A Call-By-Need Lambda Calculus

Zena M. Ariola
Computer & Information Science Department
University of Oregon
Eugene, Oregon

Matthias Felleisen
Department of Computer Science
Rice University
Houston, Texas

John Maraist and *Martin Odersky*
Institut für Programmstrukturen
Universität Karlsruhe
Karlsruhe, Germany

Philip Wadler
Department of Computing Science
University of Glasgow
Glasgow, Scotland

Abstract

The mismatch between the operational semantics of the lambda calculus and the actual behavior of implementations is a major obstacle for compiler writers. They cannot explain the behavior of their evaluator in terms of source level syntax, and they cannot easily compare distinct implementations of different lazy strategies. In this paper we derive an equational characterization of call-by-need and prove it correct with respect to the original lambda calculus. The theory is a strictly smaller theory than the lambda calculus. Immediate applications of the theory concern the correctness proofs of a number of implementation strategies, *e.g.*, the call-by-need continuation passing transformation and the realization of sharing via assignments.

1 Introduction

Lazy functional programming languages implement the call-by-name lambda calculus. Their syntax provides syntactic sugar for Λ terms; their evaluators map closed terms to values. The semantics of lazy function calls is the famous β-axiom: every call is equivalent to the body of the function with the formal parameter replaced by the actual argument. The β-axiom gives rise to an equational theory, the λ-calculus, which implementors can use to reason about the outcome of programs [26].

Taken literally, the β-axiom suggests that a procedure must evaluate the argument of a specific call for each occurrence of the corresponding formal parameter. Realistic implementations of lazy functional languages avoid such computational overhead by memoizing the argument's value when it is first evaluated. More precisely, lazy languages only reduce an argument if the value of the corresponding formal parameter is needed for the evaluation of the procedure body. Moreover, after reducing the argument, the evaluator will remember the resulting value for future references to that formal parameter. This technique of evaluating procedure parameters is called *call-by-need* or *lazy*[1] evaluation.

A simple observation justifies call-by-need: the result of reducing an expression, if any, is indistinguishable from the expression itself in all possible contexts. Some implementations attempt to exploit this observation for sharing even more computations than just those of arguments. Arvind et. al. [7] provide an overview of such implementations, particularly the so-called "fully lazy" or "graph-based" techniques.

Unfortunately, the mismatch between the operational semantics of the lambda calculus and the actual behavior of the implementation is a major obstacle for compiler writers and users. Specifically, they cannot use the calculus to reason about sharing in the evaluation of a program.

Purushothaman and Seaman [27, 29] and Launchbury [18] recently recognized this problem and developed two slightly different "natural" semantics of the call-by-need parameter passing mechanism. Roughly speaking, the semantics use store-passing to describe call-by-need in terms of (the semantics of) assignment statements. Due to the low-level nature of this approach, these semantics permit neither a simple explanation of language implementations, nor source-level reasoning about program behavior. Worse, given slightly different specifications based on "natural" or similar semantic frameworks, it is difficult, if not impossible, to compare the intentions with respect to sharing in the evaluators[2].

[1] In this paper we write "call-by-need" rather than "lazy" to avoid a name clash with the work of Abramsky [2], which describes call-by-name reduction to values.

[2] Ironically, this problem immediately showed up in the dif-

A number of researchers [1, 13, 20, 31, 28] have studied reductions that preserve sharing in calculi with explicit substitutions, especially in relation to optimal reduction strategies. Having different aims, the resulting calculi are considerably more complex than those presented here. Closest to our treatment is Yoshida's weak lambda calculus [31] which introduces explicit environments similar to let constructs. Her calculus subsumes several of our reduction rules as structural equivalences, even though, due to a different notion of observation, reduction in this calculus is not equivalent to reduction to a value.

In this paper, we pursue a different approach to the specification of call-by-need[3]. Our formulation is entirely syntactic, and formulates the sharing in a call-by-need evaluator as an equational theory of the source language. The theory is a strict sub-theory of the call-by-name λ-calculus. The key technical contribution of the paper is a proof of equivalence between the call-by-name and the call-by-need evaluator. In addition, we prove that the calculus relates to the evaluator in precisely the same manner as Plotkin's call-by-name and call-by-value calculi relate to their respective evaluators.

The next two sections present the basic ideas of the call-by-name and call-by-need calculi. The fourth section presents basic syntactic results such as confluence and standardization results. The fifth and sixth sections are devoted to the correctness proof; the sixth section also describes an interesting alternative formulation of call-by-need. The last three sections briefly discuss extensions of the language (with constants and recursive declarations), the relation of our work with natural semantics-based approaches, and applications of the calculus.

2 The Call-by-Name calculus

In this section we briefly review the call-by-name calculus; we assume a basic familiarity of the reader with this material [8].

Figure 1 details the call-by-name calculus. The set of lambda terms, called Λ, is generated over an infinite set of variables. The expression $M[x := N]$ denotes the capture-free substitution of N for each free occurrence of x in M. The *reduction theory* associated with the calculus is the result of taking the compatible, reflexive and transitive closure of β interpreted as an asymmetric relation:

$$\boldsymbol{\beta} = \{(\lambda x.M)N, M[x := N] \mid M, M \in \Lambda\}.$$

The compatible closure of $\boldsymbol{\beta}$ is written as $\xrightarrow[name]{}$; the reflexive and transitive closure of $\xrightarrow[name]{}$, as $\xrightarrow[name]{}\!\!\!\!\rightarrow$; $=_{name}$ is the symmetric closure of $\xrightarrow[name]{}\!\!\!\!\rightarrow$, or the congruence relation generated by $\xrightarrow[name]{}$. We will omit the tag $_{name}$ when we may do so unambiguously.

In an implementation of the calculus, closed expressions play the role of *programs*. An execution of a program aims at returning an observable value in realistic languages. Observable values are basic constants like numbers and booleans. If a program's result is a procedure or a lazy tree, most implementations only indicate whether or not the program has terminated and possibly what kind of higher-order result it returned. Since the pure theory only contains λ-abstractions, an evaluator only determines whether a program terminates.

Given this preliminary idea of how implementations work, we can use the calculus to define a partial evaluation function $eval_{name}$ from *programs*, or closed terms, to the singleton consisting of the tag *closure*:

$$eval_{name}(M) = \mathit{closure} \text{ iff } \boldsymbol{\lambda} \vdash M = \lambda x.N \ .$$

That is, the evaluation of a program returns the tag *closure* if, and only if, the theory can prove the program is equal to a value. It is a seminal result due to Plotkin that the evaluation function of a typical implementation is also determined by the standard reduction relation [26]. Put differently, a correct implementation of the evaluator can simply reduce the standard or leftmost-outermost redex of a program until the program becomes a value.

Evaluation contexts are a convenient way of formulating the evaluation relation [11]. A program M *standard reduces* to N, written as $M \underset{name}{\longmapsto} N$, iff $M \equiv E_n[(\lambda x.P)Q]$ and $N \equiv E_n[P[x := Q]]$. As usual, $M \underset{name}{\longmapsto\!\!\!\!\rightarrow} N$ means M standard reduces to N via the transitive-reflexive closure of $\underset{name}{\longmapsto}$; $M \underset{name}{\longmapsto}^{0/1} N$ means M standard reduces to N in zero or one step; $M \underset{name}{\longmapsto\!\!\!\!\rightarrow}^{n} N$ means M standard reduces to N in n steps. As before, we will omit the tag $_{name}$ when no confusion arises. The following characterization of the call-by-name evaluator is a consequence of the confluence property and the standardization theorem of $\boldsymbol{\lambda}$.

