The 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

Abstract

Plotkin [*Theor. Comp. Sci. 1975*] showed that the lambda calculus is a good model of the evaluation process for call-by-name functional programs. Reducing programs to constants or lambda abstractions according to the leftmost-outermost strategy exactly mirrors execution on an abstract machine like Landin's SECD machine. The machine-based evaluator returns a constant or the token `closure` if and only if the standard reduction sequence starting at the same program will end in the same constant or in some lambda abstraction. However, the calculus does not capture the sharing of the evaluation of arguments that *lazy* implementations use to speed up the execution. More precisely, a lazy implementation evaluates procedure arguments only when needed and then only once. All other references to the formal procedure parameter re-use the value of the first argument evaluation.

The mismatch between the operational semantics of the lambda calculus and the actual behavior of the prototypical implementation is a major obstacle for compiler writers. Unlike implementors of the leftmost-outermost strategy or of a call-by-value language, implementors of lazy systems cannot easily explain the behavior of their evaluator in terms of source level syntax. Hence, they often cannot explain why a certain syntactic transformation "works" and why another doesn't. In this paper we develop an equational characterization of the most popular lazy implementation technique—traditionally called "call-by-need"—and prove it correct with respect to the original lambda calculus. The theory is a strictly smaller theory than Plotkin's call-by-name 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.

Note: Some of this material first appeared in a paper presented at the 1995 ACM Conference on the Principles of Programming Languages. The paper was a joint effort with Maraist, Odersky and Wadler, who had independently developed a different equational characterization of call-by-need. We contrast our work with that of Maraist et al. in the body of this paper where appropriate.

[†] Supported by NSF grant CCR-94-10237
[‡] Supported by NSF grant CCR 91-22518

1 Introduction

In his seminal paper on call-by-name, call-by-value, and the lambda calculus Plotkin (1975) formulates two criteria concerning the relationship between a calculus—an equational theory on some syntax—and a programming language—a semantics, defined via an abstract machine or an interpreter for the same syntax as the calculus. First, the calculus should satisfy a Curry-Feys-style Standardization Theorem, and a standard reduction from a program to a value should mimic the evaluation of the program according to the language's semantics. Second, the equations of the calculus should identify terms that are "observationally indistinguishable" from each other. Plotkin illustrates the connection with two examples. The first is the call-by-value lambda calculus and the functional core of Landin's ISWIM defined via the SECD machine. The second is the lambda calculus and a variant of ISWIM that models a pragmatic call-by-name programming language. The semantics of the language is based on a variant of the SECD machine. The corresponding reduction strategy differs from the evaluation strategy proposed by lambda calculus logicians. Instead of reducing a program—a closed term—to normal form, Plotkin's evaluator reduces a program to a lambda abstraction.†

Unfortunately, the correspondence between non-strict functional languages and the lambda calculus is not satisfactory. Whereas the calculus accurately models the relation between a program and its final value, it does not accurately capture how a program gets to its value. Take for example the term $(\lambda x.x + x + x)(1 + 2)$. An implementation using "call-by-need" parameter-passing evaluates 1+2 once and remembers the result for all references to the variable x. In contrast, a standard reduction according to the λ-calculus, which models call-by-name evaluation, reduces the term to $(1 + 2) + (1 + 2) + (1 + 2)$, and thus $1 + 2$ is reduced three times.

We believe that if a calculus is to help implementors in defining and explaining their work, it must capture important *intensional* aspects of evaluation in addition to the *extensional* semantics. While the extensional semantics of a lazy implementation and a call-by-name implementation are identical, the calculus only captures the intensionality of a call-by-name machine that re-evaluates arguments for each occurrence of the parameter, and not that of a call-by-need implementation. In this paper we introduce a *calculus* that captures the amount of sharing in call-by-need implementations of call-by-name languages. We prove that the calculus defines the same interpreter as Plotkin's call-by-name calculus and that it is a strictly smaller theory than the lambda calculus. In addition, the call-by-need calculus satisfies a Curry-Feys-style Standardization Theorem, which we can take as the starting point for the derivation and verification of lower-level implementations. In short, the calculus is a good calculus for a call-by-need language in Plotkin's sense.

Unlike many others, we are not interested in capturing the optimality of reduction strategies (Maranget, 1991; Yoshida, 1993; Field, 1990; Kathail, 1990; Gonthier

† Abramsky and Ong (1990; 1988) explored the model theoretic properties of this calculus. They called it the "lazy lambda calculus", even though this calculus does not at all address the laziness of implementations.

et al., 1992; Lamping, 1990). We do not deny the possible benefits of optimal reductions but instead focus on modeling the sharing used in current implementations. Our work also differs from that of Purushothaman and Seaman (1992) and Launchbury (1993), who defined an operational semantics that characterizes call-by-need evaluation using so-called "natural" semantics. The basic idea behind their two formulations is the same. Roughly speaking, the semantics use store-passing to describe call-by-need in terms of (the semantics of) assignment statements. An evaluation judgement relates a program and the current store to the result and a modified store. Since this technique is of low-level nature, neither approach permits a simple explanation of the implementation or source-level reasoning about the behavior of programs and the correctness of syntactic transformations. Reasoning about equality between arbitrary terms is nearly impossible with these semantics. Given slightly different specifications based on "natural" or similar semantic frameworks, it is difficult, if not impossible, to compare the intensional aspects of the evaluation strategy, especially the amount of sharing. In contrast, our own work shows that call-by-need evaluation is based on an equational theory that is strictly smaller than the call-by-name theory and that Wadsworth's fully lazy strategy is yet another, closely related equational theory.

We emphasize source syntax over graphs in our work for two reasons. First, we believe that source syntax is a natural tool for explaining the semantics of a language because it is one aspect that every programmer must master. Second, whereas a graph-based model for a first-order language is simple, a graph model for a higher-order language requires many auxiliary notions. To formulate a sufficiently strong set of transformations we would have to enrich graphs in a number of ways. For example, we would need the concept of a name associated to a node in order to express the lifting of expressions out of a procedure. Once we have enriched graphs sufficiently, they are actually isomorphic to terms with explicit sharing. Given the isomorphism, it is best to choose the model that fits the purpose. In this paper, we use the enriched graph to work out the completeness of our calculus, but we use the syntactic model in all other contexts.

