The Call-by-Need Lambda Calculus

John Maraist, Martin Odersky
School of Computer and Information Science, University of South Australia
Warrendi Road, The Levels, Adelaide, South Australia 5095
{*maraist,odersky*}*@cis.unisa.edu.au*

Philip Wadler
Bell Laboratories, Lucent Technologies
700 Mountain Ave., Room 2T-304, Murray Hill, New Jersey 07974-0636
wadler@research.bell-labs.com

Appears in the *Journal of Functional Programming* 8(3):275-317 (May 1998).

Abstract

We present a calculus that captures the operational semantics of call-by-need. The call-by-need lambda calculus is confluent, has a notion of standard reduction, and entails the same observational equivalence relation as the call-by-name calculus. The system can be formulated with or without explicit let bindings, admits useful notions of marking and developments, and has a straightforward operational interpretation.

Introduction

The correspondence between call-by-value lambda calculi and strict functional languages (such as the pure subset of Standard ML) is quite good; the correspondence between call-by-name lambda calculi and lazy functional languages (such as Miranda or Haskell) is not so good. Call-by-name re-evaluates an argument each time it is used, a prohibitive expense. Thus, many lazy languages are implemented using the *call-by-need* mechanism proposed by Wadsworth (1971), which overwrites an argument with its value the first time it is evaluated, avoiding the need for any subsequent re-evaluation (Turner, 1979; Johnsson, 1984; Koopman and Lee, 1989; Peyton Jones, 1992).

Call-by-need reduction implements the observational behaviour of call-by-name in a way that requires no more substitution steps than call-by-value reduction. It seems to give us something for nothing — the rich equational theory of call-by-name without the overhead incurred by re-evaluating arguments. Yet the resulting gap between the conceptual and the implementation calculi can be dangerous since it might lead to program transformations that drastically increase the complexity of lazy functional programs. In practice, this discrepancy is dealt with in an *ad hoc* manner. One uses the laws of the call-by-name lambda calculus as support that the transformations do not alter the meaning of a program, and one uses informal reasoning to ensure that the transformations do not increase the cost of execution.

However, the reasoning required is more subtle than it may at first appear. For example, in the term

$$\begin{array}{l} \mathsf{let}\ x = 1 + 2 \\ \ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x + y \\ \quad\ \mathsf{in}\ f\ y + f\ y \end{array}$$

the variable x appears textually only once, but substituting $1 + 2$ for x in the body of the let will cause $1 + 2$ to be computed twice rather than once.

Underestimating the difficulty of this problem can be quite hazardous in practice. The Glasgow Haskell Compiler is written in Haskell, is self-compiled, and makes extensive use of program transformations. In one version of the compiler, one such transformation inadvertently introduced a loss of sharing, causing the symbol table to be rebuilt *each time* an identifier was looked up. The bug was subtle enough that it was not caught until profiling tools later pinpointed the cause of the slowdown (Sansom and Peyton Jones, 1995).

In this paper we present the call-by-need lambda calculus λ_{NEED}. We write "call-by-need" rather than "lazy" to avoid a name clash with the work of Abramsky (1990), which describes call-by-name reduction to weak head-normal form. We present our calculus in Section 2, after a review of the call-by-name and call-by-value calculi in Section 1.

The basic syntactic properties of λ_{NEED} are quite satisfying. Reduction in λ_{NEED} admits an interesting variation of the usual marking of redexes, which in turn gives the properties of finite developments and unique completions. While somewhat technical, these properties are very interesting from the point of view of reduction semantics, and make the proofs of the other results much easier and more systematic. Reduction in λ_{NEED} is confluent: reduction rules may be applied to any part of a term, including under a lambda, and regardless of order the same normal form will be reached. Confluence is valuable for modelling program transformations. We also have a notion of standard evaluation: a computable, deterministic strategy for choosing redexes which will terminate whenever any reduction sequence leads to a member of a natural class of answers. This property is valuable for modelling computation. We discuss all of these properties in Section 3. Call-by-need is observationally equivalent to call-by-name, where the notion of observation is taken to be reducibility to weak head-normal form, as in the lazy lambda calculus of Abramsky (1990) and Ong (1988). A corollary is that Abramsky and Ong's models are also sound and adequate for our calculus. We give the details of the relationship between call-by-name and call-by-need in Section 4. Our calculus is the only one which we know to satisfy all of these properties without considerably sacrificing simplicity.

Our formulation of call-by-need can also be given a natural semantics, similar to the one proposed for the lazy lambda calculus by Launchbury (1993), as we show in Section 5. There is a close correspondence between our natural semantics and our standard reduction scheme. In Section 6 we show that one can formulate λ_{NEED} with or without the use of a let construct. The reduction rules appear more intuitive if a let construct is used, but an equivalent calculus can be formed without bindings, simply taking ($\mathsf{let}\ x = M\ \mathsf{in}\ N$) and $(\lambda x.N)M$ to be indistinguishable.

We consider some of the more common extensions to basic lambda calculi in Section 7. Finally, in Section 8 we consider the relationship of our calculus to a number of other systems and concerns. In particular, we consider other formulations of call-by-need reduction,

Syntactic Domains

Variables	x, y, z		
Values	V, W	$::=$	$x \mid \lambda x.M$
Terms	L, M, N	$::=$	$V \mid M\ N$
Evaluation contexts	E	$::=$	$[\,] \mid E\ M$

Reduction Rule

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

Fig. 1. The call-by-name lambda calculus.

Syntactic Domains

Variables, values, terms	As for call-by-name
Evaluation contexts	$E \quad ::= \quad [\,] \mid E\ M \mid (\lambda x.M)\ E$

Reduction Rule

$$(\beta_V) \qquad (\lambda x.M)\ V \quad \rightarrow \quad M[x := V]$$

Fig. 2. The call-by-value lambda calculus.

and stronger notions of reduction such as full laziness and optimal reduction. We also discuss other variations on the basic β-reduction rule, the relationship to classical and linear logics, and garbage collection. Probably the most serious drawback of our system is the lack of a good model for recursion; we include a discussion of work by other researchers on including recursive bindings.

1 The call-by-name and call-by-value calculi

Figures 1 and 2 review the call-by-name and call-by-value lambda calculi. Both calculi concern classical lambda terms: applications, abstractions and variables. A context $C[\,]$ is a term with a single hole $[\,]$ in it. By $C[M]$ we denote the term that results from replacing the hole in $C[\,]$ with M.

The call-by-name calculus (Church, 1941) consists of a single reduction rule, β, which describes the simplification of the application of an abstraction to an arbitrary argument. We define the reduction relation $\xrightarrow[(\beta)]{}$ to be the *compatible* closure of β under arbitrary contexts, and $\overset{}{\underset{(\beta)}{\twoheadrightarrow}}$ to be the reflexive, transitive closure of $\xrightarrow[(\beta)]{}$. We write $M \xmapsto[(\beta)]{} N$ to mean that we have $M \equiv E[\Delta_0]$, $N \equiv E[\Delta_1]$ and $\langle \Delta_0, \Delta_1 \rangle \in \beta$, with $\underset{(\beta)}{\mapsto\!\!\!\twoheadrightarrow}$ as the reflexive, transitive closure of $\xmapsto[(\beta)]{}$.

The call-by-value calculus (Plotkin, 1975) also consists of a single axiom, β_V, which is like β except that applications are contracted only when the argument is a value. We use the same notation for the relations derived from β_V as for those derived from β, and summarise the general notation below.

Notation. Throughout this article we use the following notational conventions, largely following Barendregt (1981). We use $\mathrm{fv}(M)$ to denote the free identifiers in a term M. A term is *closed* if $\mathrm{fv}(M) = \emptyset$. We use $M \equiv N$ for syntactic equality of terms (modulo α-renaming) and reserve $M = N$ for convertibility by the symmetric closure of reduction (or for example $M \stackrel{\text{NAME}}{=} N$ to specify the particular reduction axioms). Following Barendregt, we work with equivalence classes of α-renameable terms. To avoid name capture problems in substitutions we assume that the bound and free identifiers of a representative term and all its subterms are always distinct. We say that a reduction relation R is *confluent* if for all M_0, M_1, M_2 such that

$$\begin{array}{lcl} M_0 & \xrightarrow[(R)]{} \!\!\!\!\!\rightarrow & M_1 \text{ and} \\ M_0 & \xrightarrow[(R)]{} \!\!\!\!\!\rightarrow & M_2 \end{array}$$

we have some N such that $M_1 \xrightarrow[(R)]{}\!\!\!\!\!\rightarrow N$ and $M_2 \xrightarrow[(R)]{}\!\!\!\!\!\rightarrow N$. Reduction R is *strongly normalising* if no infinite R-reduction sequence exists.

Developments and their finiteness. In the results that follow we will make use of the notion of call-by-name developments, which we recall presently. The idea is to track individual redexes as others are contracted. We can identify redexes by their location within a term via paths, strings of symbols which indicate how one "navigates" from the top level of a term into its subterms. We use symbols $\underline{@1}, \underline{@2}, \underline{\lambda 1}$ respectively to indicate the left and right subterms of an application and the body of an abstraction. We let γ, δ range over paths and $\mathcal{F}, \mathcal{G}$ range over sets of paths, writing $(M, \mathcal{F})$ suggesting that paths in $\mathcal{F}$ index subterms of M which are top-level redexes. We also write $M \xrightarrow{\gamma} N$ where the term in M indexed by γ is the top-level redex which, when contracted, transforms M into N. This association of terms with sets of paths is intuitive, but unfortunately reduction rules for sets of paths are rather complicated. One generally moves freely back and forth between pairs of a term plus a set of paths on the one hand, and terms where certain redexes are indicated directly in the writing of the term on the other hand. Barendregt justifies the equivalence of the two formulations (1981, Chapter 11). We present the syntax and reduction rules of the marked call-by-name calculus λ'_{NAME} in Figure 3. We use the same metanotation for marked terms as for unmarked terms, except with a tick $'$ after the letter: hence marked terms L', M', N' and marked values V'. A *development* is a reduction sequence which contracts only marked redexes, that is, only (β_0) steps. A *complete* development is one which ends in an unmarked term. We write

$$\begin{array}{lcl} \sigma_1 : M'_1 & \xrightarrow[\text{dev}]{} & N'_1 \\ \sigma_2 : M'_2 & \xrightarrow[\text{dev}]{}\!\!\!\!\!\rightarrow & N'_2 \end{array}$$

to indicate that the single-step reduction sequence σ_1 and multi-step sequence σ_2 contracting marked terms M'_i to N'_i are developments, and

$$\tau : M' \xrightarrow[\text{cpl}]{} N$$

to indicate that a development τ is complete.

Example 1
Let $M' \equiv \stackrel{\beta}{-}(\lambda x.\, y\ x\ x)\ (\stackrel{\beta}{-}(\lambda z.\, z)\ u)$. We have two one-step developments of M', namely

$$\begin{array}{lcl} \sigma_1 : M' & \xrightarrow[\text{dev}]{} & y\ (\stackrel{\beta}{-}(\lambda z.\, z)\ u)\ (\stackrel{\beta}{-}(\lambda z.\, z)\ u) \\ & \equiv & (\ (y\ ((\lambda z.\, z)\ u)\ ((\lambda z.\, z)\ u))\ , \{\underline{@2}, \underline{@1}\underline{@2}\}) \end{array}$$

Syntactic Domains

Variables	x, y, z		
Values	V', W'	$::=$	$x \mid \lambda x.M'$
Terms	L', M', N'	$::=$	$V' \mid M'\ N' \mid \stackrel{\beta}{-}(\lambda x.M')\ N'$

Reduction Rules

$$
\begin{array}{lrcl}
(\beta_0) & \stackrel{\beta}{-}(\lambda x.M)\ N & \rightarrow & M[x := N] \\
(\beta_1) & (\lambda x.M)\ N & \rightarrow & M[x := N]
\end{array}
$$

Fig. 3. The marked call-by-name lambda calculus λ'_{NAME}.

Syntactic Domains

Variables	x, y, z		
Values	V, W	$::=$	$x \mid \lambda x.M$
Terms	L, M, N	$::=$	$V \mid M\ N \mid \mathsf{let}\ x = M\ \mathsf{in}\ N$

Reduction Rules

$$
\begin{array}{cll}
(I) & (\lambda x.M)\ N & \rightarrow \mathsf{let}\ x = N\ \mathsf{in}\ M \\
(V) & \mathsf{let}\ x = V\ \mathsf{in}\ C[x] & \rightarrow \mathsf{let}\ x = V\ \mathsf{in}\ C[V] \\
(C) & (\mathsf{let}\ x = L\ \mathsf{in}\ M)\ N & \rightarrow \mathsf{let}\ x = L\ \mathsf{in}\ M\ N \\
(A) & \mathsf{let}\ y = (\mathsf{let}\ x = L\ \mathsf{in}\ M)\ \mathsf{in}\ N & \rightarrow \mathsf{let}\ x = L\ \mathsf{in}\ \mathsf{let}\ y = M\ \mathsf{in}\ N \\
(G) & \mathsf{let}\ x = M\ \mathsf{in}\ N & \rightarrow N \qquad \textbf{if}\ x \notin \mathsf{fv}(N)
\end{array}
$$

Fig. 4. The call-by-need λ-calculus λ_{NEED}.

and

$$
\begin{array}{rcl}
\sigma_2 : M' & \xrightarrow[\text{dev}]{} & \stackrel{\beta}{-}(\lambda x.\, y\ x\ x)\ u \\
& \equiv & ((\lambda x.\, y\ x\ x)\ u, \{\epsilon\}) \ .
\end{array}
$$

Both of these developments, when completed, end in the same term:

$$M' \xrightarrow[\text{cpl}]{} y\ u\ u \ .$$

Since developments coincide with (β_0)-reduction, which is strongly normalising and confluent, we have the following result:

Proposition 1
(Barendregt, 1981, Theorem 11.2.25) All call-by-name developments are finite, all can be extended to a complete development, and all complete developments with the same origin end in the same term.

2 The call-by-need calculus

Figure 4 details the call-by-need calculus, λ_{NEED}. We augment the term syntax of the λ-calculus with a let-construct. The underlying idea is to represent a reference to a node in a graph by a let-bound identifier. Hence, sharing in a graph corresponds to naming in a term.

The second half of Figure 4 presents reduction rules for NEED.

- Rule (I), "introduction," introduces a let binding from an application. Given an application $(\lambda x.M)\ N$, a reducer should construct a copy of the body M where all occurrences of x are replaced by a reference to a single occurrence of the graph of N. Rule (I) models this behaviour by representing the reference with a let-bound name.
- Rule (V), "value," substitutes a value for one occurrence of a let-bound variable; hence it expresses dereferencing. Note that since only values are copied, there is no risk of duplicating work in the form of reductions that should have been made to a single, shared expression.
- Rule (C), "commute," allows let-bindings to commute with applications, and thus pulls a let-binding out of the function part of an application.
- Rule (A), "associate," transforms left-nested let's into right-nested let's. It is a directed version of the associativity law for the call-by-name monad (Moggi, 1991).
- Rule (G), "garbage collection," drops a let-binding whose defined variable no longer appears in the term. Rule (G) is not strictly needed for evaluation (as seen in Section 3 where we discuss standard reduction), but it helps to keep terms shorter.

Clearly, these rules never duplicate a term which is not a value. Furthermore, we will show in Section 4.2 that a term evaluates to an answer in our calculus if and only if it evaluates to an answer in the call-by-name λ-calculus. So NEED fulfills the expectations for what a call-by-need reduction scheme should provide: no loss of sharing except inside values, and observational equivalence to the classical call-by-name calculus.

Definition 2 (Call-by-need reduction)
Let $\rightarrow$ be the smallest relation that contains (I, V, C, A, G) and that is closed under the implication $M \rightarrow N \ \Rightarrow\ C[M] \rightarrow C[N]$. As for call-by-name and call-by-value, we write reduction in a single step as $\rightarrow$ and in zero or more steps as $\twoheadrightarrow$. To distinguish call-by-need from (say) call-by-name reduction, we write $\xrightarrow[\text{NEED}]{}$ and $\xrightarrow[\text{NAME}]{}$. To express reduction according to particular individual rules in a system, we will specify the rules similarly, as in $\xrightarrow[\beta]{}$ and $\xrightarrow[(I)]{}$. We will omit subscripts whenever the context is clear. We will often omit the Greek letter lambda to reduce clutter, and write (for example) NEED to refer to either the reduction theory λ_{NEED} or the collection of terms Λ_{NEED}.

