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

John Maraist Martin Odersky
University of Karlsruhe†

David N. Turner Philip Wadler
University of Glasgow‡

10 November 1994

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's 'Call-by-name, call-by-value, and the lambda calculus' [Plo75] demonstrated how the two different calling mechanisms could be explained by two different translations into continuation passing style. At the time he wrote, the call-by-value translation was widely appreciated, but the call-by-name translation was less well known. The call-by-value translation had been reinvented several times [Rey93], while the call-by-name translation is credited by Plotkin to a private communication from Reynolds.

While we hesitate to compare our work to Plotkin's, our goal here is similar. We demonstrate how the two different calling mechanisms can 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 [Gir87], the first based on mapping $A \to B$ into $!A \multimap B$, and the second based on mapping it into $!(A \multimap B)$. The first translation is frequently studied, and known to correspond to call-by-name; the second translation is less frequently studied, and its correspondence to call-by-value is not completely obvious. As evidence of the latter we note that Girard once suggested [Gir92] that the second translation might be connected to Abramsky's lazy lambda calculus [Abr92], although Abramsky's calculus is call-by-name.

A number of different lambda calculi based on linear logic have been proposed, including work by Lafont [Laf88], Holmström [Hol88], Wadler [Wad90, Wad91, Wad93a, Wad93b], Abramsky [Abr93], Mackie [Mac91], Lincoln and Mitchell [LM92], Troelstra [Tro92], Benton, Bierman, dePaiva, and Hyland [BBdPH92], and della Rocca and Roversi [RR94]. Various embeddings of

*Appears as Technical Report #5/95, Fakultät für Informatik, Universität Karlsruhe, and as Technical Report FP-94-12, Department of Computing Science, University of Glasgow.

†Institut für Programmstrukturen, Universität Karlsruhe, Postfach 6980, 76128 Karlsruhe, F.R. Germany. E-mail: `maraist,odersky@ira.uka.de`.

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

intuitionistic logic into linear logic have also been studied, including work by Girard [Gir87], Troelstra [Tro92], and Schellinx [Sch94]. The linear lambda calculus used in this paper is a minor refinement of one previously presented by Wadler [Wad93a, Wad93b], which is based on Girard's successor to linear logic, the Logic of Unity [Gir93]. A similar calculus has been devised by Plotkin [Plo93].

Corresponding to Girard's first translation we define a mapping $*$ 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 $M^* \xrightarrow{\text{LIN}}\!\!\!\!\rightarrow N^*$, and complete, in that the converse also holds. Corresponding to Girard's second translation we define a second mapping $\dagger$ 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 $M^\dagger \xrightarrow{\text{LIN}} N^\dagger$. We believe these soundness and completeness results to be new; showing soundness is straightforward, while completeness is somewhat trickier to establish.

Our original motivation for studying these questions came from an interest not in call-by-name or in call-by-value, but instead in the call-by-need calculus, which was recently proposed by Ariola, Felleisen, Maraist, Odersky, and Wadler [AF94, MOW94, AFMOW95]. 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 set of reductions, large enough to guarantee observational equivalence with the call-by-name calculus. But like call-by-value, call-by-need 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 [Jac92].

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 $\dagger$ 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 $\dagger$ 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 $\dagger$ translation relates reductions in each to reductions of the linear calculus which have a clear logical explanation.

An additional contribution of this work is that we confirm our linear calculus to be confluent. Although many linear calculi have been described, relatively few possess claims of confluence, a notable exception being the work of della Rocca and Roversi [RR94].

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.

2 Linear lambda calculus

We begin with a linear lambda calculus similar to one introduced previously by Wadler [Wad93a, Wad93b].

In many presentations of logic a key role is played by the structural rules: contraction provides

Syntactic domains

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

Typing judgements

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

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

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

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

Reduction relation

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

Figure 1: The linear lambda calculus LIN.

the only way to duplicate an assumption, while weakening provides the only way to discard one. In linear logic [Gir87], the presence of contraction or weakening is revealed in a formula by the presence of the 'of course' connective, written '!'. The Logic of Unity [Gir93] 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. 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 be introduced only by the !-E rule. The only thing that may be done with such an 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, in the !-I rule the context $!\Gamma$ contains only assumptions of the form $!x : !A$. 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 $\to$ is defined by the union of (the compatible closure of) the four given rules. We assume free and bound variables of a term are distinct in order to avoid capture; for

instance, in rules (!$\multimap$) and (!!), variable x cannot appear free in term N.

Some general notation: We write $\twoheadrightarrow$ for the reflexive and transitive closure of $\rightarrow$. We write $=$ for the reflexive, symmetric, and transitive closure of $\rightarrow$, 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.

The action of reductions on proof trees is written out in full in Appendix A.

To evaluate a term of the form (let $!x = M$ in N), it is sufficient to first reduce M to a term of the form $!M'$, and then substitute M' for x in N. 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 general terminology: A calculus satisfies the *subject reduction* property if whenever $\Gamma \vdash M : A$ and $M \rightarrow N$ 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. LIN satisfies subject reduction.

Proposition. LIN is confluent.

Proof. The subject reduction result is inherent in what has been said above; full details appear in Appendix A. For confluence, the proof is similar to our proof of confluence for the call-by-need lambda calculus [MOW94], which uses Barendregt's technique of marked and weighted redexes with a norm for weighted terms [Bar81]. We show ($\beta\multimap$) and ($\beta!$, !$\multimap$, !!) to be each individually Church-Rosser. In the former subsystem, the method is exactly that of Barendregt for (β). In the latter, we use the norm to show strong normalization, and a tedious but straightforward case analysis to show the weak Church-Rosser property; these two together imply a fully Church-Rosser system. Finally we show that multistep ($\beta\multimap$) and ($\beta!$, !$\multimap$, !!) reductions commute, and with subject reduction we have the result for the full system. □

We conclude this section with a few words about the relation of our linear lambda calculus to other formulations. For a fuller description of this relationship, see our previous work [Wad93a, Wad93b].

The formulation given here is based on the Logic of Unity [Gir93], 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,

Syntactic domains

$$\begin{array}{lrcl} \text{Types} & A, B, C & ::= & Z \mid A \to B \\ \text{Terms} & L, M, N & ::= & V \mid M\ N \\ \text{Values} & V, W & ::= & x \mid \lambda x.\, M \end{array}$$

Typing judgements

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

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

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

Reduction relation

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

Translation

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

Figure 2: The call-by-name lambda calculus NAME.

or satisfy 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.

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.

The call-by-name calculus has one reduction rule, $(\beta\to)$. This calculus of course satisfies the usual subject reduction and confluence results.