The paper is organized as follows. Section 2 presents the basic ideas of the call-by-name calculus. Section 3 introduces the essential of call-by-need evaluation: *the sharing of argument's evaluation*. Call-by-need evaluation is formulated in Section 4 as an equational theory on the set of λ-calculus terms. In addition, Section 4 contains the proof 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 call-by-need theory is shown to be a strict sub-theory of the call-by-name λ-calculus in Section 5, which also contains the proof of equivalence between the call-by-name and the call-by-need evaluator. We briefly develop the idea of a fully lazy calculus in Section 6 and compare the modified calculus with the original one. In Section 7 we show that our calculus allows also reasoning about non-strict languages based on a lenient strategy. We sketch in Section 8 how to extend the work to additional constructs of pragmatic functional languages. The last section briefly addresses applications of the call-by-need calculus.

2 The Call-by-Name calculus

In this section we briefly review the basic concepts of the call-by-name λ-calculus, including the notions of descendant, development and complete development. We refer the reader to Barendregt's treatise (1984) for a more detailed exposition.

The set of lambda terms, called Λ, is generated by the grammar

$$M ::= x \mid \lambda x.M \mid MM$$

with x ranging over an infinite set of variables. Herein, we work with α-equivalence classes of terms. The basic axiom of the theory $\boldsymbol{\lambda}$ is the β-axiom:

$$(\lambda x.M)N = M[x := N] \ .$$

The expression $M[x := N]$ denotes the capture-free substitution of N for each free occurrence of x in M. The *reduction theory* associated to the calculus is the result of taking the compatible, reflexive and transitive closure of β interpreted as an asymmetric relation:

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

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 also make use of the notation $\xrightarrow{U}$ to indicate that U is the redex being contracted. 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 terminates with a *value*. If the value is *observable*, *e.g.*, a number or a character, the evaluator will print the value; if it is *higher-order*, *e.g.*, a lazy tree or a procedure, it only indicates that the execution ended and what kind of value was obtained. Since the pure theory only contains procedures (lambda abstractions) as values, 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 $\mathtt{Eval}_{\mathsf{name}}$ from *programs*, or closed terms, to the singleton consisting of the tag `closure`:

$$\mathtt{Eval}_{\mathsf{name}}(M) = \mathsf{closure} \text{ if and only if } \boldsymbol{\lambda} \vdash M = \lambda x.N \ .$$

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 (1975) that the evaluation function of a typical implementation is also determined by the standard reduction relation. 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.

A convenient way of formulating the evaluation relation based on the standard reduction strategy utilizes contexts for identifying the standard redex (Felleisen & Friedman, 1986), where a context is a term with a hole in it (written as [ ]). The evaluation context is such that the redex filling the hole is the standard redex.

The set of *call-by-name evaluation contexts* is the following subset of the λ-calculus contexts:

$$E_n ::= [\,] \mid E_n M \quad .$$

The above definition states that either a term is a redex or the redex (recursively) occurs in the function part of an application. No reductions under λ-abstraction occur. A program M *standard reduces* to N, written as $M \xmapsto[name]{} N$, if and only if $M \equiv E_n[(\lambda x.P)Q]$ and $N \equiv E_n[P[x := Q]]$. As usual, $M \xmapsto[name]{} N$ means M standard reduces to N via the transitive-reflexive closure of $\xmapsto[name]{}$; $M \xmapsto[name]{}^{0/1} N$ means M standard reduces to N in zero or one step; $M \xmapsto[name]{}^{n} N$ 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 (Plotkin)
For a program M, $\mathtt{Eval}_{\mathtt{name}}(M) = \mathsf{closure}$ if and only if $M \xmapsto[name]{} \lambda x.N$.

In the following sections, we will need two important lemmas about the meta-operations of the conventional lambda calculus. The first concerns substitution.

Lemma 2.2
If $x \not\equiv y$ and $x \notin \mathtt{FV}(M)$, $L[x := N][y := M] = L[y := M][x := N[y := M]]$.

The lemma implies that if y does not occur free in L:

$$L[x := N][y := M] = L[x := N[y := M]],$$

which, in turn, validates the equation

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

The next lemma shows how the evaluation relation and ordinary reductions commute. Before presenting the lemma we introduce the concept of *descendant* of a redex. Following Klop (1992) we define the concept using the technique of underlining. Let U and U_1 be two distinct redexes occurring in M, and let $M \xrightarrow{U_1} M_1$. We wish to know what happened in this step to the redex U. To that end, we underline the head symbol of U (*i.e.*, its leftmost lambda symbol), the set of underlined redexes in M_1 are the descendants of U with respect to the U_1-reduction. In the reduction:

$$M \equiv (\lambda x.xx)((\underline{\lambda} z.z)(\lambda w.w)) \rightarrow M_1 \equiv ((\underline{\lambda} z.z)(\lambda w.w))((\underline{\lambda} z.z)(\lambda w.w))$$

the descendants of the redex $((\lambda z.z)(\lambda w.w))$ are the underlined redexes in M_1.

Lemma 2.3
If $M \twoheadrightarrow N_1$ and $M \longmapsto N$ then $\exists N_2, N_1 \longmapsto^{0/1} N_2$ and $N \twoheadrightarrow N_2$.

If $\mathcal{F}$ is a set of redexes occurring in a term M, a *development* of M with respect to $\mathcal{F}$ is a series of reductions that only reduce elements of $\mathcal{F}$ and their descendants. For

example, the following reduction is a development with respect to the underlined redexes in M,

$$\begin{aligned} M \equiv (\underline{\lambda}x.xx)((\underline{\lambda}z.z)(\lambda w.w)) &\rightarrow ((\underline{\lambda}z.z)(\lambda w.w))((\underline{\lambda}z.z)(\lambda w.w)) \\ &\rightarrow (\lambda w.w)((\underline{\lambda}z.z)(\lambda w.w)) \quad . \end{aligned}$$

While the reduction

$$\begin{aligned} M \equiv (\underline{\lambda}x.xx)((\underline{\lambda}z.z)(\lambda w.w)) &\longrightarrow ((\underline{\lambda}z.z)(\lambda w.w))((\underline{\lambda}z.z)(\lambda w.w)) \\ &\longrightarrow \underbrace{(\lambda w.w)((\underline{\lambda}z.z)(\lambda w.w))}_{U} \xrightarrow{U} (\underline{\lambda}z.z)(\lambda w.w), \end{aligned}$$

is not a development, because redex U is not a descendant of an underlined redex in M, that is, U is not an underlined redex. A development is called *complete* if all redexes in $\mathcal{F}$ and their descendants are reduced. That is, if all redexes in $\mathcal{F}$ are underlined, then a development $M \twoheadrightarrow N$ is a complete development if no underlining occurs in N. The reduction

$$\begin{aligned} M \equiv (\underline{\lambda}x.xx)((\underline{\lambda}z.z)(\lambda w.w)) &\longrightarrow ((\underline{\lambda}z.z)(\lambda w.w))((\underline{\lambda}z.z)(\lambda w.w)) \\ &\longrightarrow (\lambda w.w)((\underline{\lambda}z.z)(\lambda w.w)) \longrightarrow (\lambda w.w)(\lambda w.w) \quad , \end{aligned}$$

is a complete development with respect to the underlined redexes in M. We always refer to a development and a complete development with respect to a set of redexes.