Proposition 2.1 *For a program M,*

$$eval_{name}(M) = \mathit{closure} \textit{ iff } M \longmapsto\!\!\!\!\rightarrow \lambda x.N \ .$$

This result is due to Plotkin [26].

ferences between the formulations of Purushotoman/Seaman and Launchbury.

[3] In fact, this approach unifies the similar, simultaneous and independent work of two separate groups. While many of our results are indeed quite similar, there are interesting and significant differences in overall perspectives, in specific definitions of calculi and in proof techniques. We do not describe these differences here, and instead refer the interested reader to the full technical reports of Ariola and Felleisen [4] and of Maraist, Odersky and Wadler [19] for details.

Syntactic Domains

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

Axioms

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

Figure 1: The call-by-name lambda calculus.

Syntactic Domains

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

Axioms

$$\begin{array}{lll} (\mathsf{let\text{-}I}) & (\lambda x.M)\,N & = \mathsf{let}\ x = N\ \mathsf{in}\ M \\ (\mathsf{let\text{-}V}) & \mathsf{let}\ x = V\ \mathsf{in}\ C[x] & = \mathsf{let}\ x = V\ \mathsf{in}\ C[V] \\ (\mathsf{let\text{-}C}) & (\mathsf{let}\ x = L\ \mathsf{in}\ M)\,N & = \mathsf{let}\ x = L\ \mathsf{in}\ M\,N \\ (\mathsf{let\text{-}A}) & \mathsf{let}\ y = (\mathsf{let}\ x = L\ \mathsf{in}\ M)\ \mathsf{in}\ N & = \mathsf{let}\ x = L\ \mathsf{in}\ \mathsf{let}\ y = M\ \mathsf{in}\ N \end{array}$$

Figure 2: The calculus λ_{let}.

3 From Call-by-Name to Call-by-Need

We augment the term syntax of the classical λ-calculus with a let-construct. The underlying idea is to represent a reference to an actual parameter in an instantiated function graph by a let-bound identifier[4]. Hence, sharing in a function graph will correspond to naming in a term. Figure 2 details the calculus λ_{let} over these augmented terms. As in the call-by-name calculus, we extract the related reduction theory, and will frequently make reference to its compatible closure $\xrightarrow[need]{}$, the reflexive, transitive closure $\twoheadrightarrow_{need}$ thereof, and finally the congruence $=_{need}$ which it generates, omitting tags when the meaning is clear. We will also refer to reduction theories of individual axioms in the same manner, *e.g.*, $M \xrightarrow[\text{let-I}]{} N$, and use (for example) the abbreviation $M \xrightarrow[\text{let-\{V,C,A\}}]{} N$ to mean that M reduces to N by let-V and by either let-V, let-A or let-C.

As Wadsworth pointed out, one way to avoid duplicating reductions is by replacing the abstracted variables in function bodies with *references* to arguments, rather than with the arguments themselves [30]. The let-I axiom models the creation of such a let binding by representing the reference with a let-bound name, *e.g.*,

$$(\lambda x.xx)(II) \;=\; \begin{array}{l}\mathsf{let}\ x = II \\ \mathsf{in}\ xx\end{array} \ ,$$

where $I \equiv \lambda z.z$.

The intention of this sharing of graphs is to prevent the duplication of work by reducing the same expression more than once. However, once a shared expression is a value — in this core calculus, values are simply abstractions — there is no harm in the duplication. In fact, the duplication then becomes necessary; reduction requires an actual abstraction, not just a pointer to one. This dereferencing is expressed by the let-V rule, which substitutes the value bound to a name for an occurrence of that name.

Unfortunately, such a modeling of the sharing of values in the source level syntax means that looking only for simple values is no longer adequate. A value may itself contain shared terms, so the simple abstraction may be "buried" within one or more let bindings. Consider the expression

$$(\lambda f.fI(fI))\,((\lambda z.\lambda w.zw)\,II) \ ,$$

which let-I equates with

$$\begin{array}{l}\mathsf{let}\ f = \mathsf{let}\ z = II \\ \qquad\quad\ \mathsf{in}\ \lambda w.zw \\ \mathsf{in}\ fI(fI) \ .\end{array}$$

Further reduction of this term clearly requires that an abstraction be substituted for the first f in $fI(fI)$. Were

[4] The idea of keeping pairs of procedures and their arguments in the syntax of the language is due to work on explicit substitutions [1], on graph rewriting [3, 5], and on adding state to the λ-calculus [10, 12, 21, 22]. Only the latter two though exploited the idea of modeling the sharing relation of the program heap inside of the source language.

we to obtain this abstraction by naïvely applying some relaxed form of let-V to the term as a whole, say

$$\begin{aligned}&\mathsf{let}\ x = (\mathsf{let}\ y = M\ \mathsf{in}\ V)\ \mathsf{in}\ C[x]\\ &\quad = \ \mathsf{let}\ x = (\mathsf{let}\ y = M\ \mathsf{in}\ V)\\ &\qquad\ \mathsf{in}\ C[\mathsf{let}\ y = M\ \mathsf{in}\ V]\ ,\end{aligned}$$

we would lose sharing:

$$\begin{aligned}&\mathsf{let}\ f = (\mathsf{let}\ z = II\ \mathsf{in}\ \lambda w.zw)\\ &\ \mathsf{in}\ fI(fI)\\ &\quad = \ \mathsf{let}\ f = (\mathsf{let}\ z = II\ \mathsf{in}\ \lambda w.zw)\\ &\qquad\ \mathsf{in}\ (\mathsf{let}\ z = II\ \mathsf{in}\ \lambda w.zw)\ I(fI)\ .\end{aligned} \tag{1}$$

The redex II and the work involved in reducing it have been duplicated.

The solution is to re-associate the bindings. Rather than accepting Eq. 1, re-association will allow the current let-V axiom to apply without loss of sharing:

$$\begin{aligned}&\mathsf{let}\ f = (\mathsf{let}\ z = II\ \mathsf{in}\ \lambda w.zw)\\ &\ \mathsf{in}\ fI(fI)\\ &\quad = \ \mathsf{let}\ z = II\\ &\qquad\ \mathsf{in}\ \mathsf{let}\ f = \lambda w.zw\\ &\qquad\quad\ \mathsf{in}\ fI(fI)\\ &\quad = \ \mathsf{let}\ z = II\\ &\qquad\ \mathsf{in}\ \mathsf{let}\ f = \lambda w.zw\\ &\qquad\quad\ \mathsf{in}\ (\lambda w.zw)I(fI)\end{aligned} \tag{2}$$

The rearrangement of Eq. 2 is captured by the let-A axiom. In the example, even though f does occur twice, II will be contracted only once[5].

Equation 1 points out a second situation where let-V is inadequate in the presence of values under bindings. In the expression[6]

$$(\mathsf{let}\ z = II\ \mathsf{in}\ \lambda w.zw)\ I\ ,$$

we would like to associate the abstraction $\lambda w.zw$ with the argument I, but the two are not directly adjacent as required by let-I. Again, we must rearrange the term, expecting an equality

$$\begin{aligned}&(\mathsf{let}\ z = II\ \mathsf{in}\ \lambda w.zw)\ I\\ &\quad = \ \mathsf{let}\ z = II\ \mathsf{in}\ (\lambda w.zw)\ I\ ;\end{aligned}$$

[5] But note that we avoid duplication only of argument evaluations. In the program

$$(\lambda f.fI(fI))(\lambda w.(II)w)\ ,$$

the redex II in the argument will be reduced twice. Put differently, our calculus captures neither full laziness as described by Wadsworth [30], nor the sharing required by optimal λ-calculus interpreters [13, 15, 16, 17, 20]. However, as observed by Arvind, Kathail and Pingali [7], full laziness can always be obtained by extracting the maximal free expressions of a function at compile-time [25].