Proposition. NAME satisfies subject reduction.

Proposition. NAME is confluent.

Translation $*$ 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.

Proposition. *Call-by-name translation.* The translation $*$ from NAME to LIN preserves substitution, types, and reductions.

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

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

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

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. To prove (iii) in the forward direction, we observe that reduction $(\beta\!\rightarrow)$ translates to

$$(\lambda y.\, \text{let } !x = y \text{ in } M^*)\, !N^* \quad \xrightarrow[(\beta\multimap)]{\text{LIN}} \quad \text{let } !x = !N^* \text{ in } M^* \xrightarrow[(\beta!)]{\text{LIN}} \quad M^*[x := N^*] \ .$$

For the backward direction, we consider the grammar

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

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

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

It is straighforward to show:

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

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

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

It is not hard to show that if $L^* \xrightarrow{\text{LIN}}\!\!\!\!\rightarrow M$, then there exists a NAME term N such that $M \xrightarrow{\text{LIN}}\!\!\!\!\rightarrow N^*$, and hence we also have that $M \overset{\text{NAME}}{=} N$ if and only if $M^* \overset{\text{LIN}}{=} N^*$.

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 one reduction rule, a restriction of the $(\beta\!\rightarrow)$ rule of call-by-name to the case when the function argument is a value. Again, the usual subject reduction and confluence results apply.

Proposition. VAL satisfies subject reduction.

Proposition. VAL is confluent.

Translation $\dagger$ takes types, terms, and contexts A, M, Γ of VAL to types, terms, and contexts $A^\dagger$, $M^\dagger$, $\Gamma^\dagger$ of LIN. There is also an auxiliary mapping $\ddagger$ from types and values A, V of VAL to

Syntactic domains As for NAME.

Typing judgements As for NAME.

Reduction relation

$$(\beta\to_v) \qquad (\lambda x.\, M)\, V \quad \to \quad M\,[x := V]$$

Translation

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

Figure 3: The call-by-value lambda calculus VAL.

types and terms $A^\ddagger$, $V^\ddagger$ 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 $V^\dagger \equiv !V^\ddagger$, so function arguments are forced to reduce to values.

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

(i) $(M\,[x := V])^\dagger \equiv M^\dagger[x := V^\ddagger]$.

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

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

Proof. Prove (i) and (ii) similarly to NAME. To prove (iii) in the forward direction, observe that $(\beta\to_v)$ translates to

$$\begin{array}{rcl}
(\text{let } !z = !(\lambda y.\, \text{let } !x = y \text{ in } M^\dagger) \text{ in } z)\, !V^\ddagger & \xrightarrow[(\beta!)]{\text{LIN}} & (\lambda y.\, \text{let } !x = y \text{ in } M^\dagger)\, !V^\ddagger \\
& \xrightarrow[(\beta\multimap)]{\text{LIN}} & \text{let } !x = !V^\ddagger \text{ in } M^\dagger \\
& \xrightarrow[(\beta!)]{\text{LIN}} & M^\dagger[x := V^\ddagger]\ .
\end{array}$$

The backward implication 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. □

It is easily shown that $M \overset{\text{VAL}}{=} N$ implies $M^\dagger \overset{\text{LIN}}{=} N^\dagger$, but the converse is not true. For instance, the VAL terms $((\lambda x.\, M)L)\ N$ and $(\lambda x.\, M\ N)\ L$ are not equal in VAL, but their translations are equal in LIN.

Syntactic domains As for VAL, plus the following:

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

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

Translation As for VAL, plus the following:

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

Figure 4: The call-by-let calculus LET.

5 Call-by-let

It is common to define (let $x = M$ in N) $\equiv (\lambda x.\, N)\, M$. Say we add 'let' as a new term form to the call-by-value lambda calculus. Since

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

it is reasonable to define the first term to be equal to the last. We refer to this new calculus as the call-by-let calculus LET. One might ask what set of reductions is required in order to guarantee that $M \xrightarrow{\text{LET}}\!\!\!\!\rightarrow N$ if and only if $M^\dagger \xrightarrow{\text{LIN}}\!\!\!\!\rightarrow N^\dagger$. The answer is that a somewhat larger set of reductions is now required.

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 compatible closure of) the four rules (I), (V), (C), (A), which stand for 'Introduce', 'Value', 'Commute', and 'Associate'. As before, subject reduction and confluence both hold.

Proposition. LET satisfies subject reduction.

Proposition. LET is confluent.

Proof. Subject reduction can be verified straightforwardly. Confluence is easily shown by modifying the our previous results for call-by-need [MOW94]. □

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

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

(i) $(M[x := V])^\dagger \equiv M^\dagger[x := V^\ddagger]$.

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

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

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

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

Reduction by rule (V) translates to

$$\text{let } !x = !V^\ddagger \text{ in } M^\dagger \xrightarrow[(\beta!)]{\text{LIN}} M^\dagger[x := V^\ddagger] \ .$$

Reduction by rule (C) translates to

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

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:

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

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

$$\begin{array}{rcll} x^\# & \equiv & x & \\ (\lambda y.\text{let } !x = y \text{ in } S)^\# & \equiv & \lambda x.S^\# & \\ & & & \\ (!U)^\# & \equiv & U^\# & \\ (P\,S)^\# & \equiv & P^\#\,S^\# & \\ (\text{let } !x = S \text{ in } D[x])^\# & \equiv & (D[M'])^\# & \\ (\text{let } !x = S \text{ in } T)^\# & \equiv & \text{let } x = S^\# \text{ in } T^\# & T \not\equiv D[x] \\ (\text{let } !x = S \text{ in } P)^\# & \equiv & \text{let } x = S^\# \text{ in } P^\# & P \not\equiv D[x]\end{array}$$

where D ranges over contexts in the language

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

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 LETΓ 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. Furthermore, one can show that the extension is conservative:

Proposition. For all $M, N \in$ VAL, $M \xrightarrow{\text{VAL}}\!\!\!\!\rightarrow N$ if and only if $M \xrightarrow{\text{LET}}\!\!\!\!\rightarrow N$.

Proof. Again, the direction $M \xrightarrow{\text{VAL}}\!\!\!\!\rightarrow N \Longrightarrow M \xrightarrow{\text{LET}}\!\!\!\!\rightarrow N$ is trivial.