3 Reasoning about Call-by-Need

The basic idea behind *call-by-need* is to start the evaluation of a procedure's body as soon as the procedure shows up in function position, to defer the evaluation of the argument until the evaluation of the procedure's body depends on the value of the argument, and to re-use the value of the argument for all other references to the parameter during the rest of the procedure body's evaluation. The implementation of this parameter-passing technique via interpreters or abstract machines is typically based on environments (Friedman & Wise, 1976; Henderson & Morris, 1976) or *graph reduction* (Peyton Jones, 1987; Turner, 1979; Wadsworth, 1971). An environment-based machine associates a procedure's formal parameter with the actual argument in the environment; an interpreter based on graph-reduction replaces occurrences of formal parameters with pointers to the argument expression.

These implementation techniques for call-by-need suggest that an equational characterization of call-by-need cannot rely on full-fledged substitution. Following the work on explicit substitutions (Abadi *et al.*, 1991; Lescanne, 1994; Bloo & Rose, 1995), and on adding state to the λ-calculus (Crank & Felleisen, 1990; Felleisen & Hieb, 1992), we do not reduce $(\lambda x.M)N$ but interpret it as a syntactic representation of the program M and an environment that associates x with N. If and when the evaluation of M requires the value of N, N is reduced to a value and occurrences of x are gradually replaced with N's value upon demand. This demand-driven substitution of values for variables captures the spirit of both environment-based and graph-based machines. In the former, it corresponds to a

lookup in the environment; in the latter, it corresponds to the copying of the body of a lambda-expression in function position. That is, the substitution operation does not arise because of our syntactic formulation, but truly expresses what happens in the implementation.

Formalizing our ideas must start with a replacement of the β-axiom for the λ-calculus. One first attempt could be

$$(\lambda x.M)V = M[x := V] \ . \tag{1}$$

The equation clarifies that formal parameters are never replaced with full arguments but values only. However, it fails to capture any of other aspects of the informal characterization. Specifically, the equation does not show that call-by-need replaces variables by values gradually and only when the evaluation process demands it.

A straightforward way to improve equation (1) is to use contexts. With a context that contains a single hole, we can indicate that a term consists of a specific variable and some other components. For example, if x is a variable and $C[\,]$ is a context (that does not trap x), then

$$(\lambda x.C[x])$$

is a way to indicate a specific occurrence of the procedure parameter in the procedure's body. By using this notation we can refine equation (1) to

$$(\lambda x.C[x])V = (\lambda x.C[V])V \qquad \text{for any one-hole context } C[\,],$$

which restricts the replacement of variables by values to individual occurrences of parameters. Unfortunately, this equation still does not explicate what "need" means in the "call-by-need" terminology. It permits the replacement of any occurrence of a procedure's parameter by a value, even if a replacement of this particular occurrence is *not* needed for the evaluation. For example, the replacement of x by 3 in the derivation

$$(\lambda x.\mathbf{if}(\mathbf{true}, 0, x))3 = (\lambda x.\mathbf{if}(\mathbf{true}, 0, 3))3$$

is permissible according to our second attempt, but it is clearly superfluous to determine the final value 0.[‡]

In order to solve this last problem, we adopt Felleisen and Hieb's constraint on variable dereferences. Their constraint enforces that an argument is only copied into a procedure body when the value of the procedure's parameter is needed to advance the body's evaluation. The best syntactic way for expressing this idea is to restrict the set of permissible contexts in the axiom that replaces β. Felleisen and Hieb chose call-by-value evaluation contexts because they worked with an extension of Plotkin's call-by-value theory. We must use call-by-need evaluation contexts instead because it is our goal to develop an equational characterization of call-by-need evaluation.

[‡] Adopting this equation as the replacement of β also leads to inconsistencies in extensions of the equational theory with cyclic data definitions or imperative facilities (Felleisen & Friedman, 1989; Ariola & Klop, 1994). We consider this problem a further indictment of the unrestricted context in the proposed axiom.

We thus arrive at the following axiom for substituting procedure parameters with the values of their arguments:

$$\mathit{deref}: \qquad (\lambda x.E[x])V = (\lambda x.E[V])V \quad , \qquad \text{for any evaluation context } E[\,] \ .$$

Like our first attempt at a β-replacement, the axiom only permits the replacements of variables with values. Like our second attempt, it forces a gradual, one-by-one replacement of parameters by argument values. And, most importantly, parameters are only replaced when evaluation demands it.

We have now boiled down the formulation of the replacement of the β axiom to one critical element: the set of call-by-need evaluation contexts.[§] Like any evaluation context, a call-by-need evaluation context must formalize the notion of a leftmost-outermost position. Put differently, a redex in the hole of an evaluation context must be the leftmost-outermost redex of the term.

A rigorous description of the set of call-by-need evaluation contexts demands a close look at the call-by-need evaluation strategy. Clearly, if M is a program and it is a redex, we certainly want to determine its value, and if M is an application, we need the value of the rator. Thus, the set of evaluation contexts subsumes at least the following:

$$E ::= [\,] \mid EM \quad ,$$

which is actually the set of call-by-name evaluation contexts.

Next we need to exploit that we evaluate a procedure's body without eliminating the surrounding application. Translated into syntax this means that the evaluator reduces redexes in a procedure's body if the procedure is in the function position of an application, i.e., the set of evaluation contexts also includes

$$(\lambda x.E)M \quad .$$

Unlike the previous contexts, this new kind is *not* a call-by-name evaluation context. Since call-by-name evaluation can be modeled by substituting arguments for parameters, the call-by-name theory has no need for evaluation inside of a procedure body.

Finally we need to add evaluation contexts that show when arguments are evaluated. Call-by-need demands the evaluation of an argument precisely when the sequential evaluator touches a procedure parameter for a first time. Put syntactically, if a procedure's parameter is the leftmost-outermost term of the procedure body, the focus of the call-by-need evaluator shifts to the procedure's actual argument. To express that a procedure parameter occurs in leftmost-outermost position, we can write

$$(\lambda x.E[x]) \quad .$$

This term expresses that the procedure's body can be partitioned into an evaluation

[§] Call-by-need evaluation contexts define a static notion of *need* that is quite different from that of Huet and Lévy (Huet & Lévy, 1991), which in contrast, captures the idea that a redex is needed if it is reduced in *every* reduction from the program to normal form.

context and an occurrence of the bound variable, which means that an advancement of the evaluation now depends on the value of the variable. If we can now evaluate the term that is associated with the bound variable, we shift the focus of the evaluation just in time, that is, we need evaluation contexts of the form

$$(\lambda x.E[x])E \ .$$

Since we have now considered all possible cases in an evaluator, we have a complete definition of a call-by-need evaluation context:¶

$$E ::= [\,] \mid EM \mid (\lambda x.E)M \mid (\lambda x.E[x])E.$$

We will prove its completeness when we show that the call-by-need calculus defines an evaluator that is equivalent to a call-by-name evaluator.