[6] Although we rejected the liberalized let-V rule which produced this particular expression, it is perfectly reasonable to expect similar terms to appear. In fact, *any* left-nesting of two applications $(LM)N$ could produce such a term!

this manipulation is generalized by the let-C axiom.

The let-A and let-C rules may be viewed as allowing the scope of a bound identifier to be expanded over broader expressions. To avoid copying unreduced expressions, such an expansion is required when the argument of a function contains a binding (the let-A rule) and when a expression which is itself a binding is applied to some argument (the let-C rule).

It is precisely these two rules let-A and let-C which allow us to do without a separate store. Rather than creating references to new, unique global addresses, we simply extend the scope of identifiers as needed.

Example 3.1 $(\lambda x.x\,x)\,(\lambda y.y)$.

$$\begin{aligned}&(\lambda x.x\,x)\,(\lambda y.y)\\ &\xrightarrow[\text{let-I}]{}\ \begin{aligned}&\mathsf{let}\ x = \lambda y.y\\ &\ \mathsf{in}\ x\,x\end{aligned}\ \xrightarrow[\text{let-V}]{}\ \begin{aligned}&\mathsf{let}\ x = \lambda y.y\\ &\ \mathsf{in}\ (\lambda z.z)\ x\end{aligned}\\ &\xrightarrow[\text{let-I}]{}\ \begin{aligned}&\mathsf{let}\ x = \lambda y.y\\ &\ \mathsf{in}\ \mathsf{let}\ z = x\\ &\quad\ \mathsf{in}\ z\end{aligned}\ \xrightarrow[\text{let-V}]{}\ \begin{aligned}&\mathsf{let}\ x = \lambda y.y\\ &\ \mathsf{in}\ \mathsf{let}\ z = \lambda w.w\\ &\quad\ \mathsf{in}\ z\end{aligned}\\ &\xrightarrow[\text{let-V}]{}\ \begin{aligned}&\mathsf{let}\ x = \lambda y.y\\ &\ \mathsf{in}\ \mathsf{let}\ z = \lambda w.w\\ &\quad\ \mathsf{in}\ \lambda v.v\end{aligned}\end{aligned}$$

In the call-by-name calculus, the terms we identified as observable results, or answers, were simply abstractions. However, in this formulation of call-by-need the notion of answer must reflect the possibility that such an abstraction can be under bindings. The appropriate definition given in Figure 2 means that answers are a syntactic representation of closures.

4 Basic Syntactic Properties

Following Plotkin's technique, we define the call-by-need evaluator as a partial relation $eval_{need}$ from *programs*, or closed terms, to the singleton consisting of the tag *closure* based on equality in our call-by-need calculus:

$$eval_{need}(M) = \mathit{closure}\ \text{iff}\ \boldsymbol{\lambda}_{\text{let}} \vdash M = A\ .$$

Furthermore, from the following result, a program evaluates to *closure* if and only if it reduces to an answer in the call-by-need calculus.

Theorem 4.1 λ_{let} *is confluent:*

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

Proof: The system consisting of just let-I is trivially confluent; then by marked and weighted redexes as in Barendregt [8] the remaining reductions let-V, let-C and let-A are both weakly Church-Rosser and strongly normalizing, and thus Church-Rosser as well. Since both subsystems commute, the theorem follows from the Lemma of Hindley and Rosen [8, Proposition 3.3.5]. □

This result does not imply that all reduction sequences lead to an answer in λ_{let}. We therefore introduce the notion of *standard reduction*, which always reaches an answer if there is one. Figure 3 details our notion of standard reduction. An expression is only reduced when it appears in the hole of an evaluation context. The first two productions for evaluation contexts are those of the call-by-name calculus. Since arguments to procedures are evaluated only when needed, the standard reduction system postpones the evaluation of the argument and instead proceeds with the evaluation of the procedure body. Once the formal parameter of the procedure appears in the hole of an evaluation context, and only then, the value of the argument is needed to complete the evaluation. In this case, the standard reduction system evaluates the argument before substituting it into the body of the procedure.

Given a program $M \in \Lambda_{\text{let}}$, M *standard reduces* to N, written as $M \underset{need}{\longmapsto} N$, iff $M \equiv E[K]$ and $N \equiv E[L]$, where K and L are a standard redex and its contractum, respectively. $M \underset{need}{\longmapsto\!\!\!\!\rightarrow} N$ means M and N are related via the transitive-reflexive closure of $\underset{need}{\longmapsto}$. As usual, we will omit the tags from the arrows when the meaning is clear from the context. Verifying that the *standard* relation, $\longmapsto$, is indeed a function from programs to programs relies on the usual Unique Evaluation Context Lemma [11]. This lemma states that there is a unique partitioning of a non-answer into an evaluation context and a redex, which implies that there is precisely one way to make progress in the evaluation.

Lemma 4.2 *Given a program $M \in \Lambda_{\text{let}}$, either M is an answer, or there exists a unique evaluation context E and a standard redex N such that $M \equiv E[N]$.*

Proof: By structural induction on M. □

Theorem 4.3 *Given a program $M \in \Lambda_{\text{let}}$,*

$$eval_{need}(M) = \mathit{closure} \text{ iff } \exists A, M \longmapsto\!\!\!\!\rightarrow A \ .$$

Proof: The proof relies on two subsidiary results: that all answers are in $\underset{need}{\longmapsto\!\!\!\!\rightarrow}$-normal form, and that a sequence of non-standard reductions followed by a sequence of standard reductions can be transformed into an equivalent sequence of first standard reductions, then non-standard reductions. The proof is somewhat reminiscent of Barendregt's standardization proof for the call-by-name calculus, but without the convenience of a finiteness of developments theorem for all redex types [8, Lemma 11.4.3]. □

5 Completeness of the Call-by-Need calculus

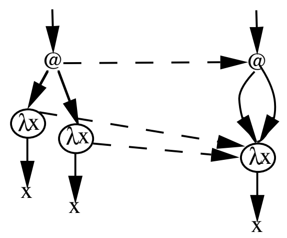
Replacing the call-by-name interpreter with the call-by-need interpreter requires an equivalence proof for the two evaluators:

$$eval_{need} = eval_{name} \ .$$

In this section, we will show the completeness direction, that is, each result obtained with the call-by-name interpreter is also produced by the call-by-need interpreter. Formally, we have for any program M,

$$eval_{name}(M) = \mathit{closure} \Longrightarrow eval_{need}(M) = \mathit{closure} \ .$$

We introduce an ordering between terms of Λ and Λ_{let}. Intuitively, $M \leq N$ if M can be obtained by unwinding N, that is, N contains more sharing than M [3]. In other words, the ordering relation $\leq$ expresses whether the terms have homomorphic graphs. For example, the tree of $M \equiv (\lambda x.x)(\lambda x.x)$ can be homomorphically embedded into the graph of $N \equiv \text{let } y = \lambda x.x$ in yy,



which shows that $M \leq N$. Notation: given a term $M \in \Lambda_{\text{let}}$, let $\mathcal{D}ag(M)$ be the corresponding dag, and given a term $N \in \Lambda$, let $\mathcal{T}ree(N)$ be the corresponding tree.

Definition 5.1 For $M \in \Lambda$ and $N \in \Lambda_{\text{let}}$, $M \leq N$ iff there exists an homomorphism $\sigma : \mathcal{T}ree(M) \rightarrow \mathcal{D}ag(N)$.