Example 2
Consider the reduction of the term $(\lambda x.x\ x)\ (\lambda y.y)$:

$$
\begin{array}{llllll}
(\lambda x.x\ x)\ (\lambda y.y) & \xrightarrow[(I)]{} & \begin{array}{l}\mathsf{let}\ x = \lambda y.y\\ \ \mathsf{in}\ x\ x\end{array} & \xrightarrow[(V)]{} & \begin{array}{l}\mathsf{let}\ x = \lambda y.y\\ \ \mathsf{in}\ (\lambda z.z)\ x\end{array} \\[2ex]
& \xrightarrow[(I)]{} & \begin{array}{l}\mathsf{let}\ x = \lambda y.y\\ \ \mathsf{in}\ \mathsf{let}\ z = x\\ \quad\ \mathsf{in}\ z\end{array} & \xrightarrow[(V)]{} & \begin{array}{l}\mathsf{let}\ x = \lambda y.y\\ \ \mathsf{in}\ \mathsf{let}\ z = x\\ \quad\ \mathsf{in}\ x\end{array} \\[2ex]
& \xrightarrow[(V)]{} & \begin{array}{l}\mathsf{let}\ x = \lambda y.y\\ \ \mathsf{in}\ \mathsf{let}\ z = x\\ \quad\ \mathsf{in}\ \lambda y.y\end{array} & \xrightarrow[(G)]{}^{2} & \lambda y.y
\end{array}
$$

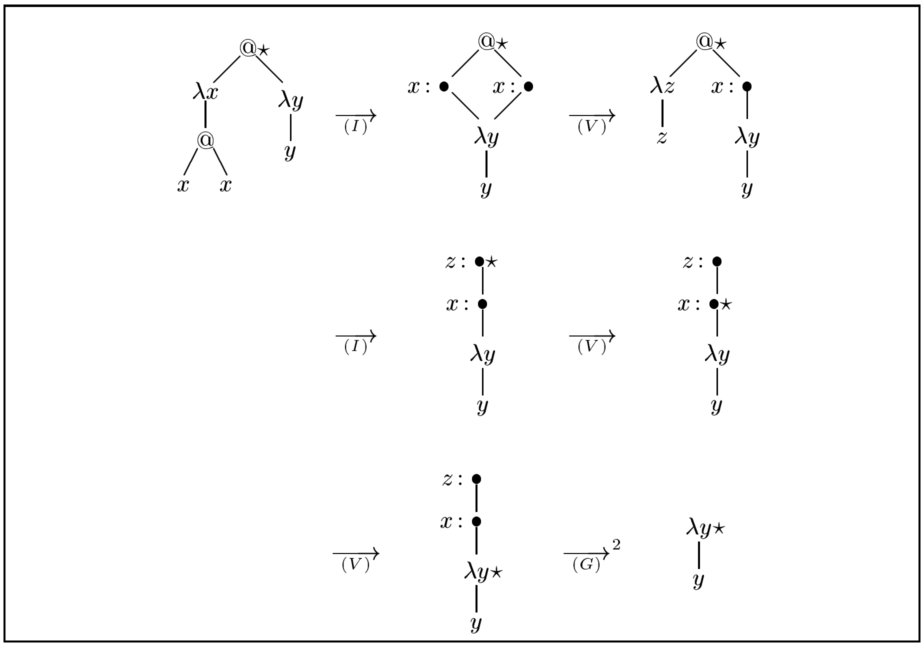

Fig. 5. Graphical rendering of Example 2.

Graphically, we have the sequence shown in Figure 5, where we mark the node currently considered the root of the graph with a star ($\star$).

The call-by-need calculus enjoys a number of properties which we summarise presently and detail over the next few sections.

- The notion of a marked redex can be adapted to call-by-need, and the resulting notion of developments has the same useful results as in call-by-name and value: all developments are finite, all can be extended to complete developments, and all complete developments of a given term and marking end in the same term. We formalise the notion of a call-by-need marking and verify these results in Section 3.1.
- The call-by-need calculus is confluent. As in the call-by-name and value systems, this result follows rather easily from the results on developments, as we show in Section 3.2.
- An *answer* is a reduction-closed set of terms that we select as an acceptable end result of a reduction sequence. In call-by-name and value one usually takes abstractions as answers; here we admit an abstraction under let-bindings as well. A *standard* reduction sequence is a subset of a reduction relation with three properties. First, every term may have at most one standard redex. Second, no answer may have a standard redex. Finally, whenever there is a reduction sequence from a term M to some answer, there is also a standard reduction sequence from M to an answer. We identify a standard strategy for selecting call-by-need redexes and show that it has these properties in Section 3.3.
- We express the correspondence between call-by-name and call-by-need in terms of *observational equivalences*, a sort of black-box testing. We make this black-box test

by wrapping both terms in the same context, and checking whether each wrapped term reduces to an answer, or *converges*. If the two terms exhibit the same behaviour (*i.e.* both converge, or both fail to converge) in every fixed situation, then we take the terms to be observationally equivalent. Then the relationship we show in Section 4 between call-by-name and call-by-need is that their theories of observational equivalence are exactly the same.

3 Syntactic issues

Lambda calculi have a number of syntactic properties that are useful in modelling programming languages, as has been demonstrated by their great success in modelling Algol, Iswim, and a host of successor languages. We discuss a number of these properties in this section. Section 3.1 concerns call-by-need developments and their finiteness. In Section 3.2 we discuss confluence. The confluence property set forth in the Church-Rosser theorem guarantees that reduction steps may occur in any order without changing the eventual final result, providing a simple model of program transformation and compiler optimisation. We discuss evaluation of call-by-need terms in Section 3.3, giving an evaluation order that contracts only one redex at a time, arriving in finitely many steps at an answer whenever possible.

3.1 Marked reduction and developments

We begin with a survey of some technical properties which are central to our proofs of confluence and standardisation, and which will also be useful in the correspondence results. The material of this section is relevant to the reduction theory of call-by-need, and is important for the results of later sections, which are arguably of more general interest. However, the reader who is less interested in those details can safely skip this section, and proceed to Section 3.2.

It is useful to track certain redexes as we contract others. To this end we *mark* redexes with tags to distinguish them from other, unmarked redexes. We track (I, V, C, A) redexes through reduction sequences with the marked call-by-need calculus $\textsc{need}'$ of Figures 6 and 7; we do not mark (G) redexes. This marked system differs in two distinct ways from more traditional marked systems such as the marked call-by-name calculus.

The first difference allows us to mark (V) steps, many of which could arise from a single let-binding. Rather than mark the binding, we mark the variable whose occurrence is to be replaced with the bound value. Since we mark variables rather than terms, we must place a restriction on let-bindings where variables are actually marked: in such bindings, the bound term must be a value. That is, in a term M,

$$M \equiv \mathsf{let}\ x = M_0\ \mathsf{in}\ M_1 \quad ,$$

if we have an occurrence of ${}^{\underline{\mathbf{V}}}x$ within M_1, then M_0 must be a value. Equivalently, we might mark the binding rather than the variables, and associate with the marking the subset of variables which marked reduction would replace; for the summary of these proofs which we present here, the marking of variables is simpler. We denote the set of variables which

Syntax

$$
\begin{array}{llll}
\text{Values} & V', W' & ::= & x \mid {}^{\underline{\mathbf{V}}}x \mid \lambda x.\, M' \\
\text{Terms} & L', M', N' & ::= & V' \mid M'\ N' \mid \mathsf{let}\ x = M'\ \mathsf{in}\ N' \\
 & & & \mid {}^{\mathbf{I}}(\lambda x.\, M')\ N' \\
 & & & \mid {}^{\underline{\mathbf{A}_n}}\mathsf{let}\ x = P'^{n}\ M'\ \mathsf{in}\ N' \\
 & & & \mid {}^{\underline{\mathbf{C}_n}}(P'^{n}\ M')\ N' \\
 & & & \quad \text{where in a term } \mathsf{let}\ x = M'\ \mathsf{in}\ N', \\
 & & & \qquad \text{if } x \in \mathrm{mv}(N') \\
 & & & \qquad \text{then } M' \text{ is a value.} \\
\text{Prefixes} & P', R' & ::= & \mathsf{let}\ x = M'\ \mathsf{in} \mid {}^{\underline{\mathbf{A}_n}}\mathsf{let}\ x = P'^{n}\ M'\ \mathsf{in}
\end{array}
$$

Trivial structural equivalences

$$
\begin{array}{rcl}
{}^{\underline{\mathbf{A}_0}}\mathsf{let}\ x = M'\ \mathsf{in}\ N' & \equiv & \mathsf{let}\ x = M'\ \mathsf{in}\ N' \\
{}^{\underline{\mathbf{C}_0}}(M'\ N') & \equiv & M'\ N'
\end{array}
$$

Top-level contraction

$$
\begin{array}{llll}
(I_0) & {}^{\mathbf{I}}(\lambda x.\, M')\ N' & \to \mathsf{let}\ x = N'\ \mathsf{in}\ M' & \\
(I_1) & (\lambda x.\, M')\ N' & \to \mathsf{let}\ x = N'\ \mathsf{in}\ M' & \\
(V_0) & \mathsf{let}\ x = V'\ \mathsf{in}\ C'[{}^{\underline{\mathbf{V}}}x] & \to \mathsf{let}\ x = V'\ \mathsf{in}\ C'[V'] & \\
(V_1) & \mathsf{let}\ x = V'\ \mathsf{in}\ C'[x] & \to \mathsf{let}\ x = V'\ \mathsf{in}\ C'[V'] & \\
(C_0) & {}^{\underline{\mathbf{C}_{n+1}}}(P'\ R'^{n}\ M')\ N' & \to P'\ ({}^{\underline{\mathbf{C}_n}}(R'^{n}\ M')\ N') & n \geq 0 \\
(C_1) & (P'\ M')\ N' & \to P'\ (M'\ N') & \\
(A_0) & {}^{\underline{\mathbf{A}_{n+1}}}\mathsf{let}\ x = (P'\ R'^{n}\ M')\ \mathsf{in}\ N' & \to P'\ ({}^{\underline{\mathbf{A}_n}}\mathsf{let}\ x = (R'^{n}\ M')\ \mathsf{in}\ N') & \\
(A_1) & \mathsf{let}\ x = (P'\ M')\ \mathsf{in}\ N' & \to P'\ (\mathsf{let}\ x = M'\ \mathsf{in}\ N') & \\
(G_1) & \mathsf{let}\ x = M'\ \mathsf{in}\ N' & \to N' & x \notin \mathrm{fv}(N')
\end{array}
$$

Fig. 6. Syntax and reduction axioms of the marked call-by-need calculus $\textsc{need}'$.

occur marked in a term M by $\mathrm{mv}(M)$, and refer to a marked (V) redex to mean a let-binding of a variable to some variable x where at least one occurrence of x is marked.

The second variation from simpler marked systems is our treatment of the (C, A) rules. Rather than single steps, for these rules we will mark consecutive sequences of redexes: for example we may have two (C) steps which arise from the same binding, although only one is contractable initially:

$$
\begin{array}{rl}
 & (\mathsf{let}\ x_1 = L_1\ \mathsf{in}\ \mathsf{let}\ x_2 = L_2\ \mathsf{in}\ M)\ N \\
\to & \mathsf{let}\ x_1 = L_1\ \mathsf{in}\ ((\mathsf{let}\ x_2 = L_2\ \mathsf{in}\ M)\ N) \\
\to & \mathsf{let}\ x_1 = L_1\ \mathsf{in}\ \mathsf{let}\ x_2 = L_2\ \mathsf{in}\ (M\ N)\ \ .
\end{array}
$$

In the marked calculus, we allow both of these bindings to be marked at the same time, distinguishing the number of bindings to be moved at any point:

$$
\begin{array}{rl}
 & {}^{\underline{\mathbf{C}_2}}(\mathsf{let}\ x_1 = L_1\ \mathsf{in}\ \mathsf{let}\ x_2 = L_2\ \mathsf{in}\ M)\ N \\
\to & \mathsf{let}\ x_1 = L_1\ \mathsf{in}\ {}^{\underline{\mathbf{C}_1}}(\mathsf{let}\ x_2 = L_2\ \mathsf{in}\ M)\ N \\
\to & \mathsf{let}\ x_1 = L_1\ \mathsf{in}\ \mathsf{let}\ x_2 = L_2\ \mathsf{in}\ (M\ N)\ \ .
\end{array}
$$

This extension of simple marks will also require a variation from the usual, rather simple

Compatible closure

$$\frac{M' \to N'}{\lambda x.M' \to \lambda x.N'}$$

$$\frac{M' \to N'}{L'\ M' \to L'\ N'} \qquad \frac{M' \to N'}{M'\ L' \to N'\ L'}$$

$$\frac{M' \to N'}{{}^{\underline{\mathbf{I}}}(\lambda x.L')\ M' \to^{\underline{\mathbf{I}}} (\lambda x.L')\ N'} \qquad \frac{M' \to N'}{{}^{\underline{\mathbf{I}}}(\lambda x.M')\ L' \to^{\underline{\mathbf{I}}} (\lambda x.N')\ L'}$$

$$\frac{M' \to N'}{{}^{\underline{\mathbf{C}_n}} L'\ M' \to^{\underline{\mathbf{C}_n}} L'\ N'} \qquad \frac{\sigma : M' \to N'}{{}^{\underline{\mathbf{C}_n}} M'\ L' \to^{\underline{\mathbf{C}_{n+\mathbf{d}(|\sigma|,n)}}} N'\ L'}$$

$$\frac{M' \to N'}{\mathsf{let}\ x = L'\ \mathsf{in}\ M' \to \mathsf{let}\ x = L'\ \mathsf{in}\ N'} \qquad \frac{M' \to N'}{\mathsf{let}\ x = M'\ \mathsf{in}\ L' \to \mathsf{let}\ x = N'\ \mathsf{in}\ L'}$$

$$\frac{M' \to N'}{{}^{\underline{\mathbf{A}_n}}\mathsf{let}\ x = L'\ \mathsf{in}\ M' \to^{\underline{\mathbf{A}_n}} \mathsf{let}\ x = L'\ \mathsf{in}\ N'}$$

$$\frac{\sigma : M' \to N'}{{}^{\underline{\mathbf{A}_n}}\mathsf{let}\ x = M'\ \mathsf{in}\ L' \to^{\underline{\mathbf{A}_{n+\mathbf{d}(|\sigma|,n)}}} \mathsf{let}\ x = N'\ \mathsf{in}\ L'}$$

Displacement function

$$\begin{array}{rcl}
\mathbf{d}(\ \mathsf{let}\ x = (\mathsf{let}\ y = L\ \mathsf{in}\ M)\ \mathsf{in}\ N & & \\
\to \mathsf{let}\ y = L\ \mathsf{in}\ \mathsf{let}\ x = M\ \mathsf{in}\ N\ , n) & = & 1,\ \text{if } n > 0. \\
\mathbf{d}(\mathsf{let}\ x = M\ \mathsf{in}\ N \to N, n) & = & -1,\ \text{if } n > 0. \\
\mathbf{d}(\mathsf{let}\ x = L\ \mathsf{in}\ M \to \mathsf{let}\ x = L\ \mathsf{in}\ N, n) & = & \mathbf{d}(M \to N, n-1),\ \text{if } n > 0. \\
\mathbf{d}(M \to N, n) & = & 0\ ,\ \text{otherwise}
\end{array}$$

Fig. 7. Compatible closure of marked NEED reduction.

notion of compatible closure. Consider the term

$${}^{\underline{\mathbf{C}_1}}(\mathsf{let}\ x = (\mathsf{let}\ y = L\ \mathsf{in}\ M)\ \mathsf{in}\ N_0)\ N_1 \quad ,$$

which has an unmarked (A) redex at position $\underline{@1}$. If this redex is contracted before the marked top-level step, we must adjust the counter associated with the marker to reflect the "new" binding separating N_0 and N_1:

$$\begin{array}{rl}
& {}^{\underline{\mathbf{C}_1}}(\mathsf{let}\ x = (\mathsf{let}\ y = L\ \mathsf{in}\ M)\ \mathsf{in}\ N_0)\ N_1 \\
\to & {}^{\underline{\mathbf{C}_2}}(\mathsf{let}\ y = L\ \mathsf{in}\ \mathsf{let}\ x = M\ \mathsf{in}\ N_0)\ N_1 \quad .
\end{array}$$

Were we to leave the counter unadjusted, we would lose confluence of marked reduction, and hence the uniqueness of complete developments as well.

We mark redexes with the four marks $^{\underline{\mathbf{I}}}$, $^{\underline{\mathbf{V}}}$, $^{\underline{\mathbf{C}_m}}$ and $^{\underline{\mathbf{A}_n}}$, where m, n are positive integers, each mark corresponding to the rule of the given name. We use the same metanotation for marked call-by-need terms as for marked call-by-name terms. In addition it is convenient to let P' range over the various marked let-bindings, and let (say) ${P'}^n$ range over n consecutive productions (not necessarily identical) of P'.

The top-level rules for NEED$'$ reduction are as usual with rules subscripted 0 contracting marked steps, and rules subscripted 1 contracting unmarked steps. We take NEED$'_0$ and NEED$'_1$ steps to refer to contraction by any of those respective sets of rules. For compatible closure we use the *displacement* function $\mathbf{d}$ on unmarked NEED sequences and integers. Intuitively, $\mathbf{d}$ returns the number of top-level let-bindings which are introduced or removed by a reduction sequence, where the first n nested let-bindings are considered top-level. This added complication in the definition of compatible closure allows marked NEED_0' reduction to be confluent.