Now that we have a replacement for β, we can check whether the axiom suffices to reduce programs to values when possible. However, this very statement immediately points out a problem. Programs are no longer reduced to values when we use $deref$ instead of β. For example, using $(deref)$, the term

$$(\lambda x.x)(\lambda z.z)$$

reduces to

$$(\lambda x.(\lambda z.z))(\lambda z.z) \ ,$$

which *represents* the value $(\lambda z.z)$ in an environment that associates x with $\lambda z.z$, but is *not* a value *per se*. Practically speaking, such an expression is a syntactic representation of a *closure*. The complete set of closures is

$$A ::= \lambda x.M \mid ((\lambda x.A)M) \ .$$

We also use *answer* when we speak about closures.

Reducing expressions to closures means that our replacement for β does not suffice to reduce all expressions to closures. Consider the application

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

The argument is an answer but not a value. Moreover, it cannot be reduced further with $(deref)$. Consequently, the entire term is irreducible because the argument cannot be used to replace f, which is in the hole of an evaluation context. One solution is to allow substitution of answers for variables, that is, we could use the following extension of $(deref)$:

$$(\lambda x.E[x])A = (\lambda x.E[A])A \ .$$

Using this revision, the above term could be transformed as follows

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

However, the effect of substituting answers is that unevaluated arguments in the

¶ The use of the fill operation on contexts is justified in this definition because it is a well-defined operation on all contexts, of which we only specify a subset.

environment of closures are duplicated. Specifically, in our example, the redex II has been duplicated and is subsequently reduced twice.

In our syntactic model of evaluation, the duplication of unevaluated arguments can be avoided by a re-association of bindings. In our running example, we can rearrange the association of f with $((\lambda z.\lambda w.zw)\,(II))$ as follows:

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

The substitution for f will then avoid copying II:

$$(\lambda z.(\lambda f.fI(fI))(\lambda w.zw))(II) =_{deref} (\lambda z.(\lambda f.(\lambda w.zw)I(fI))(\lambda w.zw))(II) \quad , \qquad (2)$$

and thus, even though f does occur twice, II will be contracted only once. The re-association of arguments and parameters is captured by the following axiom:

$$assoc: \qquad (\lambda x.E[x])((\lambda y.A)M) = (\lambda y.(\lambda x.E[x])A)M \quad .$$

It says that if an evaluation demands the value of some parameter yet the parameter is associated with a closure instead, then the bindings of the closure must be shared between the main procedure and the closure's body.

Dually, the evaluation of expressions to answers instead of values also means that the expression in the function position of an application can yield an answer in place of a function. Hence, an evaluation might become stuck because the axiom *deref* cannot deal with this situation. An instance of this problem is the expression $(\lambda z.\lambda w.zw)(II)I$. To solve this problem, we introduce an axiom like *assoc* for the function position of an application:

$$lift: \qquad (\lambda x.A)MN = (\lambda x.AN)M \quad .$$

With *lift*, we can now evaluate the above expression:

$$(\lambda z.\lambda w.zw)(II)I = (\lambda z.(\lambda w.zw)I)(II) \quad .$$

The re-association of a closure's bindings is not just a syntactic nuisance. It captures implicit operations in both an environment-based implementation and a graph-reduction machine. Specifically, the action of the *assoc* axiom corresponds to a hidden flattening of the environments in Launchbury's description of a lazy interpreter (Launchbury, 1993):

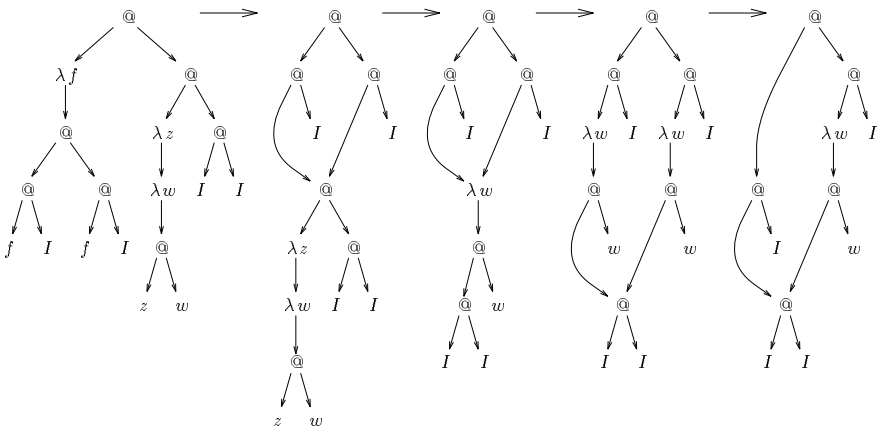
$$\dfrac{\dfrac{(z \mapsto II) : \lambda w.zw \Downarrow (z \mapsto II) : \lambda w.zw}{\{\} : \mathsf{let}\ z\ \mathsf{be}\ II\ \mathsf{in}\ \lambda w.zw \Downarrow (z \mapsto II) : \lambda w.zw}\ \mathsf{Let}}{(f \mapsto \mathsf{let}\ z\ \mathsf{be}\ II\ \mathsf{in}\ \lambda w.zw) : f \Downarrow (z \mapsto II, f \mapsto \lambda w.zw) : \lambda w.zw}\ \mathsf{Variable} \quad .$$

According to the Variable rule we evaluate let z be II in $\lambda w.zw$ in an empty environment. This evaluation, following the Let rule, leads to the value $\lambda w.zw$ and a new environment in which z is associated with II. At this point the implicit flattening occurs: instead of associating f to an environment-value pair, f is bound to $\lambda w.zw$. The *assoc* axiom makes this step explicit in our calculus.