If $M \leq N$, then each call-by-name evaluation of M can be simulated in the call-by-need calculus by evaluating N. The term obtained in the call-by-need calculus is not necessarily greater (in terms of the above ordering) than the one obtained following the call-by-name evaluation. This is because a one-step call-by-need reduction may correspond to multiple call-by-name reductions. For example, the evaluation

$$\begin{aligned} M &\equiv (\lambda x.xx)((\lambda y.y)(\lambda z.z)) \\ \underset{name}{\longmapsto\!\!\!\!\rightarrow} M_1 &\equiv (\lambda z.z)((\lambda y.y)(\lambda z.z)) \end{aligned}$$

Syntactic Domains:

Values:	V	$::=$	$\lambda x.M$
Answers:	A	$::=$	$V \mid \mathsf{let}\ x = M\ \mathsf{in}\ A$
Evaluation Contexts:	E	$::=$	$[\,] \mid EM \mid \mathsf{let}\ x = M\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E[x]$

Standard Reduction rules:

(let_s-I)	$(\lambda x.M)N$	$\longrightarrow$	$\mathsf{let}\ x = N\ \mathsf{in}\ M$
(let_s-V)	$\mathsf{let}\ x = V\ \mathsf{in}\ E[x]$	$\longrightarrow$	$\mathsf{let}\ x = V\ \mathsf{in}\ E[V]$
(let_s-C)	$(\mathsf{let}\ x = M\ \mathsf{in}\ A)N$	$\longrightarrow$	$\mathsf{let}\ x = M\ \mathsf{in}\ AN$
(let_s-A)	$\mathsf{let}\ x = (\mathsf{let}\ y = M\ \mathsf{in}\ A)\ \mathsf{in}\ E[x]$	$\longrightarrow$	$\mathsf{let}\ y = M\ \mathsf{in}\ \mathsf{let}\ x = A\ \mathsf{in}\ E[x]$

Figure 3: Standard call-by-need reduction.

corresponds to the following call-by-need evaluation:

$$\begin{aligned} M &\equiv (\lambda x.xx)((\lambda y.y)(\lambda z.z)) \\ &\xmapsto[need]{} \mathsf{let}\ x = (\lambda y.y)(\lambda z.z)\ \mathsf{in}\ xx \\ &\xmapsto[need]{} \mathsf{let}\ x = (\mathsf{let}\ y = \lambda z.z\ \mathsf{in}\ y)\ \mathsf{in}\ xx \equiv N \ . \end{aligned}$$

Obviously $M_1 \not\leq N$. However, there exists an M_2 such that $M_1 \xrightarrow[name]{} M_2$ and $M_2 \leq N$:

$$M_1 \xrightarrow[name]{} M_2 \equiv (\lambda z.z)(\lambda z.z) \leq N \ .$$

Since $M \leq M$, the last point shows how to reconstruct a call-by-name evaluation in the call-by-need calculus. Hence, the completeness direction follows.

If $M \leq N$ and M is a β-redex, N is not necessarily a let_s-I redex. Suppose

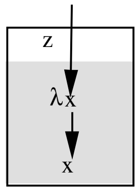
$$\begin{aligned} M &\equiv (\lambda x.x)(\lambda x.x) \\ N &\equiv \mathsf{let}\ z = (\mathsf{let}\ w = \lambda x.x\ \mathsf{in}\ w)\ \mathsf{in}\ zz \end{aligned}$$

then $M \leq N$, yet N does not contain a let_s-I redex. The example points out that our graph model of Λ_{let} terms must be able to express the let-structure of a term in addition to its sharing structure if we want to use it for a correctness proof.

We solve this problem by enriching dags with boxes and labeled edges [6]. $\mathcal{D}ag_{\Box}(M)$ is the decorated dag associated with an expression M. A box can be thought of as a refined version of a node; the label associated with an edge is just a sequence of let-bound variable names. The label can be thought of as a direction to be followed in order to get to a particular node. Each let induces one box, and each edge to the shared term is decorated with the variable name. We pictorially represent a term $\mathsf{let}\ x = N\ \mathsf{in}\ M$ by a box divided in two parts: the upper part corresponds to M (the unshaded area of a box) and the lower part contains N (the shaded area of a box). Let us illustrate the extended dag notation and terminology with a number of examples.

Example 5.2

(i) The term $\mathsf{let}\ z = \lambda x.x\ \mathsf{in}\ z$ is drawn as

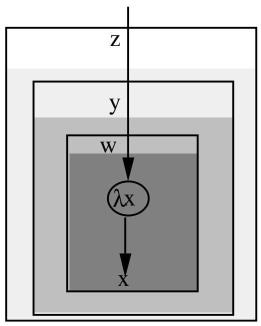


The name z associated with the root pointer is drawn outside the shaded area.

(ii) We can also have nested boxes, *e.g.*, the term

$$\begin{aligned} &\mathsf{let}\ z = \mathsf{let}\ y = \mathsf{let}\ w = \lambda x.x \\ &\qquad\qquad\qquad\qquad \mathsf{in}\ w \\ &\qquad\qquad \mathsf{in}\ y \\ &\mathsf{in}\ z \end{aligned}$$

is drawn as

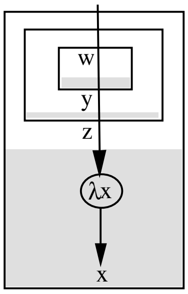


where the path to the λ-node must follow the label zyw and hit three walls.

In contrast, in the term

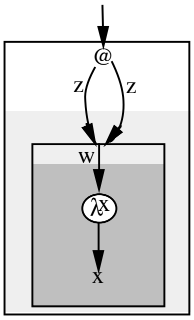
$$\begin{aligned} &\mathsf{let}\ z = \lambda x.x \\ &\mathsf{in}\ \mathsf{let}\ y = z \\ &\quad \mathsf{in}\ \mathsf{let}\ w = y \\ &\qquad \mathsf{in}\ w \ , \end{aligned}$$

drawn as



the path to the λ-node must follow the label wyz, and it penetrates three walls but leaves two encasings.

In our running example, $\mathcal{D}ag_{\Box}(N)$ is :



In this decorated dag, the path from the application (root) node to the λ-node has label zw, and it penetrates the wall of one box. To expose the redex means that the function pointer (of the root node) must point to the λ-node directly. In terms of our graphical language, we must eliminate the names z, w, the internal box, and pull the λ-node out of the shaded area, exactly the task of the let_s-A, let_s-V, and let_s-C rules. Their dag-based representation in Table 1 reveals that let_s-V pulls a value out of the shaded area, eliminating a name on an edge; let_s-C moves a wall; and let_s-A moves a wall that is in the shaded area. The sequence let_s-V, let_s-A, let_s-V suffices to expose the redex in our example:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{let}\ z = (\mathsf{let}\ w = \lambda x.x\ \mathsf{in}\ w)\ \mathsf{in}\ zz} \\ \xmapsto[\mathsf{let}_s\text{-V}]{} & \mathsf{let}\ z = (\mathsf{let}\ w = \lambda x.x\ \mathsf{in}\ \lambda x.x)\ \mathsf{in}\ zz \\ \xmapsto[\mathsf{let}_s\text{-A}]{} & \mathsf{let}\ w = \lambda x.x\ \mathsf{in}\ (\mathsf{let}\ z = \lambda x.x\ \mathsf{in}\ zz) \\ \xmapsto[\mathsf{let}_s\text{-V}]{} & \mathsf{let}\ w = \lambda x.x\ \mathsf{in}\ (\mathsf{let}\ z = \lambda x.x\ \mathsf{in}\ (\lambda x.x)z)\ . \end{array}$$

Figure 4 (without unreachable dags) illustrates these steps.

From Table 1 it is clear that let_s-C and let_s-A do not change the dag associated with a term, while let_s-V causes a duplication.