Confluence of the marked subset is somewhat surprising, as simply marking single redexes alone (*i.e.*, without the numeric subscripts) is insufficient for the uniqueness result. Consider a term with two such marked (A) steps,

$$\underline{\mathbf{A}}\,\mathsf{let}\ x = (\underline{\mathbf{A}}\,\mathsf{let}\ y = (\mathsf{let}\ z = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \quad .$$

If we contract the outer redex first and inner redex second, we have one complete development:

$$\begin{array}{ll} & \underline{\mathbf{A}}\,\mathsf{let}\ x = (\underline{\mathbf{A}}\,\mathsf{let}\ y = (\mathsf{let}\ z = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \\ \to & \underline{\mathbf{A}}\,\mathsf{let}\ y = (\mathsf{let}\ z = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ (\mathsf{let}\ x = M_2\ \mathsf{in}\ M_3) \\ \to & \mathsf{let}\ z = M_0\ \mathsf{in}\ \mathsf{let}\ y = M_1\ \mathsf{in}\ \mathsf{let}\ x = M_2\ \mathsf{in}\ M_3 \quad . \end{array}$$

But if we contract the inner redex first, we have another complete development with a different ending:

$$\begin{array}{ll} & \underline{\mathbf{A}}\,\mathsf{let}\ x = (\underline{\mathbf{A}}\,\mathsf{let}\ y = (\mathsf{let}\ z = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \\ \to & \underline{\mathbf{A}}\,\mathsf{let}\ x = (\mathsf{let}\ z = M_0\ \mathsf{in}\ \mathsf{let}\ y = M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \\ \to & \mathsf{let}\ z = M_0\ \mathsf{in}\ \mathsf{let}\ x = (\mathsf{let}\ y = M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \quad . \end{array}$$

We have a similar problem for an (A) contraction occurring at the binding of a (C) step. In λ'_{NEED} we resolve the difficulty by adding a positive integer to (C, A) markings, indicating how many prefixes should be moved, and defining residuals to consider prefixes added or removed by other steps. So in the second sequence above, we will have:

$$\begin{array}{ll} & \underline{\mathbf{A}_1}\,\mathsf{let}\ x = (\underline{\mathbf{A}_1}\,\mathsf{let}\ y = (\mathsf{let}\ z = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \\ \to & \underline{\mathbf{A}_2}\,\mathsf{let}\ x = (\mathsf{let}\ z = M_0\ \mathsf{in}\ \mathsf{let}\ y = M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \\ \to & \mathsf{let}\ z = M_0\ \mathsf{in}\ \underline{\mathbf{A}_1}\,\mathsf{let}\ x = (\mathsf{let}\ y = M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \\ \to & \mathsf{let}\ z = M_0\ \mathsf{in}\ \mathsf{let}\ y = M_1\ \mathsf{in}\ \mathsf{let}\ x = M_2\ \mathsf{in}\ M_3 \quad , \end{array}$$

which does end with the same term as the first complete development.

Once again we can move freely between marked terms and sets of paths, with the additional symbols $\underline{\ell_1}$, $\underline{\ell_2}$ indexing respectively the left and right subchildren of a let-binding. In other words, in a term $\mathsf{let}\ x = M\ \mathsf{in}\ N$, on the path $\underline{\ell_1}$ we index M, and with $\underline{\ell_2}$ we index N. We continue with the notation $M \xrightarrow{\gamma} N$ to index top-level redexes under compatible closure, and write $|M'|$ to refer to the underlying unmarked term; if $M' \equiv (M, \mathcal{F})$ then we have $|M'| \equiv M$. Similarly, given a marked reduction sequence σ', we refer to the projection $|\sigma'|$ to mean the reduction sequence between the respective projections. Finally, we write ϵ for a zero-length path string of no symbols.

Having established the notion of marking, we can define *residuals*. For a reduction sequence $\sigma : M \twoheadrightarrow N$ and a marking $\mathcal{F}$ of M, we define the residuals of $\mathcal{F}$ with respect to

σ — in symbols we write $\mathcal{F}/\sigma$ — to be the set of residuals $\mathcal{G}$ such that

$$\sigma' : (M, \mathcal{F}) \twoheadrightarrow (N, \mathcal{G})$$

where $|\sigma'| \equiv \sigma$. Developments are as before: a development of a term M and marking $\mathcal{F}$ is a reduction sequence beginning from $(M, \mathcal{F})$ which contracts only marked redexes, and a complete development is one which ends in an unmarked term. We write $\xrightarrow[\text{dev}]{}$ and $\xrightarrow[\text{cpl}]{}$ as before.

Example 3
Let $M' \equiv (M, \mathcal{F}) \equiv {}^{\underline{\mathbf{A}_1}}\mathsf{let}\ x = ({}^{\underline{\mathbf{A}_1}}\mathsf{let}\ y = (\mathsf{let}\ z = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3$. There are two single-step developments of M', namely

$$\sigma_0 : M' \xrightarrow[\text{dev}]{} {}^{\underline{\mathbf{A}_1}}\mathsf{let}\ y = (\mathsf{let}\ z = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ (\mathsf{let}\ x = M_2\ \mathsf{in}\ M_3)$$

and

$$\sigma_1 : M' \xrightarrow[\text{dev}]{} {}^{\underline{\mathbf{A}_2}}\mathsf{let}\ x = (\mathsf{let}\ z = M_0\ \mathsf{in}\ (\mathsf{let}\ y = M_1\ \mathsf{in}\ M_2))\ \mathsf{in}\ M_3 \quad .$$

We have $\mathcal{F}/\sigma_0 = \{(\epsilon, 1)\}$, and $\mathcal{F}/\sigma_1 = \{(\epsilon, 2)\}$. Assuming that the M_i are unmarked, we have one complete development of M',

$$M' \xrightarrow[\text{cpl}]{} \mathsf{let}\ z = M_0\ \mathsf{in}\ (\mathsf{let}\ y = M_1\ \mathsf{in}\ (\mathsf{let}\ x = M_2\ \mathsf{in}\ M_3)) \quad .$$

The main result on NEED-developments is the following theorem:

Theorem 3
All λ_{NEED} developments are finite, and can be extended to a complete development. Moreover, all complete developments of a particular term and marking end in the same term.

Proof
As before, finiteness of developments is equivalent to strong normalisation of marked reduction NEED'_0. The technique is standard, based on a positive integer measure of a decoration of marked terms which is decreased by reduction. We give only a summary of the proof; full details are available elsewhere (Maraist, 1997).

We construct weighted terms by giving every variable occurrence x or ${}^{\underline{\mathbf{V}}}x$ a *weight* of some positive integer, written x^i or ${}^{\underline{\mathbf{V}}}x^i$. We let $\dot{M}, \dot{N}$ and so forth range over weighted terms, $\dot{V}$ range over weighted values, and define the norm $||\ .\ ||$ on weighted terms (ignoring marks) as follows:

$$\begin{array}{rcl}
||x^i|| & = & i \\
||\lambda x.\, \dot{M}|| & = & ||\dot{M}|| \\
||\mathsf{let}\ x = \dot{M}\ \mathsf{in}\ \dot{N}|| & = & 2||\dot{M}|| + ||\dot{N}|| \\
||\dot{M}\ \dot{N}|| & = & 2||\dot{M}|| + 2||\dot{N}||
\end{array}$$

A term is said to have *decreasing weighting* if it satisfies the appropriate condition below based on its form.

- All terms x^i or ${}^{\underline{\mathbf{V}}}x^i$ have decreasing weighting.
- A term $\lambda x.\, \dot{M}$ has a decreasing weighting if $\dot{M}$ has a decreasing weighting.
- A term ${}^{\underline{\mathbf{I}}}(\lambda x.\, \dot{M})\ \dot{N}$ has decreasing weighting if:

  1. Both $\dot{M}$ and $\dot{N}$ have decreasing weighting.

2. For all x^i or $\underline{\mathbf{V}}x^i$ in $\dot{M}$, we have $i > ||\dot{N}||$.

- Other applications $(\dot{M}\ \dot{N})$ or $\underline{\mathbf{C}_n}(\dot{M}\ \dot{N})$ have decreasing weighting if both $\dot{M}$ and $\dot{N}$ have decreasing weighting.
- A (V)-marked binding $(\mathsf{let}\ x = \dot{V}\ \mathsf{in}\ \dot{M})$ has decreasing weighting if:
  1. Both $\dot{V}$ and $\dot{M}$ have decreasing weighting.
  2. For all $\underline{\mathbf{V}}x^i$ in $\dot{M}$, we have $i > ||\dot{V}||$.
- Other bindings $(\mathsf{let}\ x = \dot{M}\ \mathsf{in}\ \dot{N})$ or $\underline{\mathbf{A}_n}(\mathsf{let}\ x = \dot{M}\ \mathsf{in}\ \dot{N})$ have decreasing weighting if both $\dot{M}$ and $\dot{N}$ have decreasing weighting.

We lift marked reduction to weighted terms by just applying the same rules without regard for weights. Decreasing weightings have two key properties, both of which can be shown by a straightforward structural induction. Let M have decreasing weighting, and let $\dot{M} \xrightarrow[\text{NEED}'_0]{} \dot{N}$. Then:

1. $||\dot{M}|| > ||\dot{N}||$, and
2. $\dot{N}$ has decreasing weighting.

Moreover, every term has a decreasing weighting. To construct a decreasing weighting for an arbitrary term, we number its variable occurrences by positive integers from 1, numbering the right-hand side of an application before the left-hand side, and the bound term of a let-binding before the body of the binding. Then to a variable numbered i we give the weight f_i,

$$\begin{aligned} f_1 &= 1 \\ f_n &= \left(\sum_{i=0}^{n} 2^i\right) \cdot f_{n-1}\ , \qquad n > 1 \end{aligned}$$

Then since every term has a decreasing weighting, finiteness follows from the two key properties above.

Extension follows from strong normalisation. Uniqueness of NEED'_0-normal forms is implied by confluence of NEED'_0. Since we already have strong normalisation it suffices to show weak confluence, which requires only a simple if tedious analysis of the relative positions of redexes. $\square$

Pairing certain paths with numeric indices in markings raises a technical issue which is trivial in the calculi without bindings but which requires mention here. In (say) marked call-by-name reduction, markings $\mathcal{F}$ are simply sets of paths; given matched terms $(M, \mathcal{F}_1)$ and $(M, \mathcal{F}_2)$ we clearly have a correspondence between the marking of M with all redexes in either $\mathcal{F}_1$ or $\mathcal{F}_2$ and $(M, \mathcal{F}_1 \cup \mathcal{F}_2)$. In λ'_{NEED} this correspondence is no longer trivial, since the set-theoretic union of two markings does not necessarily correspond to any term M'.

Example 4
Let

$$M \equiv \mathsf{let}\ x_0 = (\mathsf{let}\ x_1 = M_1\ \mathsf{in}\ \mathsf{let}\ x_2 = M_2\ \mathsf{in}\ N)\ \mathsf{in}\ L\ ,$$

with $\mathcal{F}_1 \equiv \{(\epsilon, 1)\}$ and $\mathcal{F}_2 \equiv \{(\epsilon, 2)\}$. Then there exists terms M'_1, M''_2 such that for each

i we have $M'_i \equiv (M, \mathcal{F}_i)$, but there is no M'_0 such that $M'_0 \equiv (M, \mathcal{F}_1 \cup \mathcal{F}_2)$. Specifically, we have

$$\begin{aligned} M'_1 \equiv (M, \mathcal{F}_1) &\equiv \underline{\mathbf{A}_1}\,\mathsf{let}\ x_0 = (\mathsf{let}\ x_1 = M_1\ \mathsf{in}\ \mathsf{let}\ x_2 = M_2\ \mathsf{in}\ N)\ \mathsf{in}\ L \\ M'_2 \equiv (M, \mathcal{F}_2) &\equiv \underline{\mathbf{A}_2}\,\mathsf{let}\ x_0 = (\mathsf{let}\ x_1 = M_1\ \mathsf{in}\ \mathsf{let}\ x_2 = M_2\ \mathsf{in}\ N)\ \mathsf{in}\ L \end{aligned}$$

but since we allow each redex to take no more than one mark (indexed or otherwise), we can form no term $(M, \mathcal{F}_1 \cup \mathcal{F}_2)$.

Rather than simple set union $\cup$, we instead use a modified relation $\uplus$. We define $\uplus$ to select only the largest integer to form a pair with each different path that set-theoretic union $\cup$ would associate with more than one integer. For the $\mathcal{F}_1, \mathcal{F}_2$ of the above example, we would have $\mathcal{F}_1 \uplus \mathcal{F}_2 \equiv \{(\epsilon, 2)\}$. Formally, we have

Definition 4
Let $\mathcal{F}_1, \mathcal{F}_2$ mark redexes in M. Then the set $\mathcal{F}_1 \uplus \mathcal{F}_2$ is defined as:

$$\mathcal{F}_1 \uplus \mathcal{F}_2 = \{\gamma : \gamma \in \mathcal{F}_1 \cup \mathcal{F}_2\} \cup \{(\gamma, \max\{i : (\gamma, i) \in \mathcal{F}_1 \cup \mathcal{F}_2\}) : (\gamma, n) \in \mathcal{F}_1 \cup \mathcal{F}_2\}$$

where $\max$ selects the largest of a finite set of natural numbers.

This relation allows us to prove the following lemma:

Lemma 5
Let $\mathcal{F}_0, \mathcal{F}_1$ mark redexes in M. Then there exists some $\mathcal{G}$ such that for any reduction sequence σ_i which is a complete development of $(M, \mathcal{F}_i)$, σ_i is also a partial development of $(M, \mathcal{G})$.

Proof
This $\mathcal{G}$ is just $\mathcal{F}_0 \uplus \mathcal{F}_1$. $\square$

3.2 Confluence

With the results on developments, confluence follows rather easily. Confluence of the (I, V, C, A) subset follows immediately from Theorem 3 and Lemma 5.

Lemma 6
Reduction of Λ_{NEED} terms by (I, V, C, A) steps is confluent: if $M \xrightarrow[\text{NEED}]{} \!\!\!\!\!\twoheadrightarrow M_1$ and $M \xrightarrow[\text{NEED}]{}\!\!\!\!\!\twoheadrightarrow M_2$, then there exists some N such that $M_1 \xrightarrow[\text{NEED}]{}\!\!\!\!\!\twoheadrightarrow N$ and $M_2 \xrightarrow[\text{NEED}]{}\!\!\!\!\!\twoheadrightarrow N$,

$$\begin{tikzcd} & M \arrow[dl, two heads, "\text{NEED}"'] \arrow[dr, two heads, "\text{NEED}"] & \\ M_0 \arrow[dr, two heads, dotted, "\text{NEED}"'] & & M_1 \arrow[dl, two heads, dotted, "\text{NEED}"] \\ & \exists N & \end{tikzcd} .$$

We use diagrams like the one above to illustrate asserted conditions. Reduction relations which are assumed for the result are drawn in solid lines, while reduction relations predicted by the result are dotted. On occasion we will also use dashed lines to highlight correspondences by relations other than reduction.

Proof

The result follows as in Barendregt's reference (1981, Chapter 11). Note that where in Barendregt's system the union of markings is trivial, here we must rely on Lemma 5 to justify the existence by $\Cup$ of a sensible combination of two markings of the same term. The heart of the proof is the following argument: Given two reduction sequences

$$\begin{aligned} \sigma_0 : (M, \mathcal{F}_0) &\xrightarrow[\text{cpl}]{} M_0 \text{ and} \\ \sigma_1 : (M, \mathcal{F}_1) &\xrightarrow[\text{cpl}]{} M_1 \ , \end{aligned}$$

we have by Lemma 5 some $\mathcal{G}$ such that

$$\begin{aligned} \sigma_0 : (M, \mathcal{G}) &\underset{\text{dev}}{\twoheadrightarrow} (M_0, \mathcal{G}_0) \text{ and} \\ \sigma_1 : (M, \mathcal{G}) &\underset{\text{dev}}{\twoheadrightarrow} (M_1, \mathcal{G}_1) \ . \end{aligned}$$

Moreover by Theorem 3 the completions of both σ_0 and σ_1 end in the same term: that is, we have some N such that for both i

$$(M_i, \mathcal{G}_i) \xrightarrow[\text{cpl}]{} N \ .$$

Since a single reduction step is trivially a complete development, it is a valid inductive conclusion that given $(L_0, \mathcal{F}) \xrightarrow[\text{cpl}]{} L_1$ and $L_0 \twoheadrightarrow L_2$ we have some L_3 with $L_1 \twoheadrightarrow L_3$ and $L_2 \twoheadrightarrow L_3$; a second induction with this result gives confluence. $\square$

Reduction by (I, V, C, A) steps and reduction by (G) steps commute in a specific useful way:

Lemma 7

Let $M \xrightarrow[(I,V,C,A)]{} M_1$ and $M \xrightarrow[(G)]{} M_2$. Then there exists some N such that $M_1 \underset{(G)}{\twoheadrightarrow} N$ and either $M_2 \equiv N$ or $M_2 \xrightarrow[(I,V,C,A)]{} N$.

Proof

By structural induction on M, and an easy examination of the relative positions of the redexes. $\square$

The above two lemmas are sufficient to imply confluence for λ_{NEED}.

Theorem 8

Reduction in λ_{NEED} is confluent:

$$\begin{tikzcd} & M \arrow[dl, "\text{NEED}"'] \arrow[dr, "\text{NEED}"] & \\ M' \arrow[dr, dotted, two heads, "\text{NEED}"'] & & M'' \arrow[dl, dotted, two heads, "\text{NEED}"] \\ & \exists N & \end{tikzcd}$$

Proof

Follows from Lemmas 6 and 7 (Barendregt, 1981, Lemma 3.3.5–7). $\square$

Additional Syntactic Domains

Answers	A, A_i	$::=$	$\lambda x.M \mid \mathsf{let}\ x = M\ \mathsf{in}\ A$
Evaluation Contexts	E, E_i	$::=$	$[\,] \mid E\ M \mid \mathsf{let}\ x = M\ \mathsf{in}\ E$
			$\mid \mathsf{let}\ x = E_0\ \mathsf{in}\ E_1[x]$

Standard Reduction Rules

(I_s)	$(\lambda x.M)\ N$	$\mapsto$	$\mathsf{let}\ x = N\ \mathsf{in}\ M$
(V_s)	$\mathsf{let}\ x = \lambda y.M\ \mathsf{in}\ E[x]$	$\mapsto$	$\mathsf{let}\ x = \lambda y.M\ \mathsf{in}\ E[\lambda y.M]$
(C_s)	$(\mathsf{let}\ x = L\ \mathsf{in}\ A)\ N$	$\mapsto$	$\mathsf{let}\ x = L\ \mathsf{in}\ A\ N$
(A_s)	$\mathsf{let}\ y = (\mathsf{let}\ x = L\ \mathsf{in}\ A)\ \mathsf{in}\ E[y]$	$\mapsto$	$\mathsf{let}\ x = L\ \mathsf{in}\ \mathsf{let}\ y = A\ \mathsf{in}\ E[y]$

Fig. 8. Standard call-by-need reduction.