The reverse direction relies on the intuitive idea that if a LET-reduction sequence results in a VAL-term, all let-structures manipulated by (A) and (C) contractions must be (V)-contracted away. As a contrast, note that the following does *not* hold:

$$\begin{array}{ccc} M & \xrightarrow{(I)} & M' \\ {\scriptstyle(\beta\rightarrow_v)}\downarrow & & \downarrow{\scriptstyle(I,V,C,A)} \\ N & \xrightarrow{(I)} & N' \end{array} \ .$$

The details of the proof are in Appendix B. □

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:

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

Figure 5: The affine lambda calculus AFF.

Syntactic domains As for LET.

Typing judgements As for LET.

Reduction relation As for LET, plus the following:

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

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

Figure 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 compatible closure of) 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.

As required, the effect of rule (!Weakening) is to allow the term M in let $!x = M$ in N to be discarded if it is not needed, without first reducing M to the form $!M'$.

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

Proposition. AFF satisfies subject reduction.

Proposition. 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. NEED satisfies subject reduction.

Proposition. 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 [MOW94]. □

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. *Call-by-let translation.* The translation † from NEED to AFF preserves substitution of values, preserves types, and preserves reductions.

(i) $(M[x := V])^\dagger \equiv M^\dagger[x := V^\ddagger]$.

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

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

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 one have the following corollary.

Proposition. LET is observationally equivalent to VAL.

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

Proposition. NEED is observationally equivalent to NAME.

Proof. We showed in previous work [MOW94], that if M is a term and V is a value (both in NAME) such that $M \xrightarrow{\text{NAME}} V$, then there exist 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 the same result with $V \equiv V'$. By taking $V \equiv c$ and 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

We sketch here how our results extend to products, and to constants and primitives. We explain why extension to sums or to recursion are more difficult. We also note that the results extend easily to an untyped language.

7.1 Products

The extensions for products and for constants and primitives are straightforward, and are shown in Figures 7, 8, and 9. We discuss products in this section, and constants and primitives in the next.

The linear calculus may be extended with the additive product '&' of linear logic, as shown in Figure 7. The call-by-name translation of ordinary products $\times$ into additive product & is

Syntactic domains

$$\begin{array}{llll} \text{Types} & A, B, C & ::= & \cdots \mid A \,\&\, B \\ \text{Terms} & L, M, N & ::= & \cdots \mid (M, N) \mid \text{fst}\, M \mid \text{snd}\, M \mid c \mid p\, M_1 \,\cdots\, M_k \end{array}$$

Typing judgements

$$\&\text{-I}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \,\&\, B} \qquad \&\text{-E}_1\ \frac{\Gamma \vdash M : A \,\&\, B}{\Gamma \vdash \text{fst}\, M : A} \qquad \&\text{-E}_2\ \frac{\Gamma \vdash M : A \,\&\, B}{\Gamma \vdash \text{snd}\, M : B}$$

$$\text{Const}\ \frac{}{\vdash c : \text{type}_{\text{LIN}}(c)} \qquad \text{Prim}\ \frac{\Gamma_1 \vdash M_1 : \text{type}_{\text{LIN}}(p, 1) \quad \cdots \quad \Gamma_k \vdash M_k : \text{type}_{\text{LIN}}(p, k)}{\Gamma_1, \ldots, \Gamma_k \vdash p\, M_1 \,\cdots\, M_k : \text{type}_{\text{LIN}}(p, 0)}$$

Reduction relation

$$\begin{array}{llll} (\beta\&_1) & \text{fst}\ (M, N) & \to & M \\ (\beta\&_2) & \text{snd}\ (M, N) & \to & N \\ (!\&_1) & \text{fst}\ (\text{let}\ !x = L\ \text{in}\ M) & \to & \text{let}\ !x = L\ \text{in}\ (\text{fst}\ M) \\ (!\&_2) & \text{snd}\ (\text{let}\ !x = L\ \text{in}\ M) & \to & \text{let}\ !x = L\ \text{in}\ (\text{snd}\ M) \\ (\delta) & p\, c_1 \,\cdots\, c_k & \to & \text{apply}_{\text{LIN}}(p, c_1, \ldots, c_k) \end{array}$$

Figure 7: Extending the linear lambda calculus (LIN).

entirely straightforward, and follows the work of Girard, as shown in Figure 8. The call-by-value translation is also fairly straightforward, as shown in Figre 9. The call-by-value translation involves two points of note.

First, we maintain the uniform principle that values correspond to introducers while other term forms correpsond to eliminators. In call-by-value, the components of a pair are restricted to values. Hence, the only pair former is (V, W), where the two components are themselves 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 reduction rules and the translation easier to define, and corresponds to a restriction on pairs that arises naturally in call-by-need calculi [Lau93, AFMOW95].

Second, the call-by-value translation also uses the additive product '&' of linear logic. In contrast, Girard's version of this translation [Gir87] defines

$$(A \times B)^\dagger \quad \equiv \quad A^\dagger \otimes B^\dagger \ ,$$

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

$$A^\dagger \otimes B^\dagger \equiv (!A^\ddagger) \otimes (!B^\ddagger) \simeq !(A^\ddagger \,\&\, B^\ddagger) \equiv !(A \times B)^\ddagger \equiv (A \times B)^\dagger \ .$$

All results carry through for the extended calculi.

7.2 Sums

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

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

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

$$(A + B)^\dagger \quad \simeq \quad A^\dagger \oplus B^\dagger \ .$$

Syntactic domains

$$\begin{array}{llcl} \text{Types} & A, B, C & ::= & \cdots \mid A \times B \\ \text{Terms} & L, M, N & ::= & \cdots \mid \text{fst}\, M \mid \text{snd}\, M \mid p\, M_1 \cdots M_K \\ \text{Values} & V, W & ::= & \cdots \mid (M, N) \mid c \end{array}$$

Typing judgements

$$\times\text{-I}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \times\text{-E}_1\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{fst}\, M : A} \qquad \times\text{-E}_2\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{snd}\, M : B}$$

$$\text{Const}\ \frac{}{\vdash c : \text{type}_{\text{NAME}}(c)} \qquad \text{Prim}\ \frac{\Gamma_1 \vdash M_1 : \text{type}_{\text{NAME}}(p, 1) \quad \cdots \quad \Gamma_k \vdash M_k : \text{type}_{\text{NAME}}(p, k)}{\Gamma_1, \ldots, \Gamma_k \vdash p\, M_1 \cdots M_k : \text{type}_{\text{NAME}}(p, 0)}$$

Reduction relation

$$\begin{array}{lcl} (\beta\times_1) & \text{fst}\,(M, N) & \to \quad M \\ (\beta\times_2) & \text{snd}\,(M, N) & \to \quad N \\ (\delta) & p\, c_1 \cdots c_k & \to \quad \text{apply}_{\text{NAME}}(p, c_1, \ldots, c_k) \end{array}$$