Lemma 5.3
(i) Given $M \in \Lambda_{\mathrm{let}}$, *if* $M \xrightarrow[\mathrm{let}_s\text{-}\{\mathrm{C,A}\}]{} \!\!\!\!\!\!\twoheadrightarrow N$ *then* $\mathcal{D}ag(M) = \mathcal{D}ag(N)$.
(ii) Given $M \in \Lambda_{\mathrm{let}}$, *if* $M \xrightarrow[\mathrm{let}_s\text{-V}]{} \!\!\!\!\!\!\twoheadrightarrow N$ *then* $N \le M$.

The language and notation for decorated dags is useful in proving the following three key lemmas. All could be formulated in plain term-based language, but at the cost of introducing more technical details.

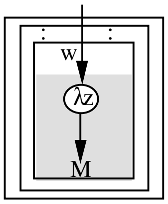
Lemma 5.4 *Given* $M \in \Lambda, N \in \Lambda_{\mathrm{let}}$, *if* $M \le N$ *and* $M \equiv E_n[(\lambda x.P)R]$ *then there exists* P', R', *and* $E[\ ]$, *such that* $N \xmapsto[\mathrm{let}_s\text{-}\{\mathrm{V,C,A}\}]{} \!\!\!\!\!\!\twoheadrightarrow E[(\lambda x.P')R']$.

Proof: Let z be the root in $\mathcal{T}ree(M)$ of the β-redex being evaluated. Let z' and $z'_{\Box}$ be the corresponding nodes in $\mathcal{D}ag(N)$ and $\mathcal{D}ag_{\Box}(N)$, respectively. We know that the left branch of z' points to a λ-node, while in $\mathcal{D}ag_{\Box}(N)$ the path from $z'_{\Box}$ to the λ-node may contain some obstacles. Thus, we show that by using let_s-A, let_s-V, and let_s-C we can remove all the obstacles from that path. We reason by induction on the number n of names associated with the path in $\mathcal{D}ag_{\Box}(N)$ from $z'_{\Box}$ to the λ-node.

$n = 0$. This means that the path from $z'_{\Box}$ to the λ-node is free of names, but it still may penetrate intervening walls. With m walls, we need m let_s-C steps to move the walls and expose the redex.

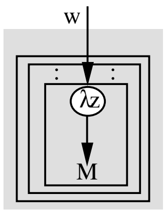
$n > 0$. By the induction hypothesis, we can remove n-1 names. Now we need to show how to eliminate the last one. There are two cases:

1. The name associated with the λ-node is w:



   Since w occurs in head position, an application of let_s-V exposes the λ-node;

2. The branch labeled w points to m boxes that surround the λ-node, *e.g.*



   Since w occurs in head position, m applications of let_s-A followed by a single application of let_s-V expose the λ-node.

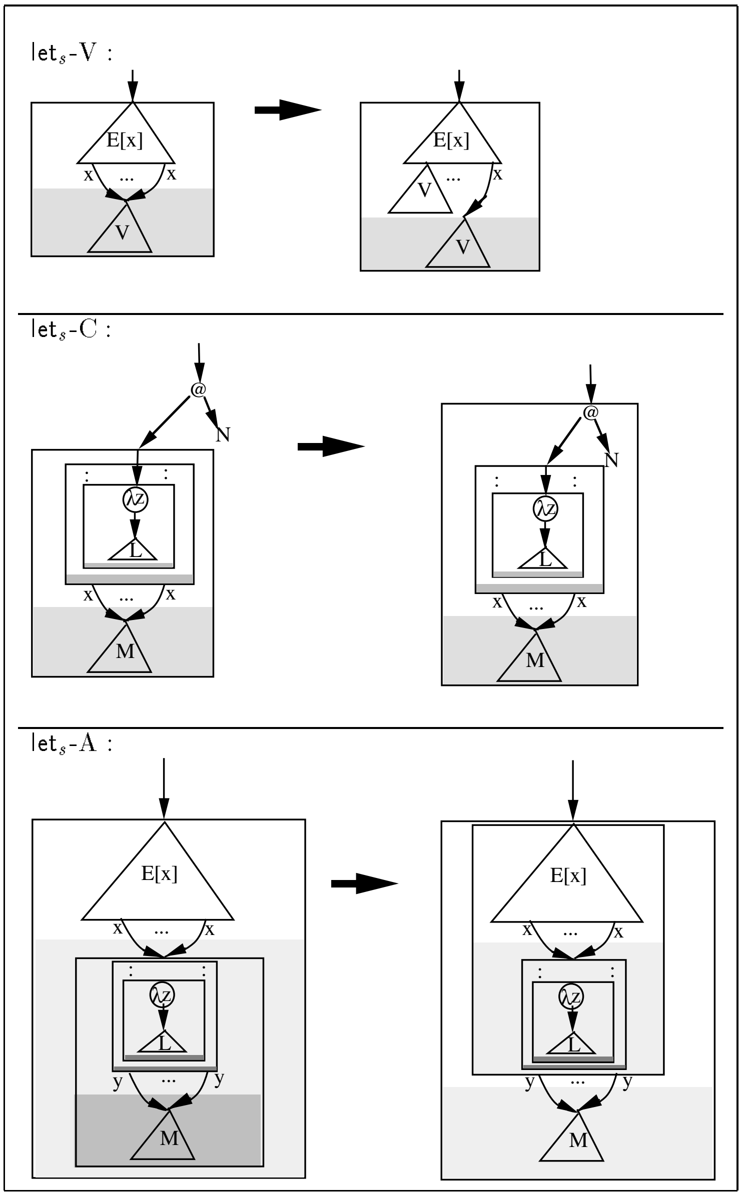


Table 1: Standard call-by-need reduction rules in dag-based form.

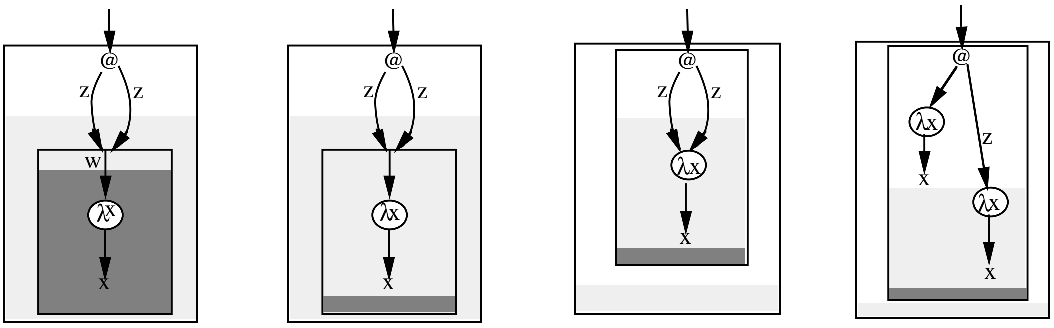


Figure 4: Exposing the redex in let $z = (\mathsf{let}\ w = \lambda x.x\ \mathsf{in}\ w)$ in zz.

Let $N', N \underset{\text{let}_s\text{-}\{\text{V,C,A}\}}{\longmapsto\!\!\!\twoheadrightarrow} N'$; under the assumption that $N' \equiv C[(\lambda x.P')R']$ for $C[\,]$ not an evaluation context (that is, the redex $(\lambda x.P')R'$ is not needed) and hence z not the root of the leftmost-outermost redex in M, we have a contradiction.