3.3 Standard evaluation

The confluence result shows that different orders of reduction cannot yield different normal forms. It might nonetheless be the case that some reduction sequences terminate with a normal form while others do not terminate at all. However, the notion of reduction can be restricted to a standard sequence that always reaches an answer if one equal to the starting term exists.

Figure 8 details our notion of standard reduction. To state the standard reduction property, we first make precise the kind of observations that can be made about NEED programs. Following the spirit of Abramsky's work (1990), we define an observation to be a reduction sequence that ends in a function term. In NEED it makes sense to allow a function term to be wrapped in let-bindings, since we can remove bindings from positions interfering with a subsequent application of that function to an argument by rule (C). Hence, an answer A is either an abstraction or a let-binding whose body is an answer.

Standard reduction is a restriction of ordinary reduction in that each redex must occupy the hole of an evaluation context. The first two productions for evaluation contexts in Figure 8 are just as for the call-by-name calculus. The third production states that evaluation is possible in the body of a let. The final production highlights the call-by-need aspect of the strategy. It says that a definition should be evaluated if the defined node is demanded (*i.e.*, it appears in evaluation position itself). The second evaluation context in this form is the key; evaluation contexts reveal demand for one branch of this term by the other.

The restriction to evaluation contexts for redex selection does not by itself make call-by-need reduction deterministic. For instance,

$$\mathsf{let}\ x = V_0\ \mathsf{in}\ \mathsf{let}\ y = V_1\ \mathsf{in}\ x\ y$$

has both let's in evaluation position, and hence would admit either the substitution of V_0 for x or the substitution of V_1 for y. For the former contraction we have $E_0[\Delta_0] \rightarrow_0 [\Delta_0']$ where $E_0 \equiv [\,]$ and Δ_0 is the entire term; for the latter we have evaluation context ($\mathsf{let}\ x = V_0\ \mathsf{in}\ [\,]$) and contractum ($\mathsf{let}\ y = V_1\ \mathsf{in}\ x\ y$). We arrive at a deterministic standard reduction by specialising reduction rules to those shown in the second half of Figure 8. Note the use of evaluation contexts within these rules: evaluation contexts describe demand within redexes as well as within the contexts surrounding them.

Definition 9 (Call-by-need evaluation)

Let $\mapsto$ be the smallest relation that contains (I_s, V_s, C_s, A_s) and that is closed under the implication $M \mapsto N \quad\Rightarrow\quad E[M] \mapsto E[N]$. As usual we write $\mapsto\!\!\!\!\twoheadrightarrow$ for the reflexive, transitive closure of $\mapsto$, and refer to reduction by specific rules by writing the name of the rule below the arrow.

Theorem 10
The relation $\mapsto$ is a standard reduction relation for λ_{NEED}: for all terms M and answers A, the following three conditions hold.

- (Uniqueness) Exactly one of the following is true:
  1. M is an answer.
  2. We have some evaluation context E and $x \in \mathrm{fv}(M)$ such that $M \equiv E[x]$.
  3. We have some evaluation context E and top-level standard redex Δ such that $M \equiv E[\Delta]$.
- (Soundness) If $M \mapsto\!\!\!\!\twoheadrightarrow A$ then $M \twoheadrightarrow A$.
- (Completeness) If $M \twoheadrightarrow A$ then there exists some answer A_0 such that $M \mapsto\!\!\!\!\twoheadrightarrow A_0$.

Proof
Uniqueness of evaluation contexts follows by an easy structural induction on M. Soundness is trivial, as all $\mapsto$ steps are also $\to$ steps. For completeness the technique is as in Barendregt's result for call-by-name (1981, §11.4). We define an *internal* redex to be any (I, V, C, A) step which is not standard, and refer to such a contraction with $\underset{i}{\to}$. Since we do not mark (G) steps, we treat them separately. Each of the following properties can be shown by a tedious but conceptually simple case analysis:

- If $M \underset{i}{\to} A$ then M is also an answer.
- If $\sigma : M \to N_0$ and $M \xrightarrow{\gamma} N_1$, both by internal steps, then every redex in γ/σ is also internal.
- If $M \xrightarrow{\gamma} N$ is internal and $N \mapsto N_0$, then M has a standard redex.
- If $\sigma : M \to N_0$ is internal and $M \overset{\gamma}{\mapsto} N_1$, then γ/σ contains a single element which is also the standard redex of N_0.

From these properties and Lemma 5, we can use the finiteness of developments result in Theorem 3 to deduce that arbitrary (I, V, C, A) sequences can be reordered as standard steps followed by internal steps,

- If $M \underset{(I,V,C,A)}{\twoheadrightarrow} N$, then there exists some M_0 such that $M \mapsto\!\!\!\!\twoheadrightarrow M_0 \underset{i}{\twoheadrightarrow} N$.

In fact the use of Theorem 3 here and in the above steps is essential; it would be very difficult to make these arguments directly, without using developments. Moreover, a separate analysis shows that the following statement holds as well:

- If $M \underset{(G)}{\to} N \mapsto\!\!\!\!\twoheadrightarrow A$, then there exists some answer A_0 such that $M \mapsto\!\!\!\!\twoheadrightarrow A_0$.

It is clear that (G) steps preserve answers, and so completeness follows by induction on the internal steps leading to the standard sequence which terminates in an answer. $\square$

4 Call-by-need and call-by-name

The call-by-need calculus is confluent and has a standard reduction order, and so it is, at the least, a workable calculus by itself. Still we have yet to explore the relationship between λ_{NEED} and λ_{NAME}. The conversion theories $=_{\text{NEED}}$ and $=_{\text{NAME}}$ are clearly different — otherwise there would be little point in studying call-by-need systems! In this section we will demonstrate the exact difference between these calling conventions. We begin in Section 4.1 with the map $\pitchfork$ which maps call-by-need terms to call-by-name terms by simply substituting all let-bound terms for the bound variables. We use $\pitchfork$ to give a rigorous comparison of their reduction relations in Section 4.2; in Section 4.3 we show the coincidence of the observational equivalence relations over the common term language.

4.1 Relating the terms

The following map formalises the intuitive relationship between call-by-name and call-by-need terms.

Definition 11 (Let contraction map $\pitchfork$)
We define the map $\pitchfork$ from (marked) call-by-need terms to (marked) call-by-name terms as follows:

$$
\begin{array}{rcl}
x^{\pitchfork} & \equiv & x \\
(\lambda x.M)^{\pitchfork} & \equiv & \lambda x.M^{\pitchfork} \\
(M\ N)^{\pitchfork} & \equiv & M^{\pitchfork}\ N^{\pitchfork} \\
(\mathsf{let}\ x = M\ \mathsf{in}\ N)^{\pitchfork} & \equiv & N^{\pitchfork}[x := M^{\pitchfork}] \\
(^{\underline{\mathbf{I}}}(\lambda x.M)\ N)^{\pitchfork} & \equiv & {}^{\underline{\beta}}(\lambda x.M^{\pitchfork})\ N^{\pitchfork}\ \ .
\end{array}
$$

For terms decorated with other redex markers, we simply drop the marker and translate according to the above rules.

Example 5
Let

$$
\begin{array}{rcl}
M_0 & \equiv & (\lambda x.x)\ (\lambda y.\mathsf{let}\ z_0 = (\mathsf{let}\ z_1 = N_1\ \mathsf{in}\ N_2)\ \mathsf{in}\ N_3) \\
& \to & \mathsf{let}\ x = (\lambda y.\mathsf{let}\ z_0 = (\mathsf{let}\ z_1 = N_1\ \mathsf{in}\ N_2)\ \mathsf{in}\ N_3)\ \mathsf{in}\ x\ \equiv M_1 \\
& \to & \lambda y.\mathsf{let}\ z_0 = (\mathsf{let}\ z_1 = N_1\ \mathsf{in}\ N_2)\ \mathsf{in}\ N_3\ \equiv M_2\ \ .
\end{array}
$$

Then

$$
\begin{array}{rcl}
M_0^{\pitchfork} & \equiv & (\lambda x.x)\ (\lambda y.N_3^{\pitchfork}[z_0 := (N_2^{\pitchfork}[z_1 := N_1^{\pitchfork}])]) \\
M_1^{\pitchfork}\ \equiv\ M_2^{\pitchfork} & \equiv & \lambda y.N_3^{\pitchfork}[z_0 := (N_2^{\pitchfork}[z_1 := N_1^{\pitchfork}])]\ \ .
\end{array}
$$

Lemma 12
Let $M, N \in \text{NEED}$, where

1. $(M[x := N])^{\pitchfork} \equiv M^{\pitchfork}[x := N^{\pitchfork}]$.
2. If $M \overset{(V,C,A,G)}{=} N$ then $M^{\pitchfork} \equiv N^{\pitchfork}$.
3. M is an answer if and only if $M^{\pitchfork} \equiv \lambda x.\, N$.

Proof
All three clauses are straightforward: The first clause follows by a straightforward induction on the structure of M; the second, by inspection of the individual rules, and structural induction to find the redex contracted in M; and the third by the obvious structural induction. $\square$

Note that we write, for example, $\gamma\mathcal{F}$ as a shorthand for $\{\gamma\gamma_0 : \gamma_0 \in \mathcal{F}\}$. We now extend the map $\pitchfork$ to paths with respect to the term which the path indexes.

Definition 13 ($\pitchfork$-images of markings with respect to terms)
Let $\mathcal{F}$ index (I)-redexes in $M \in \text{NEED}$, where $(M, \mathcal{F})^{\pitchfork} \equiv (M^{\pitchfork}, \mathcal{G})$ for some set $\mathcal{G}$ marking (β)-redexes. Then we define the $\pitchfork$-image of $\mathcal{F}$ with respect to M to be $\mathcal{G}$, in symbols $\mathcal{F}^{\pitchfork}_{[M]} \equiv \mathcal{G}$. We will write the $\pitchfork$-image of a single path to mean simply the $\pitchfork$-image of the singleton set containing just that path, *i.e.*, $\gamma^{\pitchfork}_{[M]} \equiv \{\gamma\}^{\pitchfork}_{[M]}$.

Example 6
Taking M_0 as in Example 5, we have

$$\begin{array}{rcl} \{\epsilon\}^{\pitchfork}_{[M_0]} & \equiv & \{\epsilon\} \\ \{\underline{@2}\,\underline{\lambda 1}\}^{\pitchfork}_{[M_0]} & \equiv & \{\} \end{array}$$

Taking $N \equiv \mathsf{let}\ x = ((\lambda y.y)\ (\lambda z.z))\ \mathsf{in}\ ((x\ N_1)\ (x\ N_2))$, we have

$$\{\underline{\ell 1}\}^{\pitchfork}_{[N]} = \{\underline{@1}\,\underline{@1}, \underline{@2}\,\underline{@1}\} \ .$$

The following lemma explicitly justifies what might otherwise appear to be an abuse of the notation. Since $\pitchfork$-images of (V, C, A, G)-equal terms are identical, we can associate the $\pitchfork$-image of a path from either term with the $\pitchfork$-image of either term to produce the same valid member of NAME'.

Lemma 14
Let $M \overset{(V,C,A,G)}{=} N$, let $\mathcal{F}$ index (I)-redexes in N and let $\mathcal{G} = \mathcal{F}^{\pitchfork}_{[N]}$. Then $(M^{\pitchfork}, \mathcal{G})$ is a marked call-by-name term, and $(M^{\pitchfork}, \mathcal{G}) \equiv (N^{\pitchfork}, \mathcal{G})$

Proof
Trivial, since by Lemma 12.(2) we have $M^{\pitchfork} \equiv N^{\pitchfork}$. $\square$

4.2 Relating reduction

In this section we study the relationship between NEED reduction and NAME reduction. We will begin with some basic results about the operator, including the soundness of $\pitchfork$ for mapping multi-step NEED reduction sequences to multi-step NAME reduction sequences. By soundness we mean just that $\pitchfork$ preserves reduction sequences: if $M \underset{\text{NEED}}{\twoheadrightarrow} N$, then $M^{\pitchfork} \underset{\text{NAME}}{\twoheadrightarrow} N^{\pitchfork}$ as well.

Completeness is more tricky for two reasons: first, reduction in NEED may "overshoot" reduction in NAME. For example, we can consider the term

$$M \equiv (\lambda x.f\ x\ x)\ (I\ I) \ ,$$

L	$L^{\pitchfork}$
$\mathsf{let}\ x = I\ I\ \mathsf{in}\ f\ x\ x$	$f\ (I\ I)\ (I\ I)$
$(\lambda x.f\ x\ x)\ (\mathsf{let}\ y = I\ \mathsf{in}\ y)$	$(\lambda x.f\ x\ x)\ I$
$\mathsf{let}\ x = (\mathsf{let}\ y = I\ \mathsf{in}\ y)\ \mathsf{in}\ f\ x\ x$	$f\ I\ I$

Table 1. *Possible* (I) *steps* $M \xrightarrow[(I)]{} \!\!\!\!\rightarrow L$ *from* $M \equiv (\lambda x.f\ x\ x)\ (I\ I)$.

where $I \equiv \lambda x.x$. In NAME we have

$$\begin{array}{cl} & (\lambda x.f\ x\ x)\ (I\ I) \\ \rightarrow & f\ (I\ I)\ (I\ I) \\ \rightarrow & f\ I\ (I\ I) \\ \equiv & N\ . \end{array}$$

But for all L where $M \xrightarrow[\text{NEED}]{} \!\!\!\!\rightarrow L$, we do not necessarily have $L^{\pitchfork} \equiv N$; the strongest statement we can make about these L, M, N is that we will have $L^{\pitchfork} \overset{\text{NAME}}{=} N$. Table 1 shows the possible results of (I) steps $M \xrightarrow[(I)]{} \!\!\!\!\rightarrow L$; we do not consider (V, C, A, G) contraction since (as we show below) they preserve $\pitchfork$-images. This difficulty is easily overcome: we simple relax the statement of the completeness result to allow such overshooting; such behaviour is exactly what one would expect from introducing shared subexpressions into a compatibly-closed reduction relation.

The second complication arises in finding redexes in a NEED term M which correspond to each redex in the $\pitchfork$-image of a M: in some cases, there may be no corresponding redex in the original term. For example, in the term $M \equiv \mathsf{let}\ x = I\ \mathsf{in}\ x\ y$, there is no readily markable redex corresponding to the one contracted in $M^{\pitchfork} \equiv I\ y \xrightarrow[\text{NAME}]{} y$. The only redex in M is a (V) redex, which again does not vary the $\pitchfork$ image. Moreover, other sorts of redex in NEED terms can interfere similarly. Our solution to this problem is to normalise terms with respect to the (V, C, A, G) rules: then we can always associate redexes in a $\pitchfork$-image with a redex — obviously a (I) redex — in the original term. After establishing these preliminary results, the completeness result follows naturally.

Outline of the results. The behaviour of NEED reduction sequences under $\pitchfork$ is straightforward, and leads easily to the soundness result in Lemma 16. For completeness it is easier to work in NEED terms which have no (V, C, A, G) redexes. We first establish that all terms do indeed have unique (V, C, A, G)-normal forms, and give a grammar corresponding to these forms. We link reduction of NEED terms in general to (V, C, A, G)-normalisation by Corollary 22, and link reduction of (V, C, A, G)-normal NEED terms to (β)-reduction of NAME terms through Corollary 26. These results lead to the completeness argument, which we give as Lemma 27. The soundness and completeness properties are summarised as Proposition 28. We then extend the equivalence results to convergence (Corollary 30) and observational equivalence (Theorem 32).

Technically, the soundness and completeness results rely in an essential way on the notion of developments. What we actually show for soundness is that let contraction preserves complete developments: if σ is a complete development of a marked NEED term $(M, \mathcal{F})$ ending in N, then the complete NAME development of the $\pitchfork$ image of $(M, \mathcal{F})$ is $N^{\pitchfork}$. For completeness we show that a complete NAME development σ corresponds to a complete NEED development whose $\pitchfork$-image is again a complete NAME development τ; σ will be a partial development of the redexes marked at the beginning of τ.

For soundness we need two lemmas. Lemma 12.2 tells us that (V, C, A, G) steps preserve $\pitchfork$-images; the following result treats (I) steps.

Lemma 15
Let $M \xrightarrow[(I)]{\gamma} N$ be a NEED step. Then $(M, \{\gamma\})^{\pitchfork} \xrightarrow[\text{cpl}]{} N^{\pitchfork}$.

Proof
By structural induction on M. All of the cases are immediate from the induction hypothesis except when $M \equiv (\mathsf{let}\ x = M_0\ \mathsf{in}\ M_1)$. Then we have two cases, depending on the location of the redex.