Translation

$$\begin{array}{rcl} (A \times B)^* & = & A^* \,\&\, B^* \\ \\ (M, N)^* & = & (M^*, N^*) \\ (\text{fst}\, M)^* & = & \text{fst}\, M^* \\ (\text{snd}\, M)^* & = & \text{snd}\, M^* \\ c^* & = & c \\ (p(M_1, \ldots, M_k))^* & = & p(M_1^*, \ldots, M_k^*) \\ \\ \text{type}_{\text{LIN}}(c) & = & (\text{type}_{\text{NAME}}(c))^* \\ \text{type}_{\text{LIN}}(p, i) & = & (\text{type}_{\text{NAME}}(p, i))^* \\ \text{apply}_{\text{LIN}}(p\, c_1 \cdots c_k) & = & (\text{apply}_{\text{NAME}}(p, c_1, \ldots, c_k))^* \end{array}$$

Figure 8: Extending the call-by-name lambda calculus (NAME).

As with products it is desirable to re-express the left hand side in the form $!(A+B)^\ddagger$. 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, as shown in Figures 7–9. We specify a total function 'type' giving the type of each constant and the types of the arguments and results of each primitive, and a partial function 'apply' giving a closed term for each defined application of a primitive to constants. Plotkin [Plo75] explains why it is convenient for 'apply' to yield closed terms rather than constants. 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, as shown in the figures. Again, all 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 [AFMOW95].

Syntactic domains

$$\begin{array}{llcl} \text{Types} & A, B, C & ::= & \cdots \mid A \times B \\ \text{Terms} & L, M, N & ::= & \cdots \mid \text{fst}\, M \mid \text{snd}\, M \mid p\, M_1 \cdots M_k \\ \text{Values} & V, W & ::= & \cdots \mid (V, W) \mid c \end{array}$$

Typing judgements

$$\times\text{-I}_v\ \frac{\Gamma \vdash V : A \qquad \Gamma \vdash W : B}{\Gamma \vdash (V, W) : A \times B} \qquad \times\text{-E}_1\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{fst}\, M : A} \qquad \times\text{-E}_2\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{snd}\, M : B}$$

$$\text{Const}\ \frac{}{\vdash c : \text{type}_{\text{VAL}}(c)} \qquad \text{Prim}\ \frac{\Gamma_1 \vdash M_1 : \text{type}_{\text{VAL}}(p, 1) \quad \cdots \quad \Gamma_k \vdash M_k : \text{type}_{\text{VAL}}(p, k)}{\Gamma_1, \ldots, \Gamma_k \vdash p\, M_1 \cdots M_k : \text{type}_{\text{VAL}}(p, 0)}$$

Reduction relation

$$\begin{array}{lll} (\beta\times_{1v}) & \text{fst}\ (V, W) & \to\ V \\ (\beta\times_{2v}) & \text{snd}\ (V, W) & \to\ W \\ (\delta) & p\, c_1 \cdots c_k & \to\ \text{apply}_{\text{VAL}}(p, c_1, \ldots, c_k) \end{array}$$

Translation

$$\begin{array}{rcl} (A \times B)^{\ddagger} & = & A^{\ddagger} \,\&\, B^{\ddagger} \\ \\ (\text{fst}\ M)^{\dagger} & = & \text{let}\ !z = M^{\dagger}\ \text{in fst}\ z \\ (\text{snd}\ M)^{*} & = & \text{let}\ !z = M^{\dagger}\ \text{in snd}\ z \\ (p\, M_1 \cdots M_k)^{\dagger} & = & \text{let}\ !x_1 = M_1^{\dagger}\ \text{in} \cdots \text{let}\ !x_k = M_k^{\dagger}\ \text{in}\ p\, x_1 \cdots x_k \\ \\ (V, W)^{\ddagger} & = & (V^{\ddagger}, W^{\ddagger}) \\ c^{\ddagger} & = & c \\ \\ \text{type}_{\text{LIN}}(c) & = & (\text{type}_{\text{NAME}}(c))^{\dagger} \\ \text{type}_{\text{LIN}}(p, i) & = & (\text{type}_{\text{NAME}}(p, i))^{\dagger} \\ \text{apply}_{\text{LIN}}(p, c_1, \ldots, c_k) & = & (\text{apply}_{\text{NAME}}(p, c_1, \ldots, c_k))^{\dagger} \end{array}$$

Figure 9: Extending the call-by-value lambda calculus (VAL).

7.4 Recursion

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

$$\text{Rec}\ \frac{\Gamma, x : A \vdash M : A}{\Gamma \vdash \mu x.\, M : A}$$

in NAME, we should take

$$\text{Rec}\ \frac{!\Gamma, !x : !A \vdash M : A}{!\Gamma \vdash \mu x.\, M : A}$$

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

$$\text{Rec}\ \frac{\Gamma, x : A \vdash V : A}{\Gamma \vdash \mu x.\, V : A}$$

in VAL, we should take

$$\text{Rec}\ \frac{!\Gamma, !x : !A \vdash M : !A}{!\Gamma \vdash \mu x.\, M : !A}$$

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 [AFMOW95]. 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 (let $x_1 = M_1, \ldots x_k = M_k$ in N). This doesn't appear to pose any fundamental difficulties for translation, but the expected increase in the size of the calculus is awkward.

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. While we have illuminated substantial issues in the relationship of the various calculi in terms of their translations into the linear calculus, many interesting questions remain. We conclude this summary by mentioning three such issues.

8.1 Standard reductions

Our previous work on call-by-need has actually used a slightly different version of the (V) rule,

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

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.

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

Similarly, one might add ($\eta{\multimap}$) and ($\eta!$) rules to the linear lambda calculus.

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

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)	$V\ E$	$\to$	$\text{let } x = E \text{ in } V\ x$	

Figure 10: Moggi's computational lambda calculus COMP.

The second of these is similar to the right unit law in Moggi's computation lambda calculus [Mog88, Mog89]. 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 $*$ from NAME to LIN should preserve the (η) rule, that the translation $\dagger$ from VAL to LIN should preserve the (η_v) rule, and that the translation $\dagger$ 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

In addition to reduction, one usually also considers a theory of equality. Given a confluent notion of reduction $\twoheadrightarrow$, one defines $=$ as its reflexive, symmetric, and transitive closure. It follows from the Church-Rosser theorem that $M = N$ if and only if there exists a term L such that $M \twoheadrightarrow L$ and $N \twoheadrightarrow L$.