Similarly, following Wadsworth's interpreter we can depict the reduction of

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

as follows:



The third step represents a copy operation, which is required by the fact that there are two pointers to the node labelled λw. In other words, given a β redex $(\lambda x.M)N$, if $\lambda x.M$ is shared then, before performing a substitution of the pointer to N inside M, a copy of M has to take place. However, referring to the above picture, note that not all nodes reachable from λw have been copied. In fact, the redex II has not been copied. Thus, a decision has to be made on what to copy and what not. This requires scanning the function's body, and thus is an expensive operation. To reduce its cost Arvind et al. (1984) proposed a different graph interpreter, called *flagged* interpreter. The basic idea is: given a β redex $(\lambda x.M)N$ a pointer to a "flagged" N is substituted in M. Then the copy phase of a β-step copies only those nodes of the function's body that are not reachable from a flagged node. Referring to the above reduction, this means that given the redex $(\lambda z.\lambda w.zw)(II)$, instead of substituting a pointer to II in the function's body, a pointer to a flagged II, say $(II)^*$, is substituted, obtaining $\lambda w.(II)^* w$. When a copy of the λw-abstraction is demanded, we then do not copy II. In such a way no analysis of the function's body is then carried out at run time. This operation of "flagging" corresponds to our *assoc* axiom. In fact, note that the term on the right-hand side of equation (2) represents the fourth graph depicted above, that is, the one obtained after the copy operation. As it will be discussed more in length in Section 6, more work is demanded in order to describe Wadsworth's interpreter.

4 The call-by-need calculus

We summarize the discussion of the previous section in the following definition:

Definition 4.1

[The Call-by-Need Calculus λ_{need}] The following clauses define the syntax and basic axioms of the call-by-need calculus.

Syntax:

Expressions (Λ):	M	::=	$x \mid \lambda x.M \mid MM$
Values:	V	::=	$\lambda x.M$
Answers:	A	::=	$V \mid ((\lambda x.A)\ M)$
Evaluation Contexts:	E	::=	$[\,] \mid EM \mid (\lambda x.E[x])E \mid (\lambda x.E)M$

Axioms:

$$(\lambda x.E[x])V = (\lambda x.E[V])V \qquad deref$$
$$(\lambda x.A)MN = (\lambda x.AN)M \qquad lift$$
$$(\lambda x.E[x])((\lambda y.A)M) = (\lambda y.(\lambda x.E[x])A)M \qquad assoc$$

$=$ is an equivalent and congruent relation. $\xrightarrow[need]{}$ and $\xrightarrow[need]{}\!\!\!\!\!\rightarrow$ denote the one-step and multiple-step call-by-need reductions, respectively.

The call-by-need calculus enjoys nice properties, namely, it is confluent and there exists a standard reduction that leads to an answer. We start with the proof of confluence. To that end, the following lemma proves that reductions do not interfere with each other. Huet refers to this property as "absence of critical pairs" (Huet, 1980).

Lemma 4.2

(i) A term M can be at most one redex.
(ii) If a term M is a redex then $M \not\equiv M_1M_2$ with M_1 a redex.
(iii) The sets *Answers*, *Values* and $\{E[x] \mid E \in$ Evaluation Contexts$\}$ are each closed under reduction.

Proof

(i) Trivial if M is a variable or a lambda abstraction. If M is an application the result follows from a case analysis on the type of redex.
(ii) By cases on the type of redex.
(iii) An easy induction on the structure of the term.

□

Note: In contrast, the call-by-need calculus advocated by Maraist et al. (1994; 1995), is a collection of interfering axioms. The basic axioms of Maraist et al.'s calculus are:

$$(\lambda x.C[x])V = (\lambda x.C[V])V \qquad deref'$$
$$(\lambda x.L)MN = (\lambda x.LN)M \qquad lift'$$
$$(\lambda x.L)((\lambda y.N)M) = (\lambda y.(\lambda x.L)N)M \qquad assoc'$$

Given this system of axioms, the term M:

$$M \equiv \underbrace{\underbrace{(\lambda x.x)((\lambda y.y)z)}_{U_1}\, N}_{U_2}$$

contains two overlapping redexes, U_1 and U_2. Specifically, U_1 is an *assoc'* redex, while U_2 is a *lift'* redex. The two reductions, as shown next, still converge.

$$\begin{array}{ccc}
(\lambda x.x)((\lambda y.y)z)N & \xrightarrow{assoc'} & (\lambda y.(\lambda x.x)y)zN \\
\Big\downarrow{\scriptstyle lift'} & & \Big\downarrow{\scriptstyle lift'} \\
 & & (\lambda y.(\lambda x.x)yN)z \\
 & & \Big\downarrow{\scriptstyle lift'} \\
(\lambda x.xN)((\lambda y.y)z) & \xrightarrow{assoc'} & (\lambda y.(\lambda x.xN)y)z
\end{array}$$

According to our axioms, U_1 is not an *assoc*-redex because $(\lambda y.y)z$ is not an answer. U_2 is not a *lift*-redex because $(\lambda x.x)$ is not of the form $(\lambda x.A)$. By extending the notion of value to include variables, our axioms would enable this overlapping. This example points out another important difference between our system and the one proposed by Maraist et al. As shown above, the *assoc'* step has caused an implicit duplication. In our system the only duplication is caused by the *deref* axiom, and as such is explicit. This explicit control over duplication makes meta-reasoning about the system easier. **End**

Since the metavariables occurring in the left-hand side of the axioms occur only once, λ_{need} satisfies the *left-linearity* property. Because of left-linearity and the absence of overlapping, λ_{need} can be expressed as a conditional orthogonal higher order rewriting system (HORS) (van Oostrom, 1994).‖

Lemma 4.3 (Parallel Move Lemma)
Let $M \twoheadrightarrow M_1$ and $M \xrightarrow{U} M_2$, then a common reduct M_3 of M_1 and M_2 can be found by a complete development of the set of descendants of redex U occurring in M_1.

Proof
From Lemma 4.2 the descendants of redexes are still redexes. The rewriting steps are thus independent, and as such they commute. □

Notation: Referring to the above lemma, let B be the reduction $M \twoheadrightarrow M_1$. We denote the set of descendants of U with respect to the reduction B as U/B. Moreover, the reduction $M_2 \twoheadrightarrow M_3$ is called the projection of B with respect to the U-reduction, and is denoted by B/U. We will also make use of the notation U/U_1 to denote the descendants of U with respect to the U_1-reduction.

By repeated application of the Parallel Move Lemma we get a Consistency (Church Rosser) theorem for the call-by-need calculus.

Theorem 4.4
The call-by-need calculus is confluent: for all terms $M, M_1, M_2 \in \Lambda$ such that $M \twoheadrightarrow M_1$ and $M \twoheadrightarrow M_2$ there exists M_3 such that $M_2 \twoheadrightarrow M_3$ and $M_1 \twoheadrightarrow M_3$.