1. $\gamma \equiv \underline{\ell_1}\delta$. Then $M_0 \xrightarrow{\delta} N_0$, $N \equiv (\mathsf{let}\ x = N_0\ \mathsf{in}\ M_1)$ and $N^{\pitchfork} \equiv M_1^{\pitchfork}[x := N_0^{\pitchfork}]$. By the induction hypothesis we have

   $$(M_0, \{\delta\})^{\pitchfork} \xrightarrow[\text{cpl}]{} N_0^{\pitchfork} \ ,$$

   or in other words

   $$(M_0, \{\delta\})^{\pitchfork} \underset{\text{NAME}'_0}{\twoheadrightarrow} N_0^{\pitchfork} \in \text{NAME} \ .$$

   Since marked call-by-name reduction is substitutive, we have σ such that

   $$\sigma : (M, \{\gamma\})^{\pitchfork} \equiv M_1^{\pitchfork}[x := (M_0, \{\delta\})^{\pitchfork}] \underset{\text{NAME}'_0}{\twoheadrightarrow} M_1^{\pitchfork}[x := N_0^{\pitchfork}] \equiv N \ ,$$

   and since $M_1^{\pitchfork}, N_0^{\pitchfork} \in$ NAME and unmarked, so is N, and σ is in fact a complete development.
2. $\gamma \equiv \underline{\ell_2}\delta$. Then $M_1 \xrightarrow{\delta} N_1$, $N \equiv \mathsf{let}\ x = M_0\ \mathsf{in}\ N_1$ and $N^{\pitchfork} \equiv N_1^{\pitchfork}[x := M_0^{\pitchfork}]$. The result is largely as in the previous subcase. By the induction hypothesis we have

   $$(M_1, \{\delta\})^{\pitchfork} \xrightarrow[\text{cpl}]{} N_1^{\pitchfork} \ ,$$

   or in other words

   $$(M_1, \{\delta\})^{\pitchfork} \underset{\text{NAME}'_0}{\twoheadrightarrow} N_1^{\pitchfork} \in \text{NAME} \ .$$

   Again by substitutivity of marked call-by-name reduction, we have σ such that

   $$\sigma : (M, \{\gamma\})^{\pitchfork} \equiv M_1^{\pitchfork}[x := (M_0, \{\delta\})^{\pitchfork}] \xrightarrow[\text{NAME}'_0]{} N_1^{\pitchfork}[x := M_0^{\pitchfork}] \equiv N \ ,$$

   and once again since $M_0^{\pitchfork}, N_1^{\pitchfork} \in$ NAME and unmarked, so is N, and we have that σ is a complete development.

□

Corollary 16 (Soundness of $\pitchfork$ for reduction)
Let $M, N \in$ NEED with $M \underset{\text{NEED}}{\twoheadrightarrow} N$. Then $M^{\pitchfork} \underset{\text{NAME}}{\twoheadrightarrow} N^{\pitchfork}$.

Proof

Syntactic Domains

Applicable Terms	$\bar{H}$	$::= \bar{V} \mid \bar{H}\ \bar{M}$
Values	$\bar{V}$	$::= \lambda x.\bar{M} \mid x$
Terms	$\bar{L}, \bar{M}, \bar{N}$	$::= \bar{V} \mid \mathsf{let}\ x = \bar{H}\ \bar{M}\ \mathsf{in}\ \bar{M} \mid \bar{H}\ \bar{M}$

where in a term ($\mathsf{let}\ x = \bar{H}\ \bar{M}_0\ \mathsf{in}\ \bar{M}_1$), $x \in \mathsf{fv}(\bar{M}_1)$

Fig. 9. (V, C, A, G)-normal NEED terms.

By induction on the length of the reduction sequence, with Lemma 12.(2) for (V, C, A, G) steps and Lemma 15 for (I) steps. □

For completeness we will use the subset of NEED terms shown in Figure 9. In fact we will show in the next lemma that this subset identifies the NEED terms which are in (V, C, A, G)-normal form. We let $\bar{L}, \bar{M}, \bar{N}$ range over these terms.

Lemma 17
Let $\bar{M}$ be as described in the figure.

1. $\bar{M}$ is a NEED term.
2. Moreover, a term $M \in$ NEED is in (V, C, A, G)-normal form if and only if it can be expressed as some term $\bar{M}$.

Proof
The first clause is trivial. For the second clause, it is clear that every term $\bar{M}$ has no (V, C, A, G)-redexes. For the converse, we consider structural induction on (V, C, A, G)-normal forms, and show that they are indeed equivalent to some term $\bar{M}$. Only applications and let-bindings are interesting; other term-forms are trivial.

1. Let $M \equiv M_0\ M_1$ be a (V, C, A, G)-normal form. By the induction hypothesis, each M_i is equivalent to some $\bar{M}_i$. Moreover, for M not to be a (C)-normal form, M_0 cannot be a let-binding, only a (V, C, A, G)-normal value or application: in other words, it must be some $\bar{H}$, and so we have $M \equiv \bar{H}\ \bar{M}_1$.
2. Let $M \equiv \mathsf{let}\ x = M_0\ \mathsf{in}\ M_1$ be a (V, C, A, G)-normal form. By the induction hypothesis, again each M_i is equivalent to some $\bar{M}_i$, and we clearly have $x \in \mathsf{fv}(M_1)$. For M not to be a top-level (V, A)-redex, we must have that M_0 is neither a value nor another let-binding, only a (V, C, A, G)-normal application: in other words, it must be some $(\bar{H}\ \bar{N})$, and so we have $M \equiv \mathsf{let}\ x = (\bar{H}\ \bar{N})\ \mathsf{in}\ \bar{M}_1$.

□

Definition 18 ((V, C, A, G)-normalization relation)
For terms $M, N \in$ NEED, we write $M \xrightarrow[(V,C,A,G)\text{-nf}]{} N$ if $M \twoheadrightarrow_{(V,C,A,G)} N$ and N is a (V, C, A, G)-normal form.

The following three technical lemmas follow from the technical issues we detail in the proof of Theorem 3.

Lemma 19
For all $M \in$ NEED there exists a unique N such that $M \xrightarrow[(V,C,A,G)\text{-nf}]{} N$.

Proof
We modify the argument for the finiteness of call-by-need developments from Theorem 3. We use the same notion of weighted term and norm as above, but we take a different definition of decreasing weighting:

- All free variables x^i or $\underline{\mathbf{V}}x^i$ have decreasing weighting.
- An abstraction $\lambda x.\, \dot{M}$ has a decreasing weighting if $\dot{M}$ has a decreasing weighting.
- An application $\underline{\mathbf{I}}(\dot{M}\ \dot{N})$ has decreasing weighting if both $\dot{M}$ and $\dot{N}$ have decreasing weighting.
- A binding $(\mathsf{let}\ x = \dot{M}\ \mathsf{in}\ \dot{N})$ or $\underline{\mathbf{A}_n}(\mathsf{let}\ x = \dot{M}\ \mathsf{in}\ \dot{N})$ has decreasing weighting if:
  1. Both $\dot{M}$ and $\dot{N}$ have decreasing weighting.
  2. For all x^i or $\underline{\mathbf{V}}x^i$ in $\dot{M}$, we have $i > \|\dot{N}\|$.

The idea is that we track all let-bindings, but we do not ever create any new ones, since we are not interested in (I) steps. By manual analysis of the various combinations we can show that (V, C, A, G) reduction — marked or unmarked — of a term with decreasing weighting both strictly decreases the norm and retains a decreasing weighting. We can give any term a decreasing weighting with the same algorithm as above, and thus strong normalisation follows. $\square$

Lemma 20
Let $\sigma : L_0 \xrightarrow[(V,C,A,G)]{} L_1$, $(L_0, \mathcal{F}) \xrightarrow[\mathrm{cpl}]{} M$ where $\mathcal{F}$ marks (I) steps, and take N where $(L_1, \mathcal{F}/\sigma) \xrightarrow[\mathrm{cpl}]{} N$. Then $M \xrightarrow[(V,C,A,G)]{} N$.

Proof
For σ a (V, C, A) step the result is immediate from Theorem 3. Otherwise for a (G) step we have the result by induction on the size of $\mathcal{F}$ with a simple comparison of the relative positions of the redexes at each step. $\square$

Lemma 21
Let $L \xrightarrow[(V,C,A,G)\text{-nf}]{} \bar{L}$, $M \xrightarrow[(V,C,A,G)\text{-nf}]{} \bar{M}$ and let $\mathcal{F}$ index (I)-redexes in L such that $\sigma : (L, \mathcal{F}) \xrightarrow[\mathrm{cpl}]{} M$. Then $\bar{L} \xrightarrow[\mathrm{NEED}]{} \!\!\!\!\!\rightarrow \bar{M}$.

Proof
We strengthen the obvious fact that $\bar{L} \overset{\mathrm{NEED}'}{=} \bar{M}$. We fix τ as some (V, C, A, G)-reduction sequence from L to $\bar{L}$; then by induction on the length of τ, applying Lemma 20 at each step, we have a common reduct of $\bar{L}$ and M, which by Lemma 19 reduces to $\bar{M}$. $\square$

The condition guaranteed by the next result is stronger than just confluence for (I) steps: confluence tells us only that there exists some N such that both $\bar{L}$ and $\bar{M}$ reduce to N. This lemma asserts that this N is in fact equivalent to $\bar{M}$.

Corollary 22
Let $L \xrightarrow[(V,C,A,G)\text{-nf}]{} \bar{L}$, $M \xrightarrow[(V,C,A,G)\text{-nf}]{} \bar{M}$ and $L \xrightarrow[\mathrm{NEED}]{} M$. Then $\bar{L} \xrightarrow[\mathrm{NEED}]{} \!\!\!\!\!\rightarrow \bar{M}$.

Proof
If $L \xrightarrow[\mathrm{NEED}]{} M$ is a (V, C, A, G)-step, then the result is trivial since we have unique normal forms: thus $\bar{L} \equiv \bar{M}$. Otherwise, the result follows from the above lemma since the single step can be viewed as the complete development of a single (I)-redex. $\square$

Lemma 23
Let $\bar{M}$ be a (V, C, A, G)-normal NEED term where $\bar{M}^{\pitchfork} \equiv N \in \text{NAME}$, and let γ index a (β)-redex in N. Then there exists some δ indexing an (I)-redex in $\bar{M}$ such that $\gamma \in \{\delta\}^{\pitchfork}_{[\bar{M}]}$.

Proof
By induction on the structure of the term $\bar{M}$. When $\bar{M}$ is an application or abstraction the result follows directly from the induction hypothesis. For $\bar{M} \equiv \mathsf{let}\ x = \bar{H}\ \bar{M}_1\ \mathsf{in}\ \bar{M}_2$, we distinguish between two possibilities: whether the marked redex in N originates in $\bar{M}_1$ or in $\bar{M}_2$. The distinction is made by the following technical criterion: do there exist γ_0 and γ_1 such that $\gamma \equiv \gamma_0\gamma_1$ and $\bar{M}_2^{\pitchfork}|_{\gamma_0} \equiv x$? In other words, in indexing $\bar{M}_2^{\pitchfork}$ by γ, do we run into a reference to the let-bound variable somewhere along the indexing path?

- *Such γ_0, γ_1 exist.* Then the result follows by induction on $(\bar{H}\ \bar{M}_1)$ and γ_1.
- *No such γ_0, γ_1 exist.* Then clearly we have some δ such that $\bar{M}_2|_\delta$ is an application $\bar{V}\ \ \bar{N}$ and $\delta^{\pitchfork}_{\bar{M}_2} \equiv \gamma$. Moreover, $\bar{V}$ must be an abstraction and not a variable: if it were a variable, and for M to have a redex at γ, then it would be necessary that the variable be let-bound to an abstraction, which is impossible by (V)-normalisation. We also have that $\bar{V}$ is not a let-binding, since $\bar{M}$ is (C)-normal. So δ does in fact mark a redex in $\bar{M}$.

$\square$

Corollary 24
Let $\bar{M}$ be a (V, C, A, G)-normal NEED term where $\bar{M}^{\pitchfork} \equiv N \in \text{NAME}$, and let $\mathcal{F}$ index (β)-redexes in N. Then there exists some $\mathcal{G}$ indexing (I)-redexes in $\bar{M}$ such that $\mathcal{F} \subseteq \{\mathcal{G}\}^{\pitchfork}_{[\bar{M}]}$.

Proof
This $\mathcal{G}$ is just the union of the individual corresponding redexes for every member of $\mathcal{F}$ predicted by Lemma 23 above. $\square$

The main lemma of this section follows:

Lemma 25
Let $\bar{L}$ be a (V, C, A, G)-normal NEED term, and let $M, N \in \text{NAME}$ where

$$\begin{aligned} \sigma_0 : M &\xrightarrow[\text{NAME}]{\gamma} N\ . \\ \sigma_1 : M &\xrightarrow[\text{NAME}]{} \!\!\!\!\rightarrow \bar{L}^{\pitchfork} \end{aligned}$$

Then there exists some (V, C, A, G)-normal NEED term $\bar{L}_0$ such that $\bar{L} \xrightarrow[\text{NEED}]{}\!\!\!\!\rightarrow \bar{L}_0$ and $N \xrightarrow[\text{NAME}]{}\!\!\!\!\rightarrow \bar{L}_0^{\pitchfork}$:

$$\begin{tikzcd}
\bar{L} \arrow[r, dotted, two heads, "\text{NEED}"'] & \exists \bar{L}_0 \\[-20pt]
\bar{L}^{\pitchfork} & \bar{L}_0^{\pitchfork} \\
M \arrow[u, two heads, "\text{NAME}"] \arrow[r, "\gamma", "\text{NAME}"'] & N \arrow[u, dotted, two heads, "\text{NAME}"']
\end{tikzcd}$$

Proof

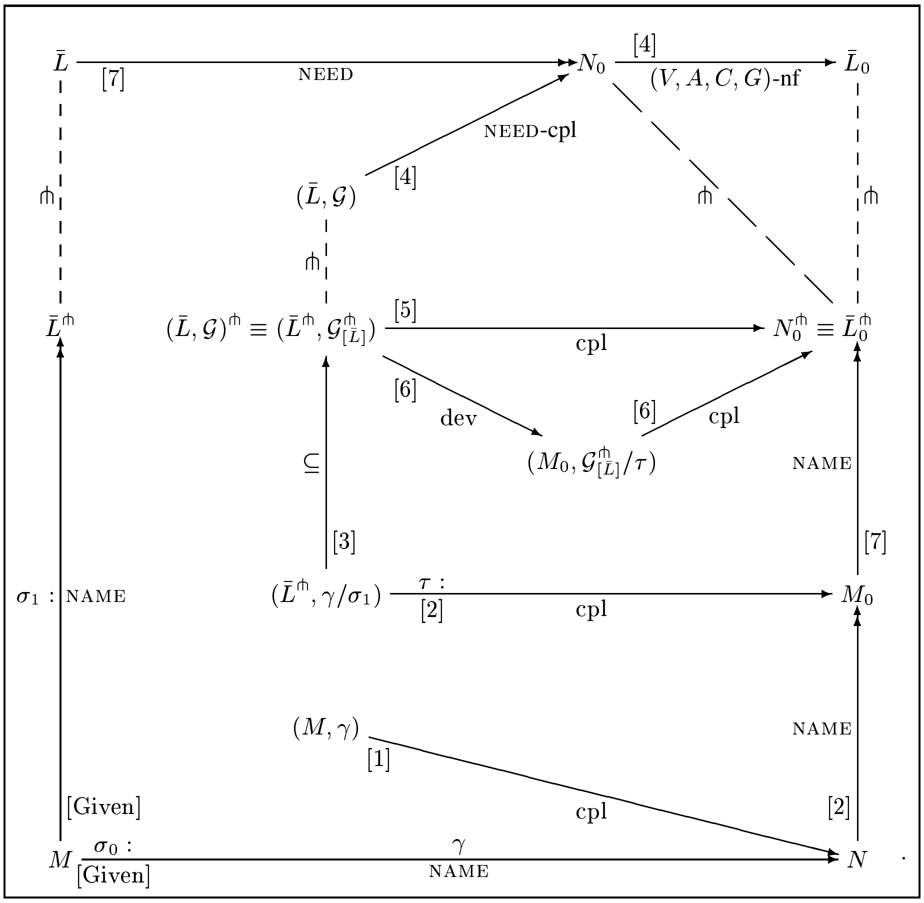

Fig. 10. Reasoning for the proof of Lemma 25. The numbers in square brackets refer to the sentence in the proof where the particular link is established. Except where explicitly indicated, all (complete) developments are NAME sequences.

[1]Since $M \xrightarrow[\text{NAME}]{} N$ in a single step, σ_0 may be viewed as a complete development of that single redex. [2]Then by Proposition 1, we have some $M_0 \in$ NAME such that $\tau : (\bar{L}^{\pitchfork}, \gamma/\sigma_1) \xrightarrow[\text{cpl}]{} M_0$ and $N \twoheadrightarrow M_0$. [3]By Lemma 24 we have some marking $\mathcal{G}$ of $\bar{L}$ such that $\gamma/\sigma_1 \subseteq \mathcal{G}^{\pitchfork}_{[\bar{L}]}$. [4]We take N_0 to be the result of the complete development of $\bar{L}$ by the (I)-redexes marked in $\mathcal{G}$, which is also unique (by Theorem 3), and which has some (V, A, C, G)-normal form $\bar{L}_0$ (by Lemma 19). [5]Since we can consider only marked (I) steps, by Lemma 21, we have $(\bar{L}, \mathcal{G})^{\pitchfork} \xrightarrow[\text{NAME-cpl}]{} \bar{L}_0^{\pitchfork}$ as well. [6]Since $\gamma/\sigma_1 \subseteq \mathcal{G}^{\pitchfork}_{[\bar{L}]}$, τ is a partial development of $(\bar{L}, \mathcal{G})^{\pitchfork}$ which can be extended again by finiteness of developments for call-by-name to a complete development ending in $\bar{L}_0^{\pitchfork}$. [7]So since developments can be projected to sequences in the unmarked calculi, we have that both $\bar{L} \xrightarrow[\text{NEED}]{}\!\!\!\!\twoheadrightarrow N_0 \xrightarrow[\text{NEED}]{}\!\!\!\!\twoheadrightarrow \bar{L}_0$ and $N \xrightarrow[\text{NAME}]{}\!\!\!\!\twoheadrightarrow M_0 \xrightarrow[\text{NAME}]{}\!\!\!\!\twoheadrightarrow \bar{L}_0^{\pitchfork}$. The reasoning is summarised in Figure 10, which refers to the sentence numbering. $\square$

Corollary 26
Let $\bar{L}$ be a (V, C, A, G)-normal NEED term, and let $M, N \in$ NAME where $M \xrightarrow[\text{NAME}]{}\!\!\!\!\twoheadrightarrow \bar{L}^{\pitchfork}$

and $M \underset{\mathrm{NAME}}{\twoheadrightarrow} N$. Then there exists some (V, C, A, G)-normal NEED term $\bar{L}_0$ such that $\bar{L} \underset{\mathrm{NEED}}{\twoheadrightarrow} \bar{L}_0$ and $N \underset{\mathrm{NAME}}{\twoheadrightarrow} \bar{L}_0^{\pitchfork}$:

$$\begin{tikzcd}
\bar{L} \arrow[r, dotted, two heads, "\mathrm{NEED}"'] & \exists \bar{L}_0 \\
\bar{L}^{\pitchfork} & \bar{L}_0^{\pitchfork} \\
M \arrow[u, two heads, "\mathrm{NAME}"'] \arrow[r, two heads, "\mathrm{NAME}"'] & N \arrow[u, dotted, two heads, "\mathrm{NAME}"']
\end{tikzcd} .$$

Proof
By the obvious induction using Lemma 25 at each step. □

Lemma 27 (Completeness of $\pitchfork$ for reduction with overshooting)
Let $M \in$ NEED and $N_0 \in$ NAME where $M^{\pitchfork} \underset{\mathrm{NAME}}{\twoheadrightarrow} N_0$. Then there exists some $N \in$ NEED such that $M \underset{\mathrm{NEED}}{\twoheadrightarrow} N$ and $N_0 \underset{\mathrm{NAME}}{\twoheadrightarrow} N^{\pitchfork}$.