For the translation $*$ from NAME to LIN, it is relatively easy to show that not only do we have that $M \xrightarrow{\text{NAME}}\!\!\!\!\rightarrow N$ if and only if $M^* \xrightarrow{\text{LIN}}\!\!\!\!\rightarrow N^*$, but we also have $M \overset{\text{NAME}}{=} N$ if and only if $M^* \overset{\text{LIN}}{=} N^*$.

For the translation $\dagger$ from LET to LIN, the fact that $M \xrightarrow{\text{LET}}\!\!\!\!\rightarrow N$ if and only if $M^\dagger \xrightarrow{\text{LIN}}\!\!\!\!\rightarrow N^\dagger$ guarantees that $M \overset{\text{LET}}{=} N$ implies $M^\dagger \overset{\text{LIN}}{=} N^\dagger$. However, the reverse implication does not hold, since we may have $M^\dagger \xrightarrow{\text{LIN}}\!\!\!\!\rightarrow L$ and $N^\dagger \xrightarrow{\text{LIN}}\!\!\!\!\rightarrow L$ without having L be of the form $K^\dagger$ for some K in LET. For instance, it is easy to see that $(\text{let } x = M \text{ in } x\,N)^\dagger \overset{\text{LIN}}{=} (M\ N)^\dagger$, but not $(\text{let } x = M \text{ in } x\,N) \overset{\text{LET}}{=} (M\ N)$.

It is reasonable to ask if more laws could be added to LET to yield a theory (LET') for which $M \overset{\text{LET}'}{=} N$ if and only if $M^* \overset{\text{LIN}}{=} N^*$. A candidate for such a theory is Moggi's computational lambda calculus COMP [Mog88, Mog89], as shown in Figure 10. (The version shown here is based on the untyped reduction calculus, which appears in the technical report [Mog88] but not the LICS paper [Mog89].) The grammar of this theory is the same as for LET, except that both values and non-values are distinguished. This system satisfies subject reduction, and Moggi shows that it is confluent. The system is designed so that, apart from rule (β_v), it is strongly normalising, even without types.

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, $M \xrightarrow{\text{LET}}\!\!\!\!\rightarrow N$ does not imply $M \xrightarrow{\text{COMP}}\!\!\!\!\rightarrow N$, nor is the 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 [Plo75]. Moggi designed COMP to be sound and complete for the monad translation (which generalises CPS) [Mog88], and Sabry and Felleisen verified that the CPS translation from COMP into VAL is both sound and complete [SF93], 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, we can at least claim to be asking the right sort of questions!

References

[Abr92] S. Abramsky, The lazy lambda calculus. In D. Turner, editor, *Research topics in functional programming*, Addison-Wesley, 1992.

[Abr93] S. Abramsky, Computational interpretations of linear logic. *Theoretical Computer Science* 111:3–57, 1993.

[AF94] Z. 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.

[AFMOW95] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. *Symposium on Principles of Programming Languages*, ACM Press, San Francisco, California, January 1995.

[Bar81] 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.

[BH90] H. P. Barendregt and K. Hemerik. Types in Lambda Calculus and Programming Languages, *Proc. ESOP '90, European Symposium on Programming*, p. 1–35. Springer-Verlag, Lecture Notes in Computer Science 432, 1990.

[BBdPH92] N. Benton, G. Bierman, V. de Paiva, and M. Hyland, Type assignment for intuitionistic linear logic. Draft paper, August 1992.

[Gir87] J.-Y. Girard, Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir92] J.-Y. Girard, Message to 'linear' electronic mailing list, September 1992.

[Gir93] J.-Y. Girard, On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[Hol88] S. Holmström, A linear functional language. Draft paper, Chalmers University of Technology, 1988.

[Jac92] B. Jacobs, Semantics of weakening and contraction. Manuscript, Cambridge University, May 1992. Available from `bjacobs@cwi.nl`.

[Laf88] Y. Lafont, The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[Lau93] J. Launchbury, A natural semantics for lazy evaluation, *Symposium on Principles of Programming Languages*, ACM Press, Charleston, South Carolina, January 1993.

[LM92] 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.

[Mac91] I. Mackie, Lilac: a functional programming language based on linear logic. Master's Thesis, Imperial College London, 1991.

[Mog88] 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.

[Mog89] E. Moggi, Computational lambda-calculus and monads. In *4'th Symposium on Logic in Computer Science*, IEEE Press, Asilomar, California, June 1989.

[MOW94] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus (Unabridged). Technical Report 28/94, Fakultät für Informatik, Universität Karlsruhe, October 1994. Also appears as a technical report of the Department of Computing Science, University of Glasgow.

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

[Plo93] G. D. Plotkin, private communication. (During the 1993 MFPS, even!)

[Rey93] J. C. Reynolds, The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–248, November 1993.

[RR94] 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`.)

[Sch94] H. Schellinx, The noble art of linear decorating. Ph.D. dissertation, Institute for Logic, Language, and Computation, University of Amsterdam, 1994.

[SF93] A. Sabry and M. Felleisen, Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, November 1993.

[Tro92] A. S. Troelstra, *Lectures on Linear Logic*. CSLI Lecture Notes, 1992.

[Wad90] 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.

[Wad91] P. Wadler, Is there a use for linear logicΓ In *Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, New Haven, Connecticut, ACM Press, June 1991.

[Wad93a] 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.

[Wad93b] P. Wadler, A taste of linear logic. *Mathematical Foundations of Computer Science*, Gdansk, Poland, August 1993. Springer Verlag, LNCS 711.

A Subject reduction for linear calculus

This appendix details the proof of the subject reduction theorem for the linear calclus LIN.

Rule ($\beta{\multimap}$) arises when a $\multimap$-I rule meets a $\multimap$-E rule.

$$\dfrac{\dfrac{\begin{array}{c} x : A \vdash x : A \\ \vdots \\ \Gamma,\, x : A \cdots \vdash M : B \end{array}}{\Gamma \vdash \lambda x.\, M : A \multimap B}\ \multimap\text{-I} \qquad \begin{array}{c} \vdots \\ \Delta \vdash N : A \end{array}}{\Gamma,\, \Delta \vdash (\lambda x.\, M)\, N : B}\ \to\text{-E}$$

$$\Downarrow$$

$$\begin{array}{c} \vdots \\ \Delta \vdash N : A \\ \vdots \\ \Gamma,\, \Delta \vdash M[x := N] : B \end{array}$$

Rule ($\beta!$) arises when a !-I rule meets a !-E rule.