‖ Personal communication with Vincent van Oostrom.

The confluence property follows also at once from λ_{need} being a conditional HORS. However the Parallel Move Lemma is stronger than confluence, since it suggests how to find a common reduct. The Parallel Move Lemma is also useful in proving the Standardization Theorem, to which we turn our attention next. Let us first define what it means for a term to standard reduce to another term.**

Definition 4.5
Given a program M, M *standard reduces* to N, written as $M \xrightarrow[need]{} N$, if and only if $M \equiv E[U]$ and $N \equiv E[L]$, where U and L are a redex and its contractum, respectively. $M \xrightarrow[need]{}\!\!\!\!\!\rightarrow N$ means M, N are related via the transitive-reflexive closure of $\xrightarrow[need]{}$.

The following is an example of standard reduction (I stands for $\lambda z.z$):

$$
\begin{array}{ll}
(\lambda f.fI(fI))(\lambda w.(II)w) \equiv E_1[(\lambda f.fI(fI))(\lambda w.(II)w)] & \xrightarrow[deref]{} \\
(\lambda f.(\lambda w.(II)w)I(fI))(\lambda w.(II)w) \equiv E_2[(II)] & \xrightarrow[deref]{} \\
(\lambda f.(\lambda w.(\lambda z.I)Iw)I(fI))(\lambda w.(II)w) \equiv E_3[(\lambda z.I)Iw] & \xrightarrow[lift]{} \\
E_3[(\lambda z.Iw)I] \ . &
\end{array}
$$

The evaluation contexts E_1, E_2 and E_3 are:

$$
\begin{array}{lcl}
E_1 & \equiv & [\,] \\
E_2 & \equiv & (\lambda f.(\lambda w.[\,]w)I(fI))(\lambda w.(II)w) \\
E_3 & \equiv & (\lambda f.(\lambda w.[\,])I(fI))(\lambda w.(II)w) \ .
\end{array}
$$

Instead, the following reduction is not standard:

$$
\begin{array}{ll}
(\lambda f.fI(fI))(\lambda w.(II)w) & \xrightarrow[deref]{} \\
(\lambda f.fI(fI))(\lambda w.(\lambda z.I)Iw) & \xrightarrow[deref]{} \\
(\lambda f.(\lambda w.(\lambda z.I)Iw)I(fI))(\lambda w.(\lambda z.I)Iw) \ . &
\end{array}
$$

In fact, there is no evaluation context such that $(\lambda f.fI(fI))(\lambda w.(II)w) \equiv E[II]$.

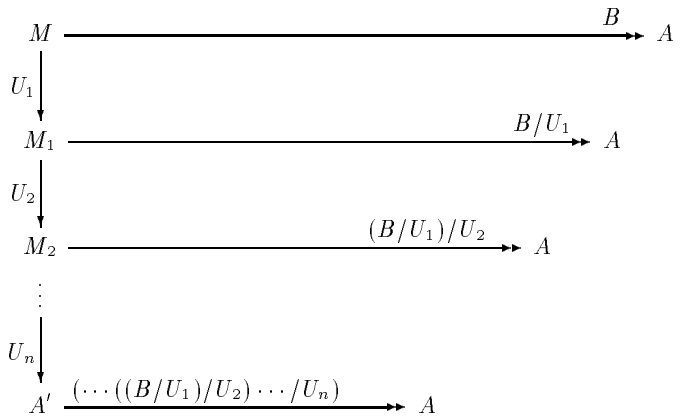
Verifying that the *standard* relation, $\xrightarrow[need]{}$, is indeed a function from programs to programs relies on the usual Unique Evaluation Context lemma (Felleisen & Friedman, 1986). It states that there is a unique partitioning of a non-answer into an evaluation context and a redex, which implies that there is precisely one way to make progress in the evaluation.

Lemma 4.6
Given a program M, M is either an answer or there exists a unique evaluation context E and redex N such that $M \equiv E[N]$.

Corollary 4.7
$\xrightarrow[need]{}$ is a partial function.

** We only prove the important part of the Standardization Theorem, namely, that a standard reduction starting with a program will find an answer if it exists.

We show next that the standard reduction leads to an answer if there exists one. We follow the proof technique of Huet and Lévy (1991). Let B be the reduction $M \twoheadrightarrow A$. We first show that the reduction B contracts the descendant of the standard redex, say U_1, occurring in M. We then construct the projection of the reduction B with respect to the U_1-reduction, *i.e.*, the reduction B/U_1. Since the reduction B/U_1 also leads to an answer, we can proceed by performing the projection $(B/U_1)/U_2$, where U_2 is the standard redex contracted by the reduction B/U_1. As before, also $(B/U_1)/U_2$ leads to an answer. The termination of such a process is guaranteed by showing that at each step the weight associated to each reduction decreases. That is, if we call $ord(B)$ the weight associated to the reduction B, then we want to show $ord(B) > ord(B/U_1) > ord((B/U_1)/U_2) > \cdots$. Pictorially:



The reduction $M \xrightarrow{U_1} M_1 \xrightarrow{U_2} M_2 \cdots \xrightarrow{U_n} A'$ is the desired standard reduction.

We first point out that the reduction of a non standard redex cannot create a standard redex.

Lemma 4.8
Let $M \xrightarrow{U} M_1$, with U a non-standard redex.
(i) If M_1 is an answer then M is an answer;
(ii) If M_1 is of the form $E[x]$ then M is also of the form $E[x]$;
(iii) If M_1 is a redex then M is a redex.

The following lemma shows that each reduction to an answer reduces the descendant of the standard redex.

Lemma 4.9
Given a program M. If $M \twoheadrightarrow A$ then either M is an answer or M is of the form $E[U]$ with U a redex, such that a descendant of U, say U', is reduced in the reduction from M to A. That is, $\exists M_1, M_2$, $M \twoheadrightarrow M_1 \xrightarrow{U'} M_2 \twoheadrightarrow A$.

Proof
By induction on the number n of reduction steps of $M \twoheadrightarrow A$.

$n = 0$: M must be an answer.

$n > 0$: Let $M \xrightarrow{U_1} M_1 \longrightarrow^{n-1} A$. If U_1 is the standard redex in M, then U_1 is the redex we are looking for. Otherwise, by induction hypothesis M_1 must be of the form $E[U']$, such that a descendant of U' is reduced in the reduction to A. By structural induction on M_1 and the previous lemma it then follows that U' is the descendant of the standard redex in M.

□

Due to the Parallel Move Lemma we can define a multiple steps reduction as follows.