□

Lemma 5.5 *Given $M \in \Lambda, N \in \Lambda_{\text{let}}$, if $M \leq N$ and $M \equiv \lambda x.P$ then there exists an answer A such that $N \underset{\text{let}_s\text{-}\{\text{V,A}\}}{\longmapsto\!\!\!\twoheadrightarrow} A$.*

Proof: The proof is similar to but simpler than the previous one. In fact, we need not move the walls surrounding the lambda-node; that is, no use of let_s-C is required. □

Lemma 5.6 *Given a program $M \in \Lambda$ and $N \in \Lambda_{\text{let}}$, if $M \leq N$ and $M \underset{\mathit{name}}{\longmapsto\!\!\!\twoheadrightarrow}^n M_1$, then $\exists M_1' \in \Lambda, N_1 \in \Lambda_{\text{let}}$ such that*

$$M_1 \underset{\mathit{name}}{\twoheadrightarrow} M_1',\ N \underset{\mathit{need}}{\longmapsto\!\!\!\twoheadrightarrow}^{\leq n} N_1 \text{ and } M_1' \leq N_1.$$

Pictorially:

$$\begin{tikzcd}
M \arrow[r, "\mapsto^n"] & M_1 \arrow[r, dotted, "\longrightarrow"] & \exists M_1' \\
N \arrow[u, "\leq"] \arrow[rr, dotted, "\mapsto^{\leq n}", "\mathit{need}"'] & & \exists N_1 \arrow[u, dotted, "\leq"']
\end{tikzcd}$$

Proof: By induction on the length n of the reduction $M \underset{\mathit{name}}{\longmapsto\!\!\!\twoheadrightarrow}^n M_1$.

$n = 1$. Let $M \equiv E_n[(\lambda x.P)R]$, and let z be the root in $\mathcal{T}ree(M)$ of the β-redex in the hole of $E_n[\,]$. From Lemma 5.4, there exists P', R' and $E[\,]$:

$$N \underset{\mathit{need}}{\longmapsto\!\!\!\twoheadrightarrow} N' \text{ and } N' \equiv E[(\lambda x.P')R'] \ .$$

Let z' be the root in $\mathcal{D}ag(N')$ of the let_s-I-redex. From Lemma 5.3 and the fact that M does not contain any sharing we have $M \leq N'$. Thus:

$$M \underset{\mathit{name}}{\longmapsto} M_1 \equiv E_n[P[x := R]]$$

and

$$N' \underset{\text{let}_s\text{-I}}{\longrightarrow} N_1 \equiv E[\mathsf{let}\ x = R' \ \mathsf{in}\ P'] \ .$$

If there exists a node z_1 in $\mathcal{T}ree(M)$, where $z_1 \neq z$, such that $\sigma(z_1) = z'$, where σ is the homomorphism associated with the ordering $M \leq N'$, then $M_1 \not\leq N_1$. Let $\mathcal{F}$ be the set of all such nodes. Let $M_1', M_1 \underset{\mathit{name}}{\twoheadrightarrow} M_1'$, by reducing all redexes in $\mathcal{F}$ and their residuals. We have: $M_1' \leq N_1$.

$n > 1$. Let $M \underset{\mathit{name}}{\longmapsto\!\!\!\twoheadrightarrow}^{n-1} M'$ and $M' \underset{\mathit{name}}{\longmapsto} M_1$. By the induction hypothesis, $\exists N_1 \in \Lambda_{\text{let}}, M'' \in \Lambda$,

$$N \underset{\mathit{need}}{\longmapsto\!\!\!\twoheadrightarrow}^{\leq(n-1)} N_1, M' \underset{\mathit{name}}{\twoheadrightarrow} M'' \text{ and } M'' \leq N_1 \ .$$

From the Strip Lemma [8] $\exists M_2$,

$$M'' \underset{\mathit{name}}{\longmapsto}^{0/1} M_2 \text{ and } M_1 \underset{\mathit{name}}{\twoheadrightarrow} M_2 \ .$$

If $M_2 \equiv M''$ then we have:

$$N \underset{\mathit{need}}{\longmapsto\!\!\!\twoheadrightarrow}^{\leq(n-1)} N_1 \text{ and } M' \underset{\mathit{name}}{\twoheadrightarrow} M''$$

where $M'' \leq N_1$. Otherwise, by the induction hypothesis $\exists N_1' \in \Lambda_{\text{let}}, M_2' \in \Lambda$,

$$N_1 \underset{\mathit{need}}{\longmapsto}^{0/1} N_1', M_2 \underset{\mathit{name}}{\twoheadrightarrow} M_2' \text{ and } M_2' \leq N_1' \ .$$

□

Lemma 5.7 *Given a program $M \in \Lambda$,*

$$M \underset{\mathit{name}}{\longmapsto\!\!\!\twoheadrightarrow}^n \lambda x.N \Longrightarrow M \underset{\mathit{need}}{\longmapsto\!\!\!\twoheadrightarrow} A \ .$$

Proof: Since $M \leq M$, from Lemma 5.6, $\exists M_1 \in \Lambda_{\text{let}}, N_1 \in \Lambda$ such that:

$$M \underset{\mathit{need}}{\longmapsto\!\!\!\twoheadrightarrow}^{\leq n} M_1 \text{ and } \lambda x.N \underset{\mathit{name}}{\twoheadrightarrow} \lambda x.N_1$$

where $\lambda x.N_1 \leq M_1$. The result then follows from Lemma 5.5. □

With these lemmas we can prove the main result of this section, namely, that call-by-need can simulate a call-by-name evaluation.

Theorem 5.8 *If $M \in \Lambda$ and $\mathit{eval}_{\mathit{name}}(M) = \mathit{closure}$, then $\mathit{eval}_{\mathit{need}}(M) = \mathit{closure}$.*

Proof: The assumption implies $M \underset{\mathit{name}}{\longmapsto\!\!\!\twoheadrightarrow} \lambda x.N$. Hence, the result follows from Lemma 5.7. □

6 A Let-Less Formulation of Call-by-Need

In the λ_{let} calculus, we have treated the expression $\mathsf{let}\ x = M\ \mathsf{in}\ N$ as a term distinct from $(\lambda x.N)\,M$. An alternate treatment is also quite reasonable: that the former is merely syntactic sugar for the latter. In other words, it is possible to completely eliminate let's from the call-by-need calculus and still have a system with the same desired properties. By expanding let-bindings into applications, we can derive the λ_ℓ calculus shown in Figure 5 from λ_{let}. There is of course no analogue of the let-I rule in λ_ℓ, since we must no longer convert away from plain applications. We call the evaluator for this language $\mathit{eval}^{\ell}_{\mathit{need}}$ to distinguish it from the evaluator for λ_{let}.

While λ_ℓ is perhaps somewhat less intuitive than λ_{let}, its simpler syntax can make some of the basic

Syntactic Domains

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

Reduction Axioms

$$\begin{array}{lllll}
(\ell\text{-V}) & (\lambda x.C[x])\,V & = & (\lambda x.C[V])\,V \\
(\ell\text{-C}) & (\lambda x.L)MN & = & (\lambda x.LN)M \\
(\ell\text{-A}) & (\lambda x.L)((\lambda y.M)N) & = & (\lambda y.(\lambda x.L)M)N
\end{array}$$

Figure 5: Let-less call-by-need.

(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_{let} and λ_ℓ are closely related. More precisely, the following theorem states that reduction in λ_{let} can be simulated in λ_ℓ, and that the converse is also true, provided we identify terms that are equal up to let-I introduction.

Proposition 6.1 *For all $M \in \Lambda_\ell$, $M' \in \Lambda_{\mathrm{let}}$,*

$$\begin{tikzcd}
M \arrow[r, twoheadrightarrow, "\lambda_\ell"'] \arrow[d, "\mathsf{let}\text{-}I"] & N \arrow[d, dotted, "\mathsf{let}\text{-}I"] \\
M' \arrow[r, dotted, twoheadrightarrow, "\lambda_{\mathrm{let}}"'] & N'
\end{tikzcd}
\qquad
\begin{tikzcd}
M \arrow[r, dotted, twoheadrightarrow, "\lambda_\ell"'] \arrow[d, "\mathsf{let}\text{-}I"] & N \arrow[d, dotted, "\mathsf{let}\text{-}I"] \\
M' \arrow[r, twoheadrightarrow, "\lambda_{\mathrm{let}}"'] & N'
\end{tikzcd} .$$

Proposition 6.1 can be used to derive the essential syntactic properties of λ_ℓ from those of λ_{let}; in particular the confluence result for λ_ℓ follows from Theorem 4.1 by the proposition.