$$\dfrac{\dfrac{\begin{array}{c} \vdots \\ !\Gamma \vdash M : A \end{array}}{!\Gamma \vdash !M : !A}\ !\text{-I} \qquad \begin{array}{c} \dfrac{}{x_1 : A \vdash x_1 : A}\ \text{Id} \quad \cdots \quad \dfrac{}{x_k : A \vdash x_k : A}\ \text{Id} \\ \vdots \\ \Delta,\, x_1 : A,\, \ldots x_k : A \vdash N_{1\ldots k} : B \\ \hline\hline \Delta,\, !x : !A \vdash N : B \end{array}}{!\Gamma,\, \Delta \vdash \text{let } !x = !M \text{ in } u : B}\ !\text{-E}$$

$$\Downarrow$$

$$\begin{array}{c} \begin{array}{ccc} \vdots & & \vdots \\ !\Gamma_1 \vdash M_1 : A & \cdots & !\Gamma_k \vdash M_k : A \end{array} \\ \vdots \\ !\Gamma_1,\, \ldots,\, !\Gamma_n,\, \Delta \vdash N_{1\ldots k}[x_1 := M_1, \ldots, x_k := M_k] : B \\ \hline\hline !\Gamma,\, \Delta \vdash N[x := M] : B \end{array}$$

This rule requires some additional explanation. The double bars stand for a number of applications of Contraction, Weakening, and Dereliction. Subscripts have been introduced to handle the renaming required by contraction. Assume $\Delta,\, !x : !A \vdash N : B$, where there are k occurrences of x in N. Let $N_{1\ldots k}$ be the same term where these occurrences are replaced respectively by the distinct fresh variables $x_1, \ldots, x_k$, so that $N = N_{1\ldots k}[x_1 := x, \ldots x_k := x]$. Assume $!\Gamma \vdash M : A$. For each i in the range $1 \ldots k$, let σ_i be a substitution taking the variables in Γ into fresh distinct variables, and take $M_i = M[\sigma_i]$ and $\Gamma_i = \Gamma[\sigma_i]$.

Rule (!!) arises when two !-E rules meet, commuting one through the other.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash L : !A\end{array} \qquad \begin{array}{c}\vdots\\ \Delta,\, !x : !A \vdash M : !B\end{array}}{\Gamma,\, \Delta \vdash \text{let } !x = L \text{ in } M : !B}\,!\text{-E} \qquad \begin{array}{c}\vdots\\ \Theta,\, !y : !B \vdash N : C\end{array}}{\Gamma,\, \Delta,\, \Theta \vdash \text{let } !y = (\text{let } !x = L \text{ in } M) \text{ in } N : C}\,!\text{-E}$$

$$\Downarrow$$

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash L : !A\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Delta,\, !x : !A \vdash M : !B\end{array} \qquad \begin{array}{c}\vdots\\ \Theta,\, !y : !B \vdash N : C\end{array}}{\Delta,\, \Theta,\, !x : !A \vdash \text{let } !y = M \text{ in } N : C}\,!\text{-E}}{\Gamma,\, \Delta,\, \Theta \vdash \text{let } x = L \text{ in } (\text{let } y = M \text{ in } N) : C}\,!\text{-E}$$

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

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash L : !A\end{array} \qquad \begin{array}{c}\vdots\\ \Delta,\, !x : !A \vdash M : B \multimap C\end{array}}{\Gamma,\, \Delta \vdash \text{let } !x = L \text{ in } M : B \multimap C}\,!\text{-E} \qquad \begin{array}{c}\vdots\\ \Theta \vdash N : B\end{array}}{\Gamma,\, \Delta,\, \Theta \vdash (\text{let } !x = L \text{ in } M)\; N : C}\,\multimap\text{-E}$$

$$\Downarrow$$

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash L : !A\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Delta,\, !x : !A \vdash M : B \multimap C\end{array} \qquad \begin{array}{c}\vdots\\ \Theta, \vdash N : B\end{array}}{\Delta,\, \Theta,\, !x : !A \vdash M\; N : C}\,\multimap\text{-E}}{\Gamma,\, \Delta,\, \Theta \vdash \text{let } x = L \text{ in } (M\; N) : C}\,!\text{-E}$$

The (!W) rule of affine lambda calculus arises when commuting (!-E) through Weakening.

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash M : !A\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Delta \vdash N : B\end{array}}{\Delta,\, !x : !A \vdash N : B}\,\text{Weakening}}{\Gamma,\, \Delta \vdash \text{let } !x = M \text{ in } N : B}\,!\text{-E}$$

$$\Downarrow$$

$$\dfrac{\begin{array}{c}\vdots\\ \Delta \vdash N : B\end{array}}{\Gamma,\, \Delta \vdash N : B}\,\text{Weakening}$$

B LET as a conservative extension of VAL

Definition. We take the operator $.^{-\text{LET}}$ to map LET-terms to VAL-terms by reverting let-expressions to applications of abstractions, *viz.*,

$$\begin{aligned} (\text{let } x = M \text{ in } N)^{-\text{LET}} &= (\lambda x.\, (M^{-\text{LET}}))\; (N^{-\text{LET}}) & (M\; N)^{-\text{LET}} &= (M^{-\text{LET}})\; (N^{-\text{LET}}) \\ x^{-\text{LET}} &= x & (\lambda x.\, M)^{-\text{LET}} &= \lambda x.\, (M^{-\text{LET}}) \;. \end{aligned}$$

Proposition. Let $M \xrightarrow[R]{\text{LET}} N$, for some rule R. Then $M[x := V] \xrightarrow[R]{\text{LET}} N[x := V]$.

Proof. Trivial, since replacing one value by another will not prevent any contractions, but note that the result does not hold for substitution of arbitrary terms. □

Proposition. Let $M \equiv \text{let } x = M_0 \text{ in } M_1$.

