 \cdots M_k)^\circ &\equiv p\, M_1^\circ \cdots M_k^\circ \end{aligned}$$

Call-by-value

$$\begin{aligned} c^+ &\equiv c \\ (p\, M_1 \cdots M_k)^* &\equiv \mathrm{let}\ x_1 = M_1^*\ \mathrm{in}\ \cdots \mathrm{let}\ x_k = M_k^*\ \mathrm{in}\ p\, x_1 \cdots x_k \end{aligned}$$

For the translation to be valid, the interpretation of primitives in the linear calculus must be related to the interpretations in both the call-by-name and call-by-value calculi:

$$\mathrm{apply}_{\mathrm{LIN}}(p, c_1, \ldots, c_k) \equiv (\mathrm{apply}_{\mathrm{NAME}}(p, c_1, \ldots, c_k))^{\circ}$$
$$\mathrm{apply}_{\mathrm{LIN}}(p, c_1, \ldots, c_k) \equiv (\mathrm{apply}_{\mathrm{VAL}}(p, c_1, \ldots, c_k))^{*} \ .$$

Again, the translation results carry through for the extended calculi.

This extension to constants and primitives is particularly useful because it supports the simple version of the operational equivalence theorems given in Section 6. A desire to avoid constants led to a somewhat more complex operational equivalence theorem in our previous work [2].

7.4 Recursion

Adding recursion to the translation is more difficult. There are two basic problems.

First, it is not clear what typing to use for a recursion operator in the linear calculus. The call-by-name translation suggests that corresponding to the typing

$$\mathrm{Rec}\ \frac{\Gamma,\ x : A \vdash M : A}{\Gamma \vdash \mathrm{fix}\ x.\, M : A}$$

in NAME, we should take

$$\mathrm{Rec}\ \frac{!\Gamma,\ !x : !A \vdash M : A}{!\Gamma \vdash \mathrm{fix}\ x.\, M : A}$$

in LIN. But the call-by-value translation suggests that corresponding to the typing

$$\mathrm{Rec}\ \frac{\Gamma,\ x : A \vdash V : A}{\Gamma \vdash \mathrm{fix}\ x.\, V : A}$$

in VAL, we should take

$$\mathrm{Rec}\ \frac{!\Gamma,\ !x : !A \vdash M : !A}{!\Gamma \vdash \mathrm{fix}\ x.\, M : !A}$$

in LIN. It is not clear how to reconcile these two choices.

Second, adding recursion to call-by-need poses additional difficulties, as noted in our previous work [2]. In order to maintain the sharing typical of call-by-need implementations of recursion, it appears necessary to shift from single, nonrecursive bindings (let $x = M$ in N) to multiple, recursive bindings (letrec $x_1 = M_1, \ldots x_k = M_k$ in N). On the surface, this change appears to pose a problem only because the expected increase in the size of the calculus is awkward. In fact, a serious complication arises from a problem pointed out by Ariola and Klop [4]: with 'letrec' one must restrict the selection of redexes, or otherwise the calculus will not be confluent. In the presence of this constraint, it is not immediately clear how to adapt our results.

7.5 Untyped languages

The confluence and translation results do not depend on types, and so carry through directly for untyped version of these calculi. The only trickiness is that we have used the type rules of the linear lambda calculus to enforce the constraint that variables bound by linear assumptions appear linearly, and that all free variables of a term $!M$ are bound by ! assumptions. It is easy to enforce the same constraints without recourse to types, for instance by using well-formedness judgements. (These can be derived from our typing judgements by simply erasing all the types!)

8 Conclusions

We have shown that call-by-name, call-by-value, call-by-let, and call-by-need can be explained by translations into linear and affine lambda calculi. These transformations begin to provide a logical explanation of call-by-value and call-by-need in the style of the Curry-Howard isomorphism. Many interesting questions remain, and we conclude this paper by mentioning three of these.

8.1 Standard reduction

Our previous work on call-by-need [2] uses a slightly different version of the (V) rule,

$$\text{(V)} \qquad \text{let } x = M \text{ in } C[x] \to C[V],$$

where C is an arbitrary context. If the $(\beta!)$ rule of the linear and affine calculi is changed in a similar way, then the translation is still preserved.

All three systems of reduction, NAME, VAL, and NEED possess notions of 'standard' reduction: normal order for call-by-need, applicative order for call-by-value, and a more complex order for call-by-need which is described in our previous work. In each case, the value of the standard reductions is that they correspond closely to the operational semantics of the language in question. One reason for adopting the variant (V) rule above is that it enables this close correspondence in the case of call-by-need.

We conjecture that there is also a standard reduction order for the linear and affine calculi LIN and AFF. We have not considered the question of whether the translations preserve standard reductions, or bear some other interesting relation to them.