λ_ℓ has close relations to both the call-by-value calculus λ_V and the call-by-name calculus λ. Its notion of equality $=_{\lambda_\ell}$ — *i.e.*, the least equivalence relation generated by the reduction relation — fits between those of the other two calculi, making λ_ℓ an extension of λ_V and λ an extension of λ_ℓ.

Theorem 6.2 $=_{\lambda_V} \subset =_{\lambda_\ell} \subset =_\lambda$.

Proof: (1) βV can be expressed by a sequence of λ_ℓ reductions as was shown at the beginning of this section. Therefore, $=_{\lambda_V} \subset =_{\lambda_\ell}$. (2) Each λ_ℓ reduction rule is an equality in λ. For instance, in the case of ℓ-V one has:

$$\begin{array}{rcl}
(\lambda x.C[x])\,V & =_\beta & [V/x](C[x]) \\
 & \equiv & [V/x](C[V]) \\
 & =_\beta & (\lambda x.C[V])\,V
\end{array}$$

The other rules have equally simple translations, and so we have $=_{\lambda_\ell} \subset =_\lambda$.
□

Each of the inclusions of Theorem 6.2 is proper, *e.g.*,

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

where Ω stands for a non-terminating computation, is an instance of rule ℓ-A, but it is not an equality in the call-by-value calculus (Ω stands for a non-terminating computation). On the other hand, the following instance of β is not an equality in λ_ℓ:

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

However, one can show that the observational equivalence theories of λ_ℓ and λ are identical (and are incompatible with the observational equivalence theory of λ_V):

Theorem 6.3 *For all programs $M \in \Lambda$,*

$$eval_{name}(M) = eval^{\ell}_{need}(M) \ .$$

Proof: Follows from Theorem 5.8 and 6.2.
□

Theorem 6.2 implies that any model of call-by-name λ-calculus is also a model of λ_ℓ, since it validates all equalities in λ_ℓ. Theorem 6.3 implies that any adequate (respectively, fully-abstract) model of λ is also adequate (fully-abstract) for λ_ℓ, since the observational equivalence theories of both calculi are the same[7]:

Corollary 6.4 *For all terms $M, N \in \Lambda$,*

$$M \cong_{name} N \text{ iff } M \cong_{need} N \ .$$

7 Extensions

Most lazy functional languages extend the pure calculus in several ways. In this section we consider two such extensions, for constructors and for recursion.

[7] For instance, Abramsky and Ong's model of the lazy lambda calculus [2] is adequate for λ_ℓ.

7.1 Constructors and Primitive Operators

Figure 6 extends λ_{let} with data constructors k^n of arbitrary arity n and primitive operators p (of which selectors are a special case). There is one new form of value: $k^n\ V_1\ ...\ V_n$ where the components V_1 through V_n must be values — otherwise sharing would be lost when copying the compound value [14]. For instance, $inl\,(1+1)$ is not a legal value, since copying it would also copy the unevaluated term $(1+1)$. Instead, one writes

$$\mathsf{let}\ x = 1+1\ \mathsf{in}\ inl\ x\ .$$

There are two new reduction rules. Rule δ-V is the usual rewrite rule for primitive operator application. It is defined in terms of a partial function — also called δ — from operators and values to terms. This function can be arbitrary, as long as it does not "look inside" lambda abstractions. That is, we postulate that for all operators p and contexts C there is a context D such that for all terms M, $\delta(p, C[\lambda x.M]) = D[\lambda x.M]$ or $\delta(p, C[\lambda x.M])$ is undefined. Note that rule δ-V makes all primitive operators unary and strict. Operators of more than one argument can still be simulated by currying. Rule δ-A allows let-bindings of operator arguments to be pulled out of applications of primitive operators.

Alternatively, one could phrase these constructs in terms of constructors and case statements in reduction rules.

7.2 Recursion

A deficiency of our treatment of call-by-need is its treatment of recursive or cyclic values. Traditionally one relies on the Y combinator for recursion. In the absence of data constructors, this solution is fine. However, once data constructors are included, the sharing in the source language no longer reflects the sharing in the evaluator. For example, the term

$$M \equiv \mathsf{Y}(\lambda y.\mathsf{cons}(1, y))$$

evaluates to a term containing *two* distinct cons cells even though an actual implementation would allocate only one cell, representing M as a cyclic structure.

To cope with recursion, we extend the call-by-need calculus with a *letrec* construct, where no ordering among the bindings is assumed. This extended calculus is given in Figure 7. Unlike the calculus λ_{let} of Section 3, we now have a restricted notion of substitution. In other words, substitutions only occur when a variable appears in the hole of an evaluation context. Otherwise, an unrestricted notion of substitution in the presence of cycles would cause interesting non-confluence phenomena [5].

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

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

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

Figure 8: Operational semantics.

This extended call-by-need calculus corresponds to Ariola and Klop's call-by-name calculus with cycles [5], in the same way that our call-by-need calculus corresponds to the call-by-name calculus. The correctness proof of the calculus with recursion can be obtained by showing its soundness and completeness with respect to a calculus of infinitary graphs.

8 Relation to 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 [18]. We state a proposition that relates the natural semantics to standard reduction.

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.[8] Furthermore, all variables bound by the heap must be distinct. Thus the heap above is well-formed if $\mathrm{fv}(M_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ 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 $\mathrm{vars}(\Phi) = \{x_1, \ldots, x_n\}$. 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 $\mathrm{fv}(M) \subseteq \mathrm{vars}(\Phi)$. The operation of evaluation takes configurations into configurations. The term of the final configuration is always a value. Thus evaluation judgments take the form $\langle\Phi\rangle\, M \Downarrow \langle\Psi\rangle\, V$.

The rules defining evaluation are given in Figure 8. There are three rules, for identifiers, abstractions and applications.

[8] So this model of the heap is incompatible with the extension for recursion given in Section 7.2; see the end of this discussion.

Syntactic Domains

Operators	p			
Constructors	k^n		(of arity n)	
Values	V, W	$::=$	$x \mid \lambda x.M \mid k^n\ V_1 \ldots V_n$	$(n \geq 0)$
Terms	L, M, N	$::=$	$V \mid M\ N \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid p$	

Additional Axioms

$$\begin{array}{llcll} (\delta\text{-V}) & p\,V & = & \delta(p, V) & (\delta(f, V)\text{ defined}) \\ (\delta\text{-A}) & p\,(\mathsf{let}\ x = M\ \mathsf{in}\ N) & = & \mathsf{let}\ x = M\ \mathsf{in}\ p\,N & \end{array}$$

Figure 6: Data constructors and primitive operations.