Definition 4.10
Let $\mathcal{F}$ be a set of distinct redexes in M. Then $M \xrightarrow[cdv\ \mathcal{F}]{} N$ denotes a complete development of M with respect to $\mathcal{F}$. We will write $M \xrightarrow[cdv]{} N$ when the set of redexes reduced is clear from the context. As usual, $\xrightarrow[cdv]{}\!\!\!\!\!\rightarrow$ denotes the transitive and reflexive closure of $\xrightarrow[cdv]{}$.

For example,

$$(\underline{\lambda}x.xN)(\lambda z.(\lambda w.w)(\lambda w.w)) \xrightarrow[cdv]{} (\lambda x.(\lambda z.(\underline{\lambda}w.w)(\lambda w.w))N)(\lambda z.(\underline{\lambda}w.w)(\lambda w.w))$$
$$\xrightarrow[cdv]{} (\lambda x.(\lambda z.(\lambda w.(\lambda w.w))(\lambda w.w))N)(\lambda z.(\lambda w.(\lambda w.w))(\lambda w.w))$$

where at each step the underlined redexes are reduced.

To a reduction $M \xrightarrow[cdv]{}\!\!\!\!\!\rightarrow N$ we associate a tuple, which represents its weight.

Definition 4.11
Let B be the reduction: $M \xrightarrow[cdv\ \mathcal{F}_1]{} M_1 \xrightarrow[cdv\ \mathcal{F}_2]{} \cdots \xrightarrow[cdv\ \mathcal{F}_n]{} M_n$. The weight of B, written as $ord(B)$, is defined as:

$$\langle |\mathcal{F}_n|, \cdots, |\mathcal{F}_2|, |\mathcal{F}_1| \rangle$$

where $|\mathcal{F}|$ denotes the size of set $\mathcal{F}$. Note the change of order.

Since $M \longrightarrow M_1$ implies that $M \xrightarrow[cdv]{} M_1$, we can thus associate a weight to each reduction. For example, to the reduction $M \rightarrow M_1 \rightarrow M_2 \rightarrow M_3$ we associate the weight : $\langle 1, 1, 1 \rangle$. Using the lexicographical ordering among tuples, written as $>$, we can then order the reductions.

Before addressing the standardization theorem, we prove that the descendants of a redex are mutually disjoint. We call two redexes disjoint if they are not nested inside each other. The underlined redexes in $(\underline{\lambda}x.xN)(\lambda z.(\underline{\lambda}w.w)(\lambda w.w))$ are not disjoint, whereas the underlined redexes in

$$(\lambda x.(\lambda z.(\underline{\lambda}w.w)(\lambda w.w))N)(\lambda z.(\underline{\lambda}w.w)(\lambda w.w))$$

are disjoint. The importance of this property is that no duplication can occur among disjoint redexes.

Lemma 4.12
Let U and U_1 be two distinct redexes in M. If $M \xrightarrow{U} M_1$ then all redexes in U_1/U are mutually disjoint.

Proof

Trivial from the analysis of the *deref* axiom, which is the only axiom that duplicates terms. □

We are now ready to state and prove the essence of the Standard Reduction Theorem, which suffices to show that a call-by-need interpreter is an implementation of the standard reduction function on non-answers.

Theorem 4.13

Given a program M, $M \twoheadrightarrow A$ if and only if there exists an answer A' such that $M \mapsto\!\!\!\!\twoheadrightarrow A' \twoheadrightarrow A$.

Proof

$\Longleftarrow$ Follows from the fact that $\mapsto\!\!\!\!\twoheadrightarrow \subset \twoheadrightarrow$.

$\Longrightarrow$ Let B be the reduction $M \twoheadrightarrow A$. Without loss of generality let us assume $M \xrightarrow{U_1} M_1 \xrightarrow{U_2} M_2 \xrightarrow{U_3} A$. If M is not an answer, then by Lemma 4.9 the reduction B reduces the descendant of the standard redex U occurring in M. Let $M \overset{U}{\longmapsto} N$ and let us construct the reduction B/U as follows:

$$\begin{array}{ccccccc} M & \xrightarrow{U_1} & M_1 & \xrightarrow{U_2} & M_2 & \xrightarrow{U_3} & A \\ {\scriptstyle U}\downarrow & & {\scriptstyle U/U_1}\downarrow & & {\scriptstyle (U/U_1)/U_2}\downarrow & & \equiv \\ N & \xrightarrow[cdv\ U_1/U]{} & N_1 & \xrightarrow[cdv\ U_2/(U/U_1)]{} & N_2 & \xrightarrow[cdv\ U_3/((U/U_1)/U_2)]{} & A \end{array} .$$

The above diagram shows that the standard redex U does not get duplicated, because the only rule that causes duplication is the *deref* rule, and the redex duplicated, if any, occurs under a lambda. Let us assume the descendant of U is U_2. Thus, $|U_2/(U/U_1)| = 0$, and $|U_3/((U/U_1)/U_2)| = 1$. Then,

$$ord(B) = \langle 1,1,1\rangle > \langle 1,0,|U_1/U|\rangle = ord(B/U) \ .$$

If N is an answer then we are done. Otherwise we can repeat the process above. That is, we compute the projection of B/U with respect to U', the standard redex in N. It remains to show that the weight keeps decreasing, that is, $ord(B/U) > ord((B/U)/U')$. If the descendant of U' is U_3 then $ord((B/U)/U')$ is $\langle 0,0,|\ U_1/U\ |\rangle$, which is obviously less than $ord(B/U)$. Otherwise, since the standard reduction cannot duplicate a redex in U_1/U (by Lemma 4.12), we have:

$$ord(B/U) = \langle 1,0,|U_1/U|\rangle > \langle 1,0,|U_1/U| - 1\rangle = ord((B/U)/U') \ .$$

Since at each projection the weight of the reduction decreases, the process will terminate. In conclusion, since the standard redex criterion is syntactic, that is, independent of the reduction, it must be the case that the constructed reduction is the standard one.

□

5 Correctness of the call-by-need evaluator

Resuming Plotkin's program again, we define the call-by-need evaluator as a partial function $\mathtt{Eval}_{\mathsf{need}}$ from *programs*, or closed terms, to the singleton consisting of the tag closure based on equality in the call-by-need calculus:

$$\mathtt{Eval}_{\mathsf{need}}(M) = \mathsf{closure} \text{ if and only if } \boldsymbol{\lambda}_{\mathrm{need}} \vdash M = A.$$

Theorem 5.1
For a program M, $\mathtt{Eval}_{\mathsf{need}}(M) = \mathsf{closure}$ if and only if $M \longmapsto\!\!\!\!\!\rightarrow A$.