1. If $M \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \in \text{VAL}$, then there exists some value V_0 such that

$$\begin{aligned} M_0 &\xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow V_0 \ , \text{ and} \\ M_1[x := V_0] &\xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \ . \end{aligned}$$

2. If $M \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow V$, then there exists some value V_0 such that

$$\begin{aligned} M_0 &\xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow V_0 \ , \text{ and} \\ M_1[x := V_0] &\xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow V \ . \end{aligned}$$

Proof. Part 1 holds because the LET-binding must be removed for N to be a VAL-term. Taking σ to be the name of the reduction

$$\sigma : M \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \ ,$$

we must clearly have $\sigma = \sigma_1 + \sigma_2 + \sigma_3$,

$$\begin{aligned} \sigma_1 : \quad & M \equiv \text{let } x = M_0 \text{ in } M_1 \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow \text{let } x = V_0 \text{ in } M_1' \\ \sigma_2 : \quad & \text{let } x = V_0 \text{ in } M_1' \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow M_1'[x := V_0] \\ \sigma_3 : \quad & M_1'[x := V_0] \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \ . \end{aligned}$$

where σ_1 is the interleaving of two independant reductions

$$\begin{aligned} \sigma_{1a} : \quad & M_0 \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow V_0 \ , \text{ and} \\ \sigma_{1b} : \quad & M_1 \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow M_1' \ . \end{aligned}$$

By the preceeding proposition, σ_{1b} gives us

$$M_1[x := V_0] \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow M_1'[x := V_0] \ ,$$

and the result follows. Part 2 is similar; again the top-level let-construct must be contracted away to produce the given result. □

Proposition. Let $\langle M, M' \rangle \in (A, C)$, $M' \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \in \text{VAL}$. Then

$$M \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \ .$$

Proof. For $\langle M, M' \rangle \in (A)$ we have

$$\begin{aligned} M &\equiv \text{let } x = (\text{let } y = M_0 \text{ in } M_1) \text{ in } M_2 \\ M' &\equiv \text{let } y = M_0 \text{ in } (\text{let } x = M_1 \text{ in } M_2) \ . \end{aligned}$$

We know that $M_0 \xrightarrow{\text{LET}}\!\!\!\!\rightarrow V_0$, $M_1[y := V_0] \xrightarrow{\text{LET}}\!\!\!\!\rightarrow V_1$ and $M_2[y := V_1] \xrightarrow{\text{LET}}\!\!\!\!\rightarrow N$ from the preceeding proposition, and the result follows.

For $\langle M, M' \rangle \in (C)$ we have

$$\begin{aligned} M &\equiv (\text{let } x = M_0 \text{ in } M_1)\, M_2 \\ M' &\equiv \text{let } x = M_0 \text{ in } (M_1\, M_2) \ . \end{aligned}$$

As before, we have $M_0 \xrightarrow{\text{LET}}\!\!\!\!\rightarrow V_0$ and

$$(M_1\ M_2)[x := V_0] \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \ ,$$

and since $X \notin \text{fv}(M_2)$,

$$(M_1\ M_2)[x := V_0] \equiv (M_1[x := V_0])\ M_2$$

we have the the result as follows,

$$\begin{aligned} M &\equiv (\text{let } x = M_0 \text{ in } M_1)\ M_2 \\ &\xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow (\text{let } x = V_0 \text{ in } M_1)\ M_2 \\ &\xrightarrow[(V)]{\text{LET}} (M_1[x := V_0])\ M_2 \\ &\xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \ . \end{aligned}$$

□

Proposition. Let $M \xrightarrow{\text{LET}} M' \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \in \text{VAL}$. Then $M \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N$.

Proof. Unfortunately the preceeding result does not generalize very easily. The problem arises from internal (A)-redexes like

$$(\lambda x.\, f\ (\lambda x.\, x)\ (f\ (\lambda x.\, x\ x)))\ (\lambda g.\, \text{let } y = \text{let } z = g\ g \text{ in } z\ z \text{ in } y\ y) \ ,$$

where the (A)-redex must be both duplicated and given substitutions before it reduces to produce two (V)-redexes.

The solution is to use an extended reduction system over *marked* let-bindings. The markings are reminiscent of Barendregt's marked $(\beta\ \rightarrow)$-redexes [Bar81], except that where Barendregt marks terms which already are redexes, we mark bindings which must — since $N \in \text{VAL}$ — becomes (V)-redexes.

We rely on a series of three basic lemmas for the result, both of which rely rely on keeping track of the marked bindings.

1. **Lemma**. For $M \in \text{LET}$ (unmarked!) and subterms and contexts such that

$$\begin{aligned} M \equiv C_1[(\lambda x.\, M_0)\ N_0] &\xrightarrow[(I')]{\text{LET}} C_1[\text{let } x = N_0 \text{ in } M_0] \\ &\xrightarrow[(V)]{\text{LET}} C_2[M_1[x := V_1]] \equiv N \ , \end{aligned}$$

   we also have

$$M \xrightarrow[(I)]{\text{LET}}\!\!\!\!\rightarrow C_2'[\text{let } x = V_1' \text{ in } M_1'] \xrightarrow[(V)]{\text{LET}} C_2'[M_1'[x := V_1']] \xrightarrow[(I')]{\text{LET}}\!\!\!\!\rightarrow N \ ,$$

   where we have at least one of the two identities $M_1 \equiv M_1'$ and $V_1 \equiv V_1'$, and if not both then the (I')-redex is within either M_1 or V_1.

   Proof. Trivial, by induction on the contexts. □

   This lemma means that we may postpone (I)-contractions until just before the (V) contraction which removes the new binding, introducing never-removed bindings only last. We may end with more (I)-contractions than before, but the number of (V)-contractions will remain constant.

2. The first lemma lets us extend the usual standardization properties for call-by-value to (I, V)-reduction. We do not need to spell out the strategy; rather, we need only its existance for the following property.

Lemma. There is a standard strategy $\underset{s}{\to}$ for selecting (I,V)-redexes with the property that for all $M \in \text{LET}$ (which (I,V)-reduces to an abstraction) we have $V_0 \in \text{LET}$ such that

$$M \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\to V \quad \Longrightarrow \quad M \underset{s}{\to} V_0 \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\to V \ .$$

Proof. We write $M \xrightarrow[(I^?V)]{\text{LET}} N$ iff either

(a) $M \xrightarrow[(V)]{\text{LET}} N$ or

(b) $M \xrightarrow[(I)]{\text{LET}} M' \xrightarrow[(V)]{\text{LET}} N$, where the (I)-contractum and (V)-contractum are the same.