Proof
Follows immediately from Corollary 26 by considering the (V, C, A, G)-normal forms $\bar{M}$ and $\bar{N}$ of M and N, respectively, which exist and are unique by Lemma 19; we have $\bar{M} \underset{\mathrm{NEED}}{\twoheadrightarrow} \bar{N}$ by Corollary 22. □

We can now prove the main equivalence result between call-by-name and call-by-need reduction.

Proposition 28 (Equivalence of call-by-name and call-by-need reduction)
The function $\pitchfork$ is sound and complete for mappings of NAME reduction sequences to NEED reduction sequences, where NEED sequences are allowed to "overshoot" NAME results:

$$\begin{tikzcd}
M \arrow[r, two heads, "\mathrm{NEED}"] & N \\
M^{\pitchfork} \arrow[r, dotted, two heads, "\mathrm{NAME}"] & N^{\pitchfork}
\end{tikzcd}
\qquad
\begin{tikzcd}
M \arrow[rr, dotted, two heads, "\mathrm{NEED}"] & & \exists N \\
M^{\pitchfork} \arrow[r, two heads, "\mathrm{NAME}"] & N_0 \arrow[r, dotted, two heads, "\mathrm{NAME}"] & N^{\pitchfork}
\end{tikzcd}$$

Proof
By Lemmas 16 and 27. □

The *convergence* relations $\Downarrow$ are defined in terms of whether the respective reduction relations lead from a term to a result, but do not consider the particular result.

Definition 29 (Convergence relations)
Let $M \in$ NAME and $N \in$ NEED.

1. We say that M converges in the call-by-name calculus, or $M \Downarrow_{\mathrm{NAME}}$, exactly when we have some abstraction $\lambda x.M_0$ such that $M \underset{\mathrm{NAME}}{\twoheadrightarrow} \lambda x.M_0$.
2. We say that N converges in the call-by-need calculus, or $N \Downarrow_{\mathrm{NEED}}$, exactly when we have some call-by-need answer A such that $M \underset{\mathrm{NEED}}{\twoheadrightarrow} A$.

Example 7
Let $\Omega \equiv (\lambda x.x\ x)\ (\lambda x.x\ x)$, $I \equiv (\lambda y.y)$ and $K \equiv (\lambda zw.z)$. Then $K\ I\ \Omega \Downarrow_{\mathrm{NAME}}$, since

$$\begin{aligned}
K\ I\ \Omega \quad &\equiv \quad (\lambda zw.z)\ I\ \Omega \\
&\underset{\mathrm{NAME}}{\longmapsto} \quad (\lambda w.I)\ \Omega \\
&\underset{\mathrm{NAME}}{\longmapsto} \quad I\ .
\end{aligned}$$

But adopting the convergence notation for call-by-value, we have $K\ I\ \Omega \not\Downarrow_{\text{VAL}}$, since

$$\begin{aligned} K\ I\ \Omega & \underset{\text{VAL}}{\longmapsto} (\lambda w.I)\ \Omega \\ & \underset{\text{VAL}}{\longmapsto} (\lambda w.I)\ \Omega \end{aligned}$$

and so on.

Proposition 28 gives us a straightforward relationship between the two convergence relations:

Corollary 30 (Convergence in call-by-name and call-by-need)
For all $M \in \Lambda_{\text{NEED}}$,

$$M \Downarrow_{\text{NEED}} \quad \text{if and only if} \quad M^{\pitchfork} \Downarrow_{\text{NAME}} \quad .$$

Proof
By Proposition 28 and Lemma 12.(3). $\square$

4.3 Relating observational equivalences

Observational equivalence is the coarsest equivalence relation over terms that still distinguishes between terms with different observational behaviour. Formally:

Definition 31 (Observational equivalence relations)
Two terms M, N of a language $\mathcal{L}$ are *observationally equivalent* under a convergence theory $\Downarrow_R$, written $M \cong_R N$, if and only if for all $\mathcal{L}$-contexts C such that $C[M]$ and $C[N]$ are closed,

$$C[M] \Downarrow_R \quad \text{if and only if} \quad C[N] \Downarrow_R \quad .$$

Example 8
It is trivially true that all reduction-related terms of the calculi we consider are observationally equivalent: if $L_0 \twoheadrightarrow L_1$ and $C[L_0] \twoheadrightarrow A_0$, then clearly $C[L_0] \twoheadrightarrow C[L_1]$, and

- By confluence there is some M_0 such that $\tau : A_0 \twoheadrightarrow M_0$ and $C[L_1] \twoheadrightarrow M_0$.
- By the first itemised property in the proof of 10 and τ, M_0 must also be an answer.

so $C[L_1] \Downarrow$ as well.

The converse is simpler; if $L_0 \twoheadrightarrow L_1$ and $C[L_1] \twoheadrightarrow A_0$, then $C[L_0] \twoheadrightarrow C[L_1] \twoheadrightarrow A_0$ as well. So for example, $K\ I\ \Omega \cong_{\text{NAME}} I$. Taking $C \equiv [\,]$ it is clear that $K\ I\ \Omega \ncong_{\text{VAL}} I$, but a simple structural induction reveals that $K\ I\ \Omega \cong_{\text{VAL}} \Omega$.

Corollary 30 implies that λ_{NEED} is a conservative observational extension of λ_{NAME}:

Theorem 32 (Observational equivalences in call-by-name and call-by-need)
The observational equivalence theories of λ_{NAME} and λ_{NEED} coincide on Λ_{NAME}. For all terms $M, N \in \Lambda_{\text{NAME}}$,

$$M \cong_{\text{NAME}} N \quad \text{if and only if} \quad M \cong_{\text{NEED}} N \quad .$$

Proof

"⇒": Assume $M \cong_{\text{NAME}} N$ and let C be a λ_{NEED}-context such that $C[M]$ and $C[N]$ are closed. Let $C^{\#}$ result from C by eliminating all let's in C using rule (I) repeatedly in reverse. Then

$$
\begin{array}{lll}
 & C[M] \Downarrow_{\text{NEED}} & \\
\Leftrightarrow & C^{\#}[M] \Downarrow_{\text{NEED}} & \text{since } C^{\#}[M] \overset{\text{NEED}}{=} C[M] \\
\Leftrightarrow & C^{\#}[M] \Downarrow_{\text{NAME}} & \text{by Corollary 30, since } (C^{\#}[M])^{\pitchfork} \equiv C^{\#}[M] \\
\Leftrightarrow & C^{\#}[N] \Downarrow_{\text{NAME}} & \text{since } M \cong_{\text{NAME}} N \\
\Leftrightarrow & C[N] \Downarrow_{\text{NEED}} & \text{by the reverse argument on Corollary 30 .}
\end{array}
$$

"⇐": Symmetrically, with C instead of $C^{\#}$, and leaving out the first step in the equivalence chain. □

Corollary 33
The rule β is an observational equivalence in λ_{NEED}: For all $M, N \in \Lambda_{\text{NEED}}$,

$$(\lambda x.M)\ N \cong_{\text{NEED}} [M/x]N \ .$$

Proof
Let $M, N \in \Lambda_{\text{NEED}}$. Let M_0, N_0 be the corresponding Λ_{NAME}-terms that result from eliminating all let's in M, N by performing (I) reductions in reverse. Then we have in λ_{NEED}:

$$(\lambda x.M)N \ = \ (\lambda x.M_0)N_0 \ \cong \ [N_0/x]M_0 \ = \ [N/x]M$$

where "≅" follows from Theorem 32. □

5 Natural semantics

This section presents an operational semantics for call-by-need in the natural semantics style of Plotkin and Kahn, similar to one given by Launchbury (1993). The natural semantics is closely related to the standard reduction order we presented above.

A *heap* abstracts the state of the store at a point in the computation. It consists of a sequence of pairs binding variables to terms,

$$x_1 \mapsto M_1, \ldots, x_n \mapsto M_n.$$

The order of the sequence of bindings is significant: all free variables of a term must be bound to the left of it, *i.e.* a term M_i may contain as free variables only $x_1, \cdots, x_{i-1}$.. Furthermore, all variables bound by the heap must be distinct. Thus the heap above is well-formed if fv$(M_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for each i in the range $1 \le i \le n$, and all the x_i are distinct. Let Φ, Ψ, Υ range over heaps. If Φ is the heap $x_1 \mapsto M_1, \ldots, x_n \mapsto M_n$, define vars$(\Phi) = \{x_1, \ldots, x_n\}$. A configuration pairs a heap with a term, where the free variables of the term are bound by the heap. Thus $\langle\Phi\rangle\, M$ is well-formed if Φ is well-formed and fv$(M) \subseteq$ vars(Φ). The operation of evaluation takes configurations into configurations. The term of the final configuration is always a value. Thus evaluation judgements take the form $\langle\Phi\rangle\, M \Downarrow \langle\Psi\rangle\, V$.

The rules defining evaluation are given in Figure 11. There are three rules, for identifiers, abstractions and applications.

$$\text{Id}\ \frac{\langle\Phi\rangle\, M \Downarrow \langle\Psi\rangle\, V}{\langle\Phi,\, x \mapsto M,\, \Upsilon\rangle\, x \Downarrow \langle\Psi,\, x \mapsto V,\, \Upsilon\rangle\, V}$$

$$\text{Abs}\ \frac{}{\langle\Phi\rangle\, \lambda x.\, N \Downarrow \langle\Phi\rangle\, \lambda x.\, N}$$

$$\text{App}\ \frac{\langle\Phi\rangle\, L \Downarrow \langle\Psi\rangle\, \lambda x.\, N \qquad \langle\Psi,\, x' \mapsto M\rangle\, [x'/x]N \Downarrow \langle\Upsilon\rangle\, V \qquad x' \text{ fresh}}{\langle\Phi\rangle\, L\ M \Downarrow \langle\Upsilon\rangle\, V}$$

Fig. 11. Operational semantics of call-by-need lambda calculus.

- Abstractions are trivial. As abstractions are already values, the heap is left unchanged and the abstraction is returned.
- Applications are straightforward. We evaluate the function to yield a lambda abstraction, extend the heap so that the the bound variable of the abstraction is bound to the argument, and then evaluate the body of the abstraction. In this rule, x' is a new name not appearing in Ψ or N. The renaming guarantees that each identifier in the heap is unique.
- Variables seem more subtle, but the basic idea is straightforward: we find the term bound to the variable in the heap, evaluate the term, then update the heap to bind the variable to the resulting value. Some care is required to ensure that the heap remains well-formed. The original heap is partitioned into $\Phi,\, x \mapsto M,\, \Upsilon$. Since the heap is well-formed, only Φ is required to evaluate M. Evaluation yields a new heap Ψ and value V. The new heap Ψ will differ from the old heap Φ in two ways: bindings may be updated (by Var) and bindings may be added (by App). The free variables of V are bound by Ψ, so to ensure the heap stays well-formed, the final heap has the form $\Psi,\, x \mapsto V,\, \Upsilon$. Note that this last statement implies that any new bindings added into Ψ will use fresh variables which are not also used in Υ.

A semantics of let terms can be derived from the above rules: the semantics of $\mathsf{let}\ x = M\ \mathsf{in}\ N$ is identical to the semantics of $(\lambda x.\, N)\ M$.

As one would expect, evaluation uses only well-formed configurations, and evaluation only extends the heap.

Lemma 34
Given an evaluation tree with root $\langle\Phi\rangle\, M \Downarrow \langle\Psi\rangle\, V$, if $\langle\Phi\rangle\, M$ is well-formed then every configuration in the tree is well-formed, and furthermore vars$(\Phi) \subseteq$ vars(Ψ).

Thanks to the care taken to preserve the ordering of heaps, it is possible to draw a close correspondence between evaluation and standard reductions. If Φ is the heap $x_1 \mapsto M_1,\, \ldots,\, x_n \mapsto M_n$, write $\mathsf{let}\ \Phi\ \mathsf{in}\ N$ for the term

$$\mathsf{let}\ x_1 = M_1\ \mathsf{in}\ \cdots\ \mathsf{let}\ x_n = M_n\ \mathsf{in}\ N.$$

Every answer A can be written $\mathsf{let}\ \Psi\ \mathsf{in}\ V$ for some heap Ψ and value V. Then a simple induction on $\Downarrow$-derivations yields the following result.

Proposition 35
For all heaps Φ, Ψ, terms M and values V,

$$\langle\Phi\rangle\, M \Downarrow \langle\Psi\rangle\, V \text{ if and only if } \mathsf{let}\ \Phi\ \mathsf{in}\ M \underset{\text{NEED}}{\longmapsto\!\!\!\!\rightarrow} \mathsf{let}\ \Psi\ \mathsf{in}\ V\ .$$

Syntactic Domains

Variables	x, y, z		
Values	V, W	$::=$	$x \mid \lambda x.M$
Terms	L, M, N	$::=$	$V \mid M\ N$
Answers	A, A_i	$::=$	$\lambda x.M \mid (\lambda x.A)\ M$
Evaluation Contexts	E, E_i	$::=$	$[\,] \mid E\ M \mid (\lambda x.E)\ M$ $\mid (\lambda x.E_0[x])\ E_1$

General Reduction Rules

$$
\begin{array}{lllll}
(V^\ell) & (\lambda x.C[x])\ V & \rightarrow & (\lambda x.C[V])\ V & \\
(C^\ell) & (\lambda x.L)MN & \rightarrow & (\lambda x.LN)M & \\
(A^\ell) & (\lambda x.L)((\lambda y.M)N) & \rightarrow & (\lambda y.(\lambda x.L)M)N & \\
(G^\ell) & (\lambda x.M)\ N & \rightarrow & M & \textbf{if } x \notin \mathrm{fv}(M)
\end{array}
$$

Standard Reduction Rules

$$
\begin{array}{llll}
(V_s^\ell) & (\lambda x.E[x])\ (\lambda y.M) & \mapsto & (\lambda x.E[(\lambda y.M)])\ (\lambda y.M) \\
(C_s^\ell) & (\lambda x.A)MN & \mapsto & (\lambda x.AN)M \\
(A_s^\ell) & (\lambda x.E[x])((\lambda y.A)N) & \mapsto & (\lambda y.(\lambda x.E[x])A)N
\end{array}
$$

Fig. 12. The let-less call-by-need calculus.

The semantics given here is similar to that presented by Launchbury (1993). An advantage of our semantics over Launchbury's is that the form of terms is standard, and care is taken to preserve ordering in the heap. Launchbury uses a non-standard syntax, in order to achieve a closer correspondence between terms and evaluations: in an application the argument to a term must be a variable, and all bound variables must be uniquely named. Here, general application is supported directly and all renaming occurs as part of the application rule. It is interesting to note that Launchbury presents an alternative formulation quite similar to ours, buried in one of his proofs.

An advantage of Launchbury's semantics over ours is that his copes more neatly with recursion, by the use of multiple, recursive let bindings. An extension of our semantics to include recursion (Ariola and Felleisen, 1994, for example) would lose the ordering property of the heap, and hence lose the close connection to standard reductions (Mossin *et al.*, 1995). We discuss other extensions for recursion below.

6 Call-by-need without bindings

In the call-by-name calculus, we have related $(\mathsf{let}\ x = M\ \mathsf{in}\ N)$ to $((\lambda x.N)\ M)$ by an explicit reduction rule: but are let-bindings really essential? It turns out that they are not; we can take the conversion to be a syntactic identity, and thus expel the bindings from call-by-need. We call the resulting calculus $\lambda^\ell_{\text{NEED}}$ (reading the ℓ as "let-less"). Its notions of general and standard reduction are shown in Figure 12. We define convergence $\Downarrow_{\text{NEED}^\ell}$ and observational equivalence $\cong_{\text{NEED}^\ell}$ in the new system as usual.

While $\lambda^\ell_{\text{NEED}}$ is perhaps somewhat less intuitive than λ_{NEED}, its simpler syntax makes some of the basic (syntactic) results easier to derive. It also allows better comparison with the call-by-name calculus, since no additional syntactic constructs are introduced.