8.2 Eta rules

It is common to include an $(\eta \to)$ rule in the call-by-name calculus, and an $(\eta \to_v)$ rule in the call-by-value calculus.

$$\begin{array}{lll} (\eta \to) & \lambda x.\,(M\ x) \to M, & \text{if } x \text{ not free in } M \\ (\eta \to_v) & \lambda x.\,(V\ x) \to V, & \text{if } x \text{ not free in } V \end{array}$$

Similarly, one might add (η–$\circ$) and (η!) rules to the linear lambda calculus.

$$\begin{array}{lll} (\eta{\multimap}) & \lambda x.\,(M\ x) \rightarrow M, & \text{if } x \text{ not free in } M \\ (\eta!) & \text{let } !x = M \text{ in } !x \rightarrow M & \end{array}$$

The second of these is similar to the right unit law (id) in Moggi's computation lambda calculus [21,22] (see the next section below). Just as a (β) rule simplifies a logical introduction followed by the corresponding logical elimination, so each of these (η) rules simplifies a logical elimination followed by the corresponding logical introduction.

One would expect that the translation $\circ$ from NAME to LIN should preserve the (η) rule, that the translation $*$ from VAL to LIN should preserve the (η_v) rule, and that the translation $*$ from NEED to AFF should preserve the (η) rule. None of these are the case, which suggests that perhaps more reduction rules should be added to LIN and AFF. Various rules seem possible, but so far we have not found rules with the same logical resonance as those considered in this paper.

8.3 Equality and Moggi's computational lambda calculus

We have shown our various translations are both sound and complete for various notions of reduction. What about equality?

Since $M = N$ if and only if there is a L such that $M \twoheadrightarrow L$ and $N \twoheadrightarrow L$, it is clear that any translation that is sound for reduction must also be sound for equality. So the only question of interest is completeness.

The call-by-name translation is complete for equality. This is easily seen from the following fact: for every NAME term L and every LIN term M, if $L^\circ \xrightarrow{\text{LIN}}\!\!\!\!\twoheadrightarrow M$, then there exists a NAME term N such that $M \xrightarrow{\text{LIN}}\!\!\!\!\twoheadrightarrow N^\circ$.

The other translations are not complete for equality. For instance, the VAL terms $(M\ N)$ and $((\lambda z.\, z\ N)\ M)$ are not equal in VAL, but their translations are equal in LIN. The same example adopts to LET and NEED.

It is reasonable to ask if more laws could be added to VAL, LET, or NEED so that the corresponding translations are sound and complete for equality as well as reduction.

A hint as to a suitable extension is provided by Moggi's computational lambda calculus COMP [21,22], as shown in Figure 7. (The version of the calculus shown here is based on the untyped reduction calculus, which Moggi calls λ_c, and which appears in his technical report [21] but not his LICS paper [22].) The terms of this theory are the same as for LET; though the grammar now distinguishes non-values E as well as values V. This system satisfies subject reduction, and Moggi shows that it is confluent. The system is designed so that it is strongly normalising even without types, apart from rule (β_v).

It is not hard to show that the equalities of LET are properly contained in the equalities of COMP; that is, $M \overset{\text{LET}}{=} N$ implies $M \overset{\text{COMP}}{=} N$, but not conversely. Furthermore, the reductions of LET and the reductions of COMP are incomparable; that is, $M \xrightarrow{\text{LET}}\!\!\!\!\twoheadrightarrow N$ does not imply $M \xrightarrow{\text{COMP}}\!\!\!\!\twoheadrightarrow N$, nor is the

Syntactic domains

Types	A, B, C	$::= Z \mid A \to B$
Terms	L, M, N	$::= V \mid E$
Values	V	$::= x \mid \lambda x.\, M$
Non-values	E	$::= M\ N \mid \text{let } x = M \text{ in } N$

Typing judgements As for LET.

Reduction relation

(β_v)	$(\lambda x.\, M)\, V$	$\to M[x := V]$	
(η_v)	$\lambda x.\, (V\ x)$	$\to V,$	if x not free in V
(let_v)	$\text{let } x = V \text{ in } M$	$\to M[x := V]$	
(id)	$\text{let } x = M \text{ in } x$	$\to M$	
(comp)	$\text{let } y = (\text{let } x = L \text{ in } M) \text{ in } N$	$\to \text{let } x = L \text{ in } (\text{let } y = M \text{ in } N)$	
(let.1)	$E\, M$	$\to \text{let } z = E \text{ in } z\, M$	
(let.2)	$V\ E$	$\to \text{let } x = E \text{ in } V\ x$	

Fig. 7. Moggi's computational lambda calculus COMP.

converse true. It is interesting open question whether there is an extension of LET that has the same equalities as COMP. It is a further interesting question to know if there is an extension of LIN such that the encoding of the extended LET into the extended LIN via $*$ is sound and complete.

This question brings us full circle. Plotkin showed that the continuation passing style (CPS) translation from VAL into itself is sound but not complete [25]. Moggi designed COMP to be sound and complete for the monad translation (which generalises CPS) [21], and Sabry and Felleisen verified that the CPS translation from COMP into VAL is both sound and complete [31], thus answering the question implicitly raised by Plotkin.

It can be seen that the questions raised here about complete extensions of LET are in a similar vein. It is not clear if the translations into linear lambda calculus described here will have the same value as the translations into continuation passing style described by Plotkin. But if our answers are not as good, perhaps we can at least claim to be asking the right sort of questions!