Then by the first lemma, we have that any LET reduction sequence can be written as $\xrightarrow[(I^?V)]{\text{LET}}$ contractions followed by $\xrightarrow[(I)]{\text{LET}}$ contractions, and the lemma follows from the correponding result for VAL sequences by identifying $\xrightarrow[(I^?V)]{\text{LET}}$ steps with $\beta_v \to$ steps in terms equivalent modulo (I)-reduction. □

3. Here we introduce terms including marked let bindings, LET′.

Lemma. If we have $N \in \text{LET}$ (unmarked!) and $C, C_i, M_i', V_i', M_i'' \in \text{LET}'$ such that

$$\begin{array}{lll}
C[\text{let}'\ y = M_0' \text{ in let}'\ x = V_0 \text{ in } M_0''] & & \\
\quad \xrightarrow[(V)]{\text{LET}} & C[\text{let}'\ y = M_0' \text{ in } M_0''[x := V_0]] & \\
\quad \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\to & C_1[\text{let}'\ y = V_1' \text{ in } M_1''] & \xrightarrow[(V')]{\text{LET}} \ C_1[M_1''[y := V_1']] \\
\quad \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\to & C_2[\text{let}'\ y = V_2' \text{ in } M_2''] & \xrightarrow[(V')]{\text{LET}} \ C_2[M_2''[y := V_2']] \\
\quad \vdots & & \\
\quad \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\to & C_n[\text{let}'\ y = V_n' \text{ in } M_n''] & \xrightarrow[(V')]{\text{LET}} \ C_n[M_n''[y := V_n']]
\end{array}$$

then we have V_0', $C_{i,j}'$ all in LET′ such that

$$\begin{array}{lllllllll}
C[\text{let'}\ y = M_0 \text{ in let'}\ x = V_0 \text{ in } M_0''] & & & & & & & & \\
\xrightarrow[(I,V)]{\text{LET}} & C[\text{let'}\ y = V_0' \text{ in let'}\ x = V_0 \text{ in } M_0''] & & & & & & & \\
\xrightarrow[(V)]{\text{LET}} & C[\text{let'}\ x = V_0[y := V_0'] \text{ in } M_0''[y := V_0']] & & & & & & & \\
\xrightarrow[(V)]{\text{LET}} & C[(M_0''[y := V_0'])[x := (V_0[y := V_0'])]] & & & & & & & \\
\equiv & C_{1,1}'[V_0'] & \xrightarrow[(I,V)]{\text{LET}} & C_{1,1}'[V_1'] & \equiv & C_{1,2}'[V_0'] & \xrightarrow[(I,V)]{\text{LET}} & C_{1,2}'[V_1'] & \\
 & & & & \equiv & \cdots & \equiv & C_{1,i_1}'[V_0'] & \xrightarrow[(I,V)]{\text{LET}} \ C_{1,i_1}'[V_1'] \\
\equiv & C_{2,1}'[V_0'] & \xrightarrow[(I,V)]{\text{LET}} & C_{2,1}'[V_1'] & \equiv & C_{2,2}'[V_0'] & \xrightarrow[(I,V)]{\text{LET}} & C_{2,2}'[V_1'] & \\
 & & & & \equiv & \cdots & \equiv & C_{2,i_1}'[V_0'] & \xrightarrow[(I,V)]{\text{LET}} \ C_{2,i_1}'[V_1'] \\
 & \vdots & & & & & & & \\
\equiv & C_{n,1}'[V_0'] & \xrightarrow[(I,V)]{\text{LET}} & C_{n,1}'[V_1'] & \equiv & C_{n,2}'[V_0'] & \xrightarrow[(I,V)]{\text{LET}} & C_{n,2}'[V_1'] & \\
 & & & & \equiv & \cdots & \equiv & C_{n,i_1}'[V_0'] & \xrightarrow[(I,V)]{\text{LET}} \ C_{n,i_1}'[V_1'] \ \equiv \ N \ .
\end{array}$$

Proof. Follows immediately from the preceeding result. □

Then to prove the proposition, we observe that elementary analysis of term structures will resolve all cases *except* that described by the last lemma, so the result follows immediately. □

Proposition. Let $M \xrightarrow{\text{LET}}\!\!\!\!\rightarrow N \in \text{VAL}$. Then

$$M \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \quad . \tag{1}$$

Proof. Follows immediately from the above result by induction on the length of the length of reduction sequence. □

Proposition. Let $M, M_0 \in \text{LET}$ and $M_0', N \in \text{VAL}$ such that

$$\begin{tikzcd}
M_0 \arrow[rr, two heads, "\text{LET}", "(I,V)"'] \arrow[dr, "-\text{LET}"'] & & N \in \text{VAL} \\
& M_0' \arrow[ur, "\text{VAL}"'] &
\end{tikzcd} \quad .$$

1. If $M \xrightarrow[(V)]{\text{LET}}\!\!\!\!\rightarrow M_0$, then $M^{-\text{LET}} \xrightarrow{\text{VAL}}\!\!\!\!\rightarrow M_0'$:

$$\begin{tikzcd}
M \arrow[r, "\text{LET}", "(V)"'] \arrow[dr, dotted, "\text{LET}"'] & M_0 \arrow[r, two heads, "\text{LET}", "(I,V)"'] \arrow[dr, "-\text{LET}"'] & N \in \text{VAL} \\
& \exists M' \arrow[r, dotted, "\text{VAL}"] & M_0' \arrow[u, "\text{VAL}"']
\end{tikzcd} \quad .$$

2. If $M \xrightarrow[(I)]{\text{LET}}\!\!\!\!\rightarrow M_0$, then $M^{-\text{LET}} \equiv M_0'$:

$$\begin{tikzcd}
M \arrow[r, "\text{LET}", "(I)"'] \arrow[dr, dotted, "\text{LET}"'] & M_0 \arrow[r, two heads, "\text{LET}", "(I,V)"'] \arrow[d, "-\text{LET}"'] & N \in \text{VAL} \\
& M_0' \arrow[ur, "\text{VAL}"'] &
\end{tikzcd} \quad .$$

Proof. Part 1 is by a simple structural induction on M; part 2 is trivial. □

Proposition. For $M \in \text{LET}$, if $M \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \in \text{VAL}$ then $M^{-\text{LET}} \xrightarrow{\text{VAL}}\!\!\!\!\rightarrow N$.

Proof. Follows from the previous result by induction on the length of the reduction sequence.

Proposition. For $M, N \in \text{VAL}$, if $M \xrightarrow{\text{LET}}\!\!\!\!\rightarrow N$ then $M \xrightarrow{\text{VAL}}\!\!\!\!\rightarrow N$.

Proof. Follows from the preceeding proposition and Equation 1, since for $M \in \text{VAL}$ we have $M^{-\text{LET}} \equiv M$:

$$\begin{aligned}
M \xrightarrow{\text{LET}}\!\!\!\!\rightarrow N \in \text{VAL} \quad &\Longrightarrow \quad M \xrightarrow[(I,V)]{\text{LET}}\!\!\!\!\rightarrow N \in \text{VAL} \\
&\Longrightarrow \quad M^{-\text{LET}} \xrightarrow{\text{VAL}}\!\!\!\!\rightarrow N \\
&\Longrightarrow \quad M \xrightarrow{\text{VAL}}\!\!\!\!\rightarrow N \quad .
\end{aligned}$$