Clearly, λ_{NEED} and $\lambda^\ell_{\text{NEED}}$ are closely related. More precisely, the following theorem states that reduction in λ_{NEED} can be simulated in $\lambda^{\ell}_{\text{NEED}}$, and that the converse is also true, provided we identify terms that are equal up to (I) introduction.

Proposition 36
For all $M_0 \in \Lambda^{\ell}_{\text{NEED}}$, $M_1 \in \Lambda_{\text{NEED}}$,

$$\begin{tikzcd} M_0 \arrow[r, twoheadrightarrow, "\text{NEED}^{\ell}"] \arrow[d, "(I)"'] & N_0 \arrow[d, dotted, "(I)"] \\ M_1 \arrow[r, dotted, twoheadrightarrow, "\text{NEED}"] & N_1 \end{tikzcd} \qquad \begin{tikzcd} M_0 \arrow[r, dotted, twoheadrightarrow, "\text{NEED}^{\ell}"] \arrow[d, "(I)"'] & N_0 \arrow[d, dotted, "(I)"] \\ M_1 \arrow[r, twoheadrightarrow, "\text{NEED}"] & N_1 \end{tikzcd} \quad .$$

Proposition 36 can be used to derive the essential syntactic properties of $\lambda^{\ell}_{\text{NEED}}$ from those of λ_{NEED}:

Theorem 37
Reduction in $\lambda^{\ell}_{\text{NEED}}$ is Church Rosser.

Theorem 38
The relation $\xmapsto[\text{NEED}^{\ell}]{}$ is a standard reduction relation for $\lambda^{\ell}_{\text{NEED}}$. For all terms M and answers $A \in \Lambda^{\ell}_{\text{NEED}}$,

- **Soundness.** If $M \mapsto\!\!\!\twoheadrightarrow A$ then $M \twoheadrightarrow A$.
- **Completeness.** If $M \twoheadrightarrow A$ then there exists some answer $A_0 \in \Lambda^{\ell}_{\text{NEED}}$ such that $M \mapsto\!\!\!\twoheadrightarrow A_0$.

The let-less calculus $\lambda^{\ell}_{\text{NEED}}$ has close relations to both the call-by-value calculus λ_{VAL} and the call-by-name calculus λ_{NAME}. Its notion of equality $=_{\lambda^{\ell}_{\text{NEED}}}$ — *i.e.* the least equivalence relation generated by the reduction rules — fits between those of the other two calculi, making $\lambda^{\ell}_{\text{NEED}}$ an extension of λ_{VAL} and λ_{NAME} an extension of $\lambda^{\ell}_{\text{NEED}}$.

Theorem 39

$$=_{\lambda_{\text{VAL}}} \;\subset\; =_{\lambda^{\ell}_{\text{NEED}}} \;\subset\; =_{\lambda_{\text{NAME}}} \quad .$$

Proof
Rule β_V can be expressed by a series of (I, V, G) steps, as shown in Example 9, so we have $=_{\lambda_{\text{VAL}}} \subseteq =_{\lambda^{\ell}_{\text{NEED}}}$. To show that the inclusion is proper, we take Ω to be the usual divergent expression

$$\Omega \equiv (\lambda x.x\ x)\ (\lambda x.x\ x) \quad ,$$

and have

$$(\lambda x.x)\ ((\lambda y.y)\ \Omega) \quad = \quad (\lambda y.(\lambda x.x)\ y)\ \Omega$$

by the (A^{ℓ}) rule; this equality does not hold in call-by-value, so $=_{\lambda_{\text{VAL}}} \subset =_{\lambda^{\ell}_{\text{NEED}}}$.

For the second inclusion, we can see that each $\lambda^{\ell}_{\text{NEED}}$ reduction rule is an equality in λ. For instance, in the case of (V^{ℓ}) we have:

$$(\lambda x.C[x])\ V \quad =_{\beta} \quad [V/x](C[x]) \quad \equiv \quad [V/x](C[V]) \quad =_{\beta} \quad (\lambda x.C[V])\ V \quad .$$

The other rules have equally simple translations; the left- and right-hand sides of the axioms always have a common (β)-reduct which can be constructed by contracting the applications mentioned in the rules, and identifying the two sides based on the (non-)occurrence of substituted variables in certain subexpressions. Thus we have $=_{\lambda^\ell_{\text{NEED}}} \subseteq =_{\lambda_{\text{NAME}}}$. For the proper inclusion, we have the following instance of β which is not an equality in $\lambda^\ell_{\text{NEED}}$:

$$(\lambda x.x)\ \Omega \quad = \quad \Omega\ ,$$

and so $=_{\lambda^\ell_{\text{NEED}}} \subset =_{\lambda_{\text{NAME}}}$. $\square$

As in the calculus with bindings, one can show that the observational equivalence theories of $\lambda^\ell_{\text{NEED}}$ and λ_{NAME} are identical; the proof is by a simple application of Theorem 32 together with Proposition 36. The observational equivalence theories of both $\lambda^\ell_{\text{NEED}}$ and λ_{NAME} are incompatible with the theory for λ_{VAL}.

Theorem 40
For all terms $M, N \in \Lambda$,

$$M \cong_{\text{NAME}} N \quad \Longleftrightarrow \quad M \cong_{\text{NEED}^\ell} N\ .$$

Theorem 39 implies that any model of the call-by-name calculus is also a model of $\lambda^\ell_{\text{NEED}}$, since it validates all equalities in $\lambda^\ell_{\text{NEED}}$. Theorem 40 implies that any adequate (respectively fully-abstract) model of λ_{NAME} is also adequate (fully-abstract) for $\lambda^\ell_{\text{NEED}}$, since the observational equivalence theories of both calculi are the same. For instance, Abramsky and Ong's adequate model of the lazy lambda calculus (Abramsky, 1990) is also adequate for $\lambda^\ell_{\text{NEED}}$.

7 Extensions

The formulation of call-by-need we have reviewed is rather basic, and lacks a number of common syntactic conveniences, which we consider now. In Section 7.1 we consider the algebraic data types which are central to elegance of real functional programs. Section 7.2 discusses how we can include constants and primitive functions to the calculus. One also often considers recursive let-bindings; we do not consider recursion is detail here, but sketch a number of others' approaches in the conclusion.

7.1 Constructors and selectors

Functional programs rely in an essential way on distinguishable tagged packages of informations. The ubiquitous list is one such datatype with two such constructors, Cons and Nil. The former tag accompanies two items, the head and tail of the list; the latter tag is unaccompanied.

Of course, these additions can be simulated in the base language via Church encodings, but a more high-level treatment is often desirable for reasons of both clarity and efficiency. The syntax and semantics of the extension are shown in Figure 13; we write $\vec{S}$ to abbreviate many occurrences of S, and let $\mathsf{let}\ \vec{x} = \vec{M}\ \mathsf{in}\ N$ as an abbreviation for

$$\mathsf{let}\ x_1 = M_1\ \mathsf{in}\ \cdots\ \mathsf{let}\ x_{a_i} = M_{a_i}\ \mathsf{in}\ N\ .$$

Syntactic Domains

Terms	L, M, N	$::=$	$\cdots \mid \mathsf{case}\ M\ \mathsf{in}\ \langle\ \vec{S}\ \rangle$	
			$\mid K_i\ M_1\ ...\ M_{(a_i)}$	$(a_i \geq 0)$
Clauses	S	$::=$	$K_i\ \vec{x}_1^{a_i}.M$	
Answers	A	$::=$	$\cdots \mid K_i\ M_1\ ...\ M_{(a_i)}$	$(a_i \geq 0)$
Evaluation contexts	E	$::=$	$\cdots \mid \mathsf{case}\ E\ \mathsf{in}\ \langle\ \vec{S}\ \rangle$	

Additional Reduction Rules

(I^K)	$\mathsf{case}\ K_i\ \vec{M}\ \mathsf{in}\ \langle \cdots,\ K_i\ \vec{x}.N,\ \cdots \rangle$	$\to$	$\mathsf{let}\ \vec{x} = \vec{M}\ \mathsf{in}\ N$
(V^K)	$\mathsf{let}\ x = K_i\ \vec{M}\ \mathsf{in}\ N$	$\to$	$\mathsf{let}\ \vec{y} = \vec{M}\ \mathsf{in}\ N[x := K_i\vec{y}]$
(A^K)	$\mathsf{case}\ (\mathsf{let}\ x = M\ \mathsf{in}\ N)\ \mathsf{in}\ \langle\ \vec{S}\ \rangle$	$\to$	$\mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{case}\ N\ \mathsf{in}\ \langle\ \vec{S}\ \rangle$

Additional Evaluation Rules

(I_s^K)	$\mathsf{case}\ K_i\ \vec{M}\ \mathsf{in}\ \langle \cdots,\ K_i\ \vec{x}.N,\ \cdots \rangle$	$\mapsto$	$\mathsf{let}\ \vec{x} = \vec{M}\ \mathsf{in}\ N$
(V_s^K)	$\mathsf{let}\ x = K_i\ \vec{M}\ \mathsf{in}\ E[x]$	$\mapsto$	$\mathsf{let}\ \vec{y} = \vec{M}\ \mathsf{in}\ (E[x])[x := K_i\vec{y}]$
(A_s^K)	$\mathsf{case}\ (\mathsf{let}\ x = M\ \mathsf{in}\ A)\ \mathsf{in}\ \langle\ \vec{S}\ \rangle$	$\mapsto$	$\mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{case}\ A\ \mathsf{in}\ \langle\ \vec{S}\ \rangle$

Fig. 13. Data constructors and selectors.

In a tagged expression, a tag K_i expects a_i component items. We distinguish between different tags and access their components via a case expression. A clause S of a case expression has the form

$$K_i\, x_1\ \cdots\ x_{a_i}.\, M_i\ \ .$$

A case expression then consists of one subexpression to be considered, plus a series of clauses of distinct constructors:

$$\mathsf{case}\ M\ \mathsf{in}\ \langle\ S_1,\ S_2,\ \ldots\ ,\ S_n\ \rangle\ \ .$$

Reduction of the case statement involves matching the constructor of the subterm M. Since we do not want to force the constructor subterms to be evaluated until they are individually demanded, we create new bindings to the pattern variables in the (I^K) rule:

$$\begin{array}{rl} & \mathsf{case}\ (K_i\, M_1\ \cdots\ M_{a_i})\ \mathsf{in}\ \langle \cdots,\ K_i\, x_1\ \cdots\ x_{a_i}.N,\ \cdots \rangle \\ \to & \mathsf{let}\ x_1 = M_1\ \mathsf{in}\ \cdots\ \mathsf{let}\ x_{a_i} = M_{a_i}\ \mathsf{in}\ N\ \ . \end{array}$$

The (V^K) rule facilitates let-bound constructor expressions, again creating bindings for the subexpressions rather than duplicating them in the substitution. The separate rules (V) for freely copyable values (abstractions and variables) and (V^K) for constructor terms is awkward, but avoids the need for separate tags which indicate whether the subexpressions are copyable. One might further refine this scheme by including $K_i\,\vec{V}$ as a value, and restricting (V^K) to the case where at least one of the subexpressions is not a value. Finally we also have a new structural rule (A^K), which allows us to rearrange a let-binding in the term under examination.

Most other formalisations of call-by-need, including the representation of terms for the STG machine (Peyton Jones, 1992), Launchbury's natural semantics (Launchbury, 1993), and our earlier work on the subject with Ariola and Felleisen (1995), restrict constructor subcomponents to either variables or values, and copy the subcomponents in the rule analogous to (V^K). In our (evaluation) rules, the case expression is both an evaluator of its

Syntactic Domains

Constants and functions	c, p		
Values	V, W	$::=$	$\lambda x.M \mid c$
Evaluation contexts	E	$::=$	$\cdots \mid p\ E$

Additional Reduction Rules

$$(\delta) \qquad p\ c \ \rightarrow\ \delta(p, c) \quad \delta(p, c) \text{ defined}$$

Additional Evaluation Rules

$$\begin{array}{lll} (G_s) & \mathsf{let}\ x = M\ \mathsf{in}\ c & \mapsto\ c \\ (\delta_s) & p\ c & \mapsto\ \delta(p, c) \quad \delta(p, c) \text{ defined} \end{array}$$

Fig. 14. Constants and primitive functions.

subterm and a memory allocator for the new let-bound terms. Since the STG machine is intended to directly reflect low-level details of an actual compilation, a more orthogonal design is appropriate. In the STG machine the case expression is essentially just a subroutine call to evaluate the subterm, and only case expressions correspond to such subroutine calls. Likewise, STG let-bindings suggest only memory allocation on the heap, and no other construct allocates heap space. Thus it is desirable in the STG machine to restrict the subcomponents to variables, and assume the presence of some preprocessor which repeatedly lifts out non-variable subcomponents via let-bound variables. The other two approaches follow this implementation philosophy, but for a general calculus the restriction is rather artificial.

7.2 Constants and primitive functions

A further aspect of real functional programming languages is the inclusion of constants and primitive function in the language. Like constructors and selectors, constants and primitive functions may simply be Church-encoded, but again at the cost of readability and a distortion of the actual effort required in program reduction as compared to the actual implementation.

Figure 14 describes the extension of the call-by-need calculus for constants. Following Plotkin, we add a set of unique names to the set of values, and assume the existence of some (probably partial) function δ from pairs of these names to names. We let c, p range over these constants, generally using p to refer to constants used as functions. We let A range as usual over abstractions possibly under bindings, although the result below deals with observation of constants rather than these “answer” closures. Thus as discussed in Section 8, garbage collection becomes essential in evaluation to constants. The following result relates reduction to basic elements in call-by-need and in call-by-name. The result is an easy extension of Proposition 28, and relies on (G) to discard unneeded bindings from around the primitive in the call-by-need sequence.

Corollary 41
For all terms $M \in \Lambda$ with primitives and constants c,

$$M \xrightarrow[\text{NEED}]{} \!\!\!\!\!\rightarrow c \quad \Longleftrightarrow \quad M \xrightarrow[\text{NAME}]{} \!\!\!\!\!\rightarrow c\ .$$

8 Concluding remarks

We conclude with a discussion of our call-by-need calculus in relation to a number of other systems and notion of reduction.

On other formulations of call-by-need. Josephs (1989) gives a continuation- and store-based denotational semantics of lazy evaluation. Purushothaman and Seaman (1992) give a structured operational semantics of call-by-name PCF with explicit environments that is then shown to be equivalent to a standard denotational semantics for PCF. Launchbury (1993) presents a system with a simpler operational semantics and gives in addition rules for recursive let-bindings that capture call-by-need sharing behaviour. The key point about all this work is that while it does provide an operational model of call-by-need, it does not provide anything like a calculus or a reduction system for equational reasoning.

In work done independently of ours, Ariola and Felleisen proposed a similar calculus (1994). We have taken the position that call-by-need, in a general sense, should unite the observational behaviour of call-by-name with the restrictions on copying of call-by-value. Thus since none of the (V, C, A, G) rules copy top-level non-values, and since (as we see in the next section) they do preserve call-by-name observations, it is appropriate to adopt the rules without restriction. Ariola and Felleisen take a narrower view of what one should permit within a reduction relation, and interpret the "need" in call-by-need literally. Their system can be characterised as the relation (I_s, V_s, C_s, A_s) compatibly closed under all contexts. In other words, the restriction to subexpressions which we impose only in standard reduction rules, they impose universally. Their calculus captures only "the intentional aspects of modern call-by-need evaluators," which we find appropriate for the standard reduction relation but too restrictive for the general calculus. Their system proves fewer program transformations as equalities, requiring instead the more difficult notion of observational equivalence. It is interesting to note that Ariola and Felleisen's summary of Plotkin's criteria for the development of calculi to capture language properties (1975) does note that "the equations of [a] calculus should identify terms that are 'observationally indistinguishable' from each other;" as such we feel that our system more closely adheres to Plotkin's program. However it should be noted that without their restrictions, confluence may be lost when extending the system for mutually recursive bindings, which we address as a separate point below; this point is certainly one advantage of their formulation. Ariola and Felleisen's restriction to the bodies of the general rules does streamline the transition from general to standard reduction, since one needs only to consider an alternate notion of compatible closure, under evaluation rather than arbitrary contexts. Our system differs further from Ariola and Felleisen's in our inclusion of a rule for garbage collection, which we also discuss separately below.

Ariola and Felleisen also raise the somewhat more practical possibility that their system admits easier proofs of the various syntactic properties. Strictly speaking this claim is not invalid; their restriction of the general reduction rules allows results on certain classes of term rewriting systems to be applied directly, making confluence immediate. While our results on developments are somewhat less immediate, once proven the same main syntactic results are in fact straightforward; the same results on developments were also quite useful in the proofs about the observational equivalence theories, whereas with Ariola and

Felleisen's weaker notion of marked redexes an additional layer of diagramming notation is necessary. Although we do believe that the technical results we present allow a more systematic technical exposition, this issue is separate from the question of which formulation of the general reduction rules is more appropriate.

On call-by-need and explicit substitutions. At first glance the call-by-need system seems to be little aside from yet one more formulation of explicit substitutions (Abadi *et al.*, 1990, for example). However, the assumptions made by explicit substitution schemes regarding what the "expensive" operation is in reduction are different. Explicit substitution schemes track substitutions through a term, but do not place any restrictions on the duplication of substitutions. As suggested by their name, the explicit steps of pushing a substitution through the structure of a term, plus accounting for the interaction of unpropagated substitutions with other structures, is the difference with an implicit formulation. In our call-by-need scheme, we have no interest in how substitutions move through the term, but rather under what circumstances substitutions — implicit or explicit — may be created in a term. A clear advantage of call-by-need over explicit substitutions is simplicity; explicit substitution schemes have considerably more rules, and correspondingly one has more difficulty in establishing its syntactic properties.