References

[1] S. Abramsky, Computational interpretations of linear logic. *Theoretical Computer Science* 111:3–57, 1993.

[2] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Principles of Programming Languages (POPL)*, ACM Press, San Francisco, California, January 1995.

[3] Z. M. Ariola and M. Felleisen, The call-by-need lambda calculus. Technical report CIS-TR-94-23, Department of Computer Science, University of Oregon, October 1994.

[4] Z. M. Ariola and J. W. Klop, Cyclic lambda graph rewriting. In *Logic in Computer Science (LICS)*, Paris, France, 1994.

[5] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Computer Science*, North-Holland Publishing Company, 1981.

[6] N. Benton, G. Bierman, V. de Paiva, and M. Hyland, Type assignment for intuitionistic linear logic. Technical report 262, Computing Laboratory, University of Cambridge, August 1992.

[7] P. N. Benton, Strong normalisation for the linear term calculus. Technical report 305, Computing Laboratory, University of Cambridge, July 1993. To appear in *Journal of Functional Programming.*

[8] G. Bierman, What is a categorical model of intuitionistic linear logic? Technical report 333, Computing Laboratory, University of Cambridge, April 1994. To appear in *Proceedings of the Conference on Typed Lambda Calculus and Applications*, Springer Verlag LNCS, April 1995.

[9] G. Bierman, On Intuitionistic Linear Logic. Technical report 346, Computing Laboratory, University of Cambridge, August 1994.

[10] H. P. Barendregt and K. Hemerik, Types in Lambda Calculus and Programming Languages. In *European Symposium on Programming (ESOP)*, p. 1–35. Springer-Verlag, Lecture Notes in Computer Science 432, 1990.

[11] J.-Y. Girard, Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[12] J.-Y. Girard, On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[13] G. Gonthier, M. Abadi, and J.-J. Levy, The geometry of optimal lambda reduction. In *Principles of Programming Languages (POPL)*, ACM Press, Albuquerque, New Mexico, January 1992.

[14] S. Holmström, A linear functional language. Draft paper, Chalmers University of Technology, 1988.

[15] B. Jacobs, Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69: 73–106, 1994.

[16] Y. Lafont, The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[17] J. Launchbury, A natural semantics for lazy evaluation, *Symposium on Principles of Programming Languages*, ACM Press, Charleston, South Carolina, January 1993.

[18] P. Lincoln and J. Mitchell, Operational aspects of linear lambda calculus. In *7'th Symposium on Logic in Computer Science*, IEEE Press, Santa Cruz, California, June 1992.

[19] I. Mackie, Lilac: a functional programming language based on linear logic. Master's Thesis, Imperial College London, 1991.

[20] I. Mackie, The Geometry of Implementation. Doctoral Thesis, Imperial College London, 1994.

[21] E. Moggi, Computational lambda-calculus and monads. Technical report ECS-LFCS-88-66, Laboratory for the Foundations of Computer Science, University of Edinburgh, October 1988.

[22] E. Moggi, Computational lambda-calculus and monads. In *4'th Symposium on Logic in Computer Science*, IEEE Press, Asilomar, California, June 1989.

[23] J. Maraist, M. Odersky, D. N. Turner, and P. Wadler, Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. Technical report, Fakultät für Informatik, Universität Karlsruhe and Department of Computing Science, University of Glasgow, March 1995.

[24] J. Maraist, M. Odersky, and P. Wadler, The call-by-need lambda calculus, Technical report, Fakultät für Informatik, Universität Karlsruhe, and Department of Computing Science, University of Glasgow, October 1994.

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

[26] G. D. Plotkin, Private communication. (During the 1993 MFPS, even!)

[27] J. C. Reynolds, The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–248, November 1993.

[28] S. R. della Rocca and L. Roversi, Lambda calculus and intuitionistic linear logic. Manuscript, July 1994. (Available from L. Roversi, University of Pisa, `rover@di.unipi.it`.)

[29] H. Schellinx, The noble art of linear decorating. Ph.D. dissertation, Institute for Logic, Language, and Computation, University of Amsterdam, 1994.

[30] R. A. G. Seely, Linear logic, $*$-autonomous categories, and cofree coalgebras. In *Categories in Computer Science and Logic*, June 1989. AMS Contemporary Mathematics 92.

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

[32] A. S. Troelstra, *Lectures on Linear Logic.* CSLI Lecture Notes, 1992.

[33] D. N. Turner, P. Wadler, and C. Mossin, Once upon a type. In *Functional Programming and Computer Architecture (FPCA)*, San Diego, California, June 1995.

[34] P. Wadler, Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, North Holland, April 1990.

[35] P. Wadler, Is there a use for linear logic? In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, New Haven, Connecticut, ACM Press, June 1991.

[36] P. Wadler, A syntax for linear logic. *Ninth International Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana, April 1993. Springer Verlag, LNCS 802.

[37] P. Wadler, A taste of linear logic. *Mathematical Foundations of Computer Science*, Gdansk, Poland, August 1993. Springer Verlag, LNCS 711.