Syntactic Domains

Values	V	$::=$	$x \ \mid\ \lambda x.M$
Terms	M, N	$::=$	$x \ \mid\ V \ \mid\ M\ N \ \mid\ \langle M \mid x_1 = N_1, \cdots, x_n = N_n \rangle$
Evaluation contexts	E	$::=$	$[\,] \ \mid\ E\ M \ \mid\ \langle E \mid D \rangle \ \mid\ \langle E[x] \mid D[x, x_n], x_n = E, D \rangle$
	$D[x, x_n]$	$::=$	$x = E[x_1], x_1 = E[x_2], \cdots, x_{n-1} = E[x_n]$

Axioms

$$\begin{array}{llcl} (\beta_{need}) & (\lambda x.M)N & = & \langle M \mid x = N \rangle \\ (lift) & (\langle V \mid D \rangle)N & = & \langle V N \mid D \rangle \\ (deref) & \langle E[x] \mid x = V, D \rangle & = & \langle E[V] \mid x = V, D \rangle \\ (deref_i) & \langle E[x] \mid D[x, x_n], x_n = V, D \rangle & = & \langle E[x] \mid D[x, V], x_n = V, D \rangle \\ (assoc) & \langle \langle V \mid D_1 \rangle \mid D_2 \rangle & = & \langle V \mid D_1, D_2 \rangle \\ (assoc_i) & \langle E[x] \mid D[x, x_n], x_n = \langle V \mid D \rangle, D_1 \rangle & = & \langle E[x] \mid D[x, x_n], x_n = V, D, D_1 \rangle \end{array}$$

Figure 7: Recursion.

- Abstractions are trivial. As abstractions are already values, the heap is left unchanged and the abstraction is returned.
- Applications are straightforward. 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, 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 are more subtle. The basic idea is straightforward: 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. But some care is required to ensure that the heap remains well-formed. The original heap is partitioned into Φ, $x \mapsto M$, Υ. 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: binding 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 Ψ, $x \mapsto V$, Υ.

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

Lemma 8.1 *Given an evaluation tree with root configuration $\langle \Phi \rangle\, M \Downarrow \langle \Psi \rangle\, V$, if $\langle \Phi \rangle\, M$ is well-formed then every configuration in the tree is well-formed, and furthermore $vars(\Phi) \subseteq vars(\Psi)$.*

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 8.2 $\langle \Phi \rangle\, M \Downarrow \langle \Psi \rangle\, V$ *iff*

$$\lambda_\ell \vdash \mathsf{let}\,\Phi\,\mathsf{in}\,M \xrightarrow[need]{} \mathsf{let}\,\Psi\,\mathsf{in}\,V \ .$$

The semantics given here is similar to that presented by Launchbury [18]. 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.

One advantage of Launchbury's semantics over ours is that his copes more neatly with recursion, by the use of multiple, recursive let bindings. In particular, our heap structure is incompatible with the extension for recursion of Section 7.2. This extension would alter both the ordering property and the connection to standard reduction.

9 Applications

Call-by-need calculi have a number of potential applications. Their primary purpose is as a reasoning tool for the implementation of lazy languages. We sketch three ideas.

Call-by-need and assignment

Call-by-need can be implemented using assignments. Crank [9, 10] briefly discusses a rewriting semantics of call-by-need based on Felleisen and Hieb's λ-calculus with assignments [12]. We believe that a call-by-need calculus is the correct basis for proving this implementation technique correct with a simple simulation theorem for the respective standard reductions.

Call-by-need and cps conversion

Okasaki et. al. [24] recently suggested a continuation-passing transformation for call-by-need languages. In principle, this transformation should satisfy the same theorems as the continuation-passing transformation for call-by-name and call-by-value calculi [26]. Plotkin's proof techniques should immediately apply. Since this transformation appears to be used in several implementations of lazy languages, it is important to explore its properties with standard tools.

Garbage collection

Modeling the sharing relationship of an evaluator's memory in the source syntax suggests that the calculus can also model garbage collection. Indeed, garbage collection can be easily expressed in our call-by-need calculus by adapting the garbage collection rule for reference cells of Felleisen and Hieb [10, 12]:

$$\mathsf{let}\ x = M\ \mathsf{in}\ N = N \qquad \text{if } x \notin FV(N)$$

We expect that the work on garbage collection in functional languages by Morrisett et. al. [23] will apply to call-by-need languages. Such a rigorous treatment of garbage collection would strengthen the calculus and its utility for reasoning about the implementations of lazy languages.

10 Conclusion

The calculus we have presented here has several nice properties which make it suitable as a reasoning tool for lazy functional programs. With operations on the lambda-terms themselves (or perhaps a mildly sugared version) rather than on a separate store of bindings, and with a small set of straightforward rules, we feel that our approach is clearer and simpler than previous approaches. The unsugared calculus fits naturally between the call-by-name and call-by-value versions of λ.

Acknowledgements. The authors would like to thank H. Barendregt, J. Field, J.W. Klop and D. N. Turner for numerous helpful discussions.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 4(1), 1991.

[2] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Declarative Programming*. Addison-Wesley, 1990.

[3] Z. M. Ariola and Arvind. Properties of a first-order functional language with sharing. *Theoretical Computer Science*, September 1995.

[4] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. Technical Report CIS-TR-94-23, Department of computer science, University of Oregon, October 1994.

[5] Z. M. Ariola and J. W. Klop. Cyclic lambda graph rewriting. In *Proc. of the Eighth IEEE Symposium on Logic in Computer Science, Paris*, 1994.

[6] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Acta Informatica*, 1994.

[7] Arvind, V. Kathail, and K. Pingali. Sharing of computation in functional language implementations. In *Proc. International Workshop on High-Level Computer Architecture*, 1984.

[8] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, 1984.

[9] E. Crank. Parameter-passing and the lambda calculus. Master's thesis, Rice University, 1990.

[10] E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *Proc. ACM Conference on Principles of Programming Languages*, 1990.

[11] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, Ebberup, Denmark, August 1986.

[12] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102, 1992.

[13] J. Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proc. ACM Conference on Principles of Programming Languages, San Francisco*, 1990.

[14] D. P. Friedman and D. S. Wise. *Cons* should not evaluate its arguments. In *Proc. International Conference on Automata, Languages and Programming*, 1976.

[15] G. Gonthier, M. Abadi, and J.-J Lévy. The geometry of optimal lambda reduction. In *Proc. ACM Conference on Principles of Programming Languages*, 1992.

[16] V. K. Kathail. *Optimal Interpreters for Lambda-calculus Based Funtional Languages.* PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, 1990.

[17] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. ACM Conference on Principles of Programming Languages, San Francisco*, January 1990.

[18] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. ACM Conference on Principles of Programming Languages*, 1993.

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

[20] L. Maranget. Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. In *Proc. ACM Conference on Principles of Programming Languages, Orlando, Florida*, January 1991.

[21] I. A. Mason and C. Talcott. Reasoning about programs with effects. In *Proc. of Programming Language Implementation and Logic Programming, Springer-Verlag LNCS 456, Berlin*, 1990.

[22] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(2), 1991.

[23] G. Morrisett, M. Felleisen, and R. Harper. Modeling memory management. Technical report, Departement of computer science, Carnegie Mellon University, forthcoming 1994.

[24] C. Okasaki, P. Lee, and T. Tarditi. Call-by-need and continuation-passing style. In *Lisp and Symbolic Computation*, 1994.

[25] S. L. Peyton Jones. A fully-lazy lambda lifter in haskell. *Software Practice and Experience*, 21, 1991.

[26] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1, 1975.

[27] Purushothaman and J. Seaman. An adequate operational semantics of sharing in lazy evaluation. In *Proc. 4th European Symposium on Programming, Springer Verlag LNCS 582*, 1992.

[28] K. H. Rose. Explicit cyclic substitutions. In *3rd International Workshop on Conditional Term Rewriting Systems*, July 1992.

[29] J. M. Seaman. *An Operational Semantics of Lazy Evaluation for Analysis.* PhD thesis, The Pennsylvania State University, 1993.

[30] C. Wadsworth. *Semantics And Pragmatics Of The Lambda-Calculus.* PhD thesis, University of Oxford, September 1971.

[31] N. Yoshida. Optimal reduction in weak-λ-calculus with shared environments. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Copenhagen*, 1993.