Benaisaissa, Lescanne and Rose (1996) have presented a hybrid system which incorporates sharing, explicit substitutions and explicit address references, and which is quite useful for expressing space complexity. Their system is quite general, and can simulate ours, as well as a number of other interesting systems, as a subset of its rules, but as a result is a rather large, complex system. The particular calculus which they present allows weak reduction only, but is easily generalised to allow reduction in any context (Rose, private communication).

On call-by-need, full laziness and optimal reduction. Although we allow only values to replace a variable in a substitution, it is not true that only values are ever copied. In the contraction

$$\begin{array}{rcl} M & \equiv & \mathsf{let}\ x = (\lambda y.M_0\ y\ y)\ \mathsf{in}\ C[x] \\ & \to & \mathsf{let}\ x = (\lambda y.M_0\ y\ y)\ \mathsf{in}\ C[\lambda y.M_0\ y\ y]\ \ , \end{array}$$

the subexpression $(M_0\ y\ y)$ is obviously not a value, but is nonetheless copied. A number of issues apply to this situation, but the motivation behind our formulation is the behaviour of graph reduction implementations of lazy functional languages in the style of the G-machine and its descendants. In these designs, lambda abstractions correspond to subroutines in the machine code, *i.e.* simple addresses which may be copied freely. The (V) rule is faithful to this design principle: we replace a reference to x with a reference to code which will seek an argument y and then construct the graph of $M_0\ y\ y$.

We have explicitly declined certain opportunities for greater sharing. In the above example, if y does not occur in M_0 then a more space-efficient representation of M could be written as

$$N \equiv \mathsf{let}\ f = M_0\ \mathsf{in}\ \mathsf{let}\ x = (\lambda y.f\ y\ y)\ \mathsf{in}\ C[x]\ \ .$$

Rather than reducing M to N at runtime, we view the conversion of M to N as appropriate to a transformation carried out *before* program execution. In particular, the full laziness transformation enables sharing of such subterms (Wadsworth, 1971; Hughes, 1983).

Even after a full laziness pass, we would still copy the non-value $(f\ y\ y)$:

$$\begin{array}{ll} & \mathsf{let}\ f = M_0\ \mathsf{in}\ \mathsf{let}\ x = (\lambda y.f\ y\ y)\ \mathsf{in}\ C[x] \\ \rightarrow & \mathsf{let}\ f = M_0\ \mathsf{in}\ \mathsf{let}\ x = (\lambda y.f\ y\ y)\ \mathsf{in}\ C[\lambda y.f\ y\ y]\ . \end{array}$$

Such expressions are indeed copied in lazy functional graph reduction implementations, and we do not view this effect as a shortcoming. Sharing of subterms across different instantiations of bound variables is addressed by *optimal* reduction strategies (Lévy, 1980; Lamping, 1990; Field, 1990; Abadi *et al.*, 1990; Maranget, 1991). Although the additional sharing of those calculi does allow the fewest possible reduction steps, it is not clear how useful optimal reduction is for compilation to efficient low-level code.

Yoshida (1993) presents a weak lambda calculus with explicit environments similar to let constructs, and gives an optimal reduction strategy. Her calculus subsumes several of our reduction rules as structural equivalences. However, due to a different notion of observation, reduction in this calculus is not equivalent to reduction to weak head-normal form.

On call-by-need and generalisations of classical β-reduction. Much work exists in discovering future redexes which are simply blocked by another contraction which has not yet occurred. For example, in the term

$$(\lambda x.\lambda y.L)\ M\ N$$

it is clear that the occurrences of y in L will be replaced by N, but that substitution will not be possible until we have first replaced x with M. Nederpelt proposed a notion of generalised β reduction $\hookrightarrow$ which allows this contraction to occur at once (Nederpelt, 1973):

$$\begin{array}{rcl} (\lambda x_1.\lambda x_2.L)\ M_1\ M_2 & \hookrightarrow & (\lambda x_1.L[x_2 := M_2])\ M_1 \\ (\lambda x_1.\lambda x_2.\lambda x_3.L)\ M_1\ M_2\ M_3 & \hookrightarrow & (\lambda x_1.\lambda x_2.L[x_3 := M_3])\ M_1\ M_2 \end{array}$$

and so on. The manipulation made explicit by our (C) rule is implicit in Nederpelt's rule, appearing only when necessary for a beta-like contraction to occur, but Nederpelt does not address all of call-by-need reduction, and some massaging of $\hookrightarrow$ is necessary to capture reduction by (A) as well (Maraist, 1997).

On types and logic. It is straightforward to assign simple types to call-by-need terms; in addition to the usual rules for terms we have

$$\frac{\Gamma \vdash M : A \qquad\qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : B}\ \mathrm{Let}\ .$$

It is easy to verify that call-by-need reduction satisfies the subject reduction property, and it is also clear that this judgement corresponds to the Cut Theorem of the underlying natural deduction formulation of minimal intuitionistic logic.

In related work with David N. Turner (1995), we have explored the connection between the typed versions of the call-by-name, call-by-value and call-by-need calculi using *linear* systems based on the work of Girard (1987), where the use of the structural rules which allow copying and discarding of terms is restricted by a special ! operator. Girard described two translations of intuitionistic logic into an intuitionistic fragment of linear logic. The intuitionistic fragment of linear logic admits a linear lambda calculus in the same manner that the intuitionistic fragment of classical logic is related to the lambda calculus, for example the systems of Wadler (1993a; 1993b) and Barber (1995). The translations may be extended to the term level, and in fact one corresponds to call-by-name reduction, and the other to call-by-value. The former is sound and complete for mapping call-by-name reduction sequences into linear lambda sequences; the latter is sound but not complete for mapping call-by-value sequences (Maraist *et al.*, 1995). Both are sound and complete for the respective notions of standard reduction (Maraist, 1997). Mackie (1994) has shown the soundness — but not completeness — of these translations into a system based on proof nets of full (classical) linear logic for both β and η axioms.

To study call-by-need via transformation into a linear system, it is necessary to alter the (V) rule slightly: rather than substituting one use of the bound value at a time, we replace all occurrences of the bound variable, and discharge the binding:

$$(\tilde{V}) \qquad \mathsf{let}\ x = V\ \mathsf{in}\ M \to [V/x]M \ \ .$$

This reformulation allows a better fit into the logics — it is just a restricted form of cut elimination — and can also simplify a number of syntactic results about reduction.

Still, call-by-need does not fit directly into the logical framework. The fragment of call-by-need without the (G) rule, which is a conservative extension of call-by-value as discussed above, may be soundly mapped by an extension of the call-by-value translation. To include the (G) rule, we can take the target of the translations to be not linear logic, but rather affine logic, which allows arbitrary formulas to be introduced, but not used. This translation of call-by-need is sound for reduction; the affine lambda calculus also has a reasonable evaluation order under which the translation is sound and complete for standard reduction.

Jacobs' decomposition in the model theory of the ! operator into separate operators for each of the two restricted structural operations (1994) suggests another treatment of call-by-need. In the call-by-name translation, all arguments to functions are explicitly allowed to be copied or discarded; in the call-by-value translation, all values have this explicit allowance. For call-by-need it would be necessary to allow discarding of any function argument, but copying only of values. In a calculus where the corresponding syntactic operators enable the structural rules separately, this distinction is possible. Such a hybrid translation is sound and complete for both reduction and evaluation (Maraist, 1997).

On the relevance of garbage collection. One could question the inclusion of the garbage collection rule (G) in the basic system: since it is excluded from the standard reduction relation, it could be accused of irrelevance. Ariola and Felleisen believe that the rule should be optional; because nearly every implementation does include a garbage collector we feel it is important to include the rule to establish the intuitively obvious results that garbage collection does not cause evaluation to go wrong (*viz.* confluence and standardisation).

In a real sense, the (G) rule is exactly the difference between call-by-need and call-by-value. Reduction in NEED is clearly an extension of reduction in the call-by-value calculus.

Example 9
A (β_v) step

$$(\lambda x.M)\ V \to [V/x]M$$

can be expressed by the following sequence of NEED-reductions, where there is one (V) step for each occurrence of x in M:

$$\begin{array}{rll} (\lambda x.M)\ V & \xrightarrow[(I)]{} & \mathsf{let}\ x = V\ \mathsf{in}\ M \\ & \xrightarrow[(V)]{} \!\!\!\!\! \rightarrow & \mathsf{let}\ x = V\ \mathsf{in}\ [V/x]\ M \\ & \xrightarrow[(G)]{} & [V/x]\ M\ \ . \end{array}$$

If we exclude the (G) rule and use the alternate version $(\tilde{V})$ of (V) discussed above, then the extension becomes conservative (Maraist *et al.*, 1995): without the (G) rule, we are thinking more of call-by-value than call-by-name, and so the relevance of (G) to call-by-need reduction is clear.

Relevance to evaluation, on the other hand, is what one seems to miss. The fact that unneeded bindings in the closure may simply be ignored is precisely the reason why there is no rule (G_s). If we did include a garbage collection rule in $\mapsto$, we would no longer be guaranteed that only a single standard redex would be available at any point; we would also lose the simple and intuitive notion of answers as simply functions under bindings since such terms might then have a standard redex. The relevance to evaluation lies in reduction to constants, but since we do not include constants in the core functional system, we cannot yet see this role. Arguably, the inclusion of the (G) rule but exclusion of constants at this stage might seem uneven. We have chosen the present formulation based on the overall importance of the rule, while initially avoiding extensions beyond the core syntax.

On recursion. A shortcoming of our approach is its treatment of recursion. We express recursion with a fixpoint combinator (which is definable since our calculus is untyped). This agrees with Wadsworth's original treatment and most subsequent formalisations of call-by-need, with the notable exception of Launchbury's natural semantics (1993). However, implementations of lazy functional languages generally express recursion by a back-pointer in the function graph. The two schemes are equivalent for recursive function definitions but they have different sharing behaviour in the case of circular data structures. A circular pointer can allow more efficient sharing in the cases such as (say) the "infinite" list denoted by the expression

$$\mathsf{letrec}\ xs\ =\ (1+1) : xs\ \mathsf{in}\ xs\ \ .$$

Unfortunately, as Ariola and Klop (1994) have discovered, the naïve extension of a system with let's to one allowing arbitrary letrec's will not be confluent.

Ariola and Blom (1997) give a thorough treatment of recursive let-bindings in call-by-name, call-by-value and call-by-need reduction systems. Their work is based on a theory cyclic graphs constrained in a way which gives a sensible notion of the scope of bound variables, which is then related to β reduction and finally constrained to respect sharing of subterms.

Three earlier approaches to letrec's in call-by-need and similar calculi are also noteworthy: Ariola and Felleisen (1994) extend their call-by-need calculus with letrec's where selection of redexes is restricted by the use of evaluation contexts as we discussed above. This restriction does allow the extension with letrec's to be confluent, although as with their non-recursive system, it is the restrictions to the internals of the reduction axioms which makes confluence immediate. Turner, Wadler and Mossin (1995) describe a variant of the call-by-need calculus for an update analysis of Haskell programs. While their calculus does not restrict reduction contexts, it instead allows letrec's to bind only a single identifier to a value, which is a significant restriction on the recursion that can be expressed. Finally, Rose extended explicit substitutions to explicit cyclic substitutions in a $\lambda\mu$ calculus (1993). Although his formulation is simpler than Ariola and Felleisen's extension for recursion, it is not confluent†, and as his work concerns explicit substitutions rather than call-by-need, his rules do not guarantee that only values will be duplicated. A number of the rules do allow duplication of arbitrary terms, and whether one could restrict these rules to copy only values is an open question.

Acknowledgements. The authors would like to thank Zena Ariola, Matthias Felleisen, John Field, Jeremy Gibbons, Christian Mossin, Kris Rose, David N. Turner, Karen Wood and a number of anonymous referees for valuable comments and discussions.

References

Abramsky, S. (1990). *The Lazy Lambda Calculus*, Chapter 4, pages 65–116. The *UT Year of Programming* Series. Addison-Wesley Publishing Company, Inc.

Abadi, M., Cardelli, L., Curien, P.-L. and Lévy, J.-J. (1990). Explicit substitutions. In *Proc. 18th ACM Symposium on Principles of Programming Languages, San Francisco, California*. ACM Press (January).

Ariola, Z. M. and Blom, S. (1997). Cyclic lambda calculi. In *Proc. Third Int. Sym. on Theoretical Aspects of Computer Science (TACS'97)*, Sendai, Japan (September).

Ariola, Z. M. and Felleisen, M. (1994). The call-by-need lambda calculus. Technical Report CIS-TR-94-23, Department of Computer Science, University of Oregon (October).

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995). A call-by-need lambda calculus. In *Proc. 22nd Sym. on Principles of Programming Languages*, San Francisco, California. ACM Press (January).

Ariola, Z. M. and Klop, J. W. (1994). Cyclic lambda graph rewriting. In *Proc. LICS'94, Eighth IEEE Symposium on Logic in Computer Science*, Paris.

Barber, A. (1995). DILL — dual intuitionistic linear logic. Draft paper, available from the author (October).

Barendregt, H. (1981). *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company.

Benaisaissa, Z.-El-A., Lescanne, P. and Rose, K. H. (1996). Modelling sharing and recursion for weak reduction strategies using explicit substitution. In *Proc. PLILP'96, Eighth Int. Sym. on Programming Languages, Implementations, Logics and Programs*, Aix-la-Chapelle, Germany (September).

Church, A. (1941). *The Calculi of Lambda Conversion*. Princeton University Press, Princeton.

† Rose, 1997, private communication of unpublished result attributed to Stefan Blom and Zena Ariola.

Field, J. (1990). On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, San Francisco, California. ACM Press (January).

Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, 50: 1–102.

Hughes, R. J. M. (1983). *The Design and Implementation of Programming Languages.* Doctoral Thesis, Programming Research Group, Oxford University.

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

Johnsson, T. (1984). Efficient compilation of lazy evaluation. In *Proceedings of the 1984 ACM SIGPLAN Conference on Compiler Construction*, New York, ACM (June).

Josephs, M. B. (1989). The semantics of lazy functional languages. *Theoretical Computer Science*, 68: 105–111.

Koopman, P. J., Jr. and Lee, P. (1989). A fresh look at combinator graph reduction. In *Proc. SIGPLAN'89, ACM Conference on Programming Language Design and Implementation*. ACM Press (June).

Lamping, J. (1990). An algorithm for optimal lambda calculus reduction. In *Proc. 17th ACM Symposium on Principles of Programming Languages, San Francisco, California*. ACM Press (January).

Launchbury, J. (1993). A natural semantics for lazy evaluation. In *Proc. 21st ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*. ACM Press (January).

Lévy, J.-J. (1980). Optimal reductions in the lambda-calculus. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin and J. R. Harding, eds., pages 159–191 Academic Press.

Mackie, I. (1994). *The Geometry of Implementation.* Doctoral Thesis, Imperial College, London.

Maraist, J., Odersky, M., Turner, D. N. and Wadler, P. (1995). Call-by-name, call-by-value, call-by-need and the linear lambda calculus. In *Proc. MFPS'95, Eleventh Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana. Elsevier Publishers, Electronic Notes in Theoretical Computer Science 1 (March).

Maraist, J. (1997). *Comparing Reduction Strategies in Resource-Conscious Lambda Calculi*. Doctoral thesis, University of Karlsruhe.

Maraist, J., Odersky, M. and Wadler, P. (1994). The call-by-need lambda calculus (unabridged). Technical Report 28/94, Universität Karlsruhe (October).

Maranget, L. (1991). Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. In *Proc. POPL'91, 19th ACM Symposium on Principles of Programming Languages*, Orlando, Florida, pages 255–269. ACM Press (January).

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

Mossin, C., Turner, D. N. and Wadler, P. (1995). Once upon a type. In *FPCA'95: Conference on Functional Programming Languages and Computer Architecture*, La Jolla, California. ACM Press (June).

Ong, C.-H. L. (1988). Fully abstract models of the lazy lambda calculus. In *Proceedings of the 29th Symposium on Foundations of Computer Science*, pages 368–376. IEEE.

Nederpelt, R. P. (1973). *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Typed*. Doctoral thesis, Dept. of Mathematics and Computer Science, Eindhoven University of Technology.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice Hall.

Peyton Jones, S. L. (1992). Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2 (2): 127–202 (July).

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

Purushothaman, S. and Seaman, J. (1992). An adequate operational semantics of sharing in lazy evaluation. In *Proc. ESOP'92, Fourth European Symposium on Programming*, B. Krieg-Brückner, editor, *Lecture Notes in Computer Science 582*, pages 435–450, Springer-Verlag, New York (February).

Rose, K. H. (1993). Explicit cyclic substitution. Technical report D-166, Dept. of Computer Science, University of Copenhagen (DIKU) (March).

Sansom, P. M. and Peyton Jones, S. L. (1995). Time and space profiling for non-strict, higher-order functional languages. In *Proc. 22nd Sym. on Principles of Programming Languages*, San Francisco. ACM Press (January).

Turner, D. A. (1979). A new implementation technique for applicative programming languages. *Software — Practice and Experience*, 9 (31–49).

Wadler, P. (1993). A syntax for linear logic. In *Proc. MFPS'93, Ninth International Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana (April). Springer Verlag, LNCS 802.

Wadler, P. (1993). A taste of linear logic. In *Proc. Mathematical Foundations of Computer Science*, Gdansk, Poland (August). Springer Verlag, LNCS 711.

Wadsworth, C. P. (1971). *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University.

Yoshida, N. (1993). Optimal reduction in weak-lambda-calculus with shared environments. In *FPCA'93: Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark. ACM Press (June).