Proof
From the confluency property (Theorem 4.4) and the Standardization Theorem of λ_{need} (Theorem 4.13). □

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

$$\mathtt{Eval}_{\mathsf{need}} = \mathtt{Eval}_{\mathsf{name}} \ .$$

In this section, we will show that the two (interpreters treated as) sets are indeed subsets of each other. One direction is nearly obvious: the call-by-name calculus can clearly prove every equation that the call-by-need calculus can prove.

Theorem 5.2 (Soundness)
$\boldsymbol{\lambda}_{\mathrm{need}} \subset \boldsymbol{\lambda}$.

Proof
Each call-by-need axiom is an equation in the call-by-name calculus:

assoc: $\boldsymbol{\lambda}_{\mathrm{need}} \vdash (\lambda x.E[x])((\lambda y.A)M) = (\lambda y.(\lambda x.E[x])A)M$. By the variable convention, $x \neq y$ and $x \notin FV(M)$. From the Substitution Lemma (Lemma 2.2),

$$\boldsymbol{\lambda} \vdash E[x][y := M][x := A[y := M]] = E[x][x := A][y := M] \ .$$

Since $y \notin FV(E[x])$ (variable convention):

$$\boldsymbol{\lambda} \vdash E[x][x := A[y := M]] = E[x][x := A][y := M] \ .$$

The same for *deref* and *lift*. A simple example for the strictness of the inclusion is the equation

$$(\lambda x.xx)\Omega = \Omega\Omega \ .$$

As a straightforward corollary of the Standard Reduction Theorem, the call-by-need calculus cannot prove this equation. □

Intermezzo 5.3
Whereas $\boldsymbol{\lambda}_{\mathrm{need}}$ is incomparable with $\boldsymbol{\lambda}_V$, the call-by-value theory, Maraist et al.'s call-by-need theory (Maraist *et al.*, 1994), $\boldsymbol{\lambda}_{mow}$, is a proper superset of $\boldsymbol{\lambda}_V$ and a proper subset of $\boldsymbol{\lambda}$, that is,

$$\boldsymbol{\lambda}_V \subset \boldsymbol{\lambda}_{mow} \subset \boldsymbol{\lambda} \ .$$

To accomplish this relationship, Maraist et al. use the axiom

$$deref' \qquad (\lambda x.C[x])V = (\lambda x.C[V])V$$

as the replacement for (β). However, since this change does not suffice to turn their calculus into a superset of the call-by-value theory, they make further modifications. First, they add an axiom on garbage collection because β_V, the basic call-by-value axiom, implicitly deletes values that are no longer needed. The garbage collection axiom proves equations like

$$(\lambda x.\lambda y.y)V = \lambda y.y \ ,$$

which a theory based on just *deref'* cannot prove. Second, Maraist et al. include variables in the set of values so that their theory can prove equations like

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

As explained in Section 3, we disagree with *deref'* because the axiom does not faithfully reflect the intensional behavior of a call-by-need machine. Since the purpose of a call-by-need calculus is to capture the intensional aspects of modern call-by-need evaluators, *deref'* should not be the basic axiom of an equational call-by-need characterization. Furthermore, we disagree with the inclusion of variables in the set of values because the first lookup of a variable triggers an unbounded computation. The inclusion of variables in the set of values is therefore misleading because it may represent a proper computation. Finally, we believe that the inclusion of the garbage collection axiom should be optional. **End**

The direction:

$$\mathtt{Eval}_{\mathsf{name}}(M) = \mathsf{closure} \Longrightarrow \mathtt{Eval}_{\mathsf{need}}(M) = \mathsf{closure} \ ,$$

requires more machinery, as we explain next.

5.1 Completeness of the call-by-need calculus

Thus far we have used lambda-bound variables to express sharing. In order to show that the call-by-need calculus can simulate a call-by-name reduction, we now introduce "lets" to express sharing in Λ terms. Even though a different proof could be done directly on Λ, we believe that our approach increases the reader's intuition. Specifically, we introduce a new axiom, as follows:

$$\beta_{let} : \qquad (\lambda x.M)N = \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M \ .$$

We then reformulate the axioms of Definition 4.1 on the extended syntax.

Definition 5.4

[λ_{let}] The following clauses define the syntax and basic axioms of the let-calculus.

Syntax:

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

Axioms:

$$(\lambda x.M)N = \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M \qquad \beta_{let}$$

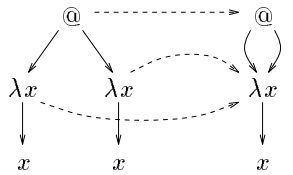
$$\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[x] = \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[V] \qquad deref_{let}$$

$$(\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ A)N = \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ AN \qquad lift_{let}$$

$$\mathsf{let}\ x\ \mathsf{be}\ \mathsf{let}\ y\ \mathsf{be}\ M\ \mathsf{in}\ A\ \mathsf{in}\ E[x] = \mathsf{let}\ y\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ A\ \mathsf{in}\ E[x] \qquad assoc_{let}$$

Notation: $\xrightarrow[let]{}$, $\xrightarrow[let]{}\!\!\!\!\!\rightarrow$ denote the one-step and multiple-step reductions in λ_{let}. Analogously to Definition 4.5 we can define a notion of standard reduction, written as $\xmapsto[let]{}$ and $\xmapsto[let]{}\!\!\!\!\!\rightarrow$.

The key step of the completeness proof is the introduction of an ordering between terms of λ_{let}. Intuitively, $M \leq N$ if M can be obtained by unwinding N, that is, the $\leq$-ordering captures the amount of sharing contained in a term (Ariola & Arvind, 1995). The ordering relation $\leq$ expresses whether the terms have homomorphic graphs. For example, the graph of $M \equiv (\lambda x.x)(\lambda x.x)$ can be homomorphically embedded into the graph of $N \equiv \mathsf{let}\ y = \lambda x.x\ \mathsf{in}\ yy$,



which shows that $M \leq N$. Notation: given a term $M \in \Lambda_{\text{let}}$, let $\mathcal{D}ag(M)$ be the corresponding directed acyclic graph (dag).

Definition 5.5
For $M, N \in \Lambda_{\text{let}}$, $M \leq N$ if and only if there exists an homomorphism $\sigma : \mathcal{D}ag(M) \rightarrow \mathcal{D}ag(N)$.

The main result of this section is that if $M \leq N$, and M is a call-by-name term, that is, M does not contain a let expression, then each standard call-by-name reduction of M can be simulated in the let-calculus by reducing N. The term obtained in the let-calculus is not necessarily greater (in terms of the above ordering) than the one obtained following the call-by-name reduction, because the single step in the let-calculus that simulates a call-by-name step may actually correspond to many steps in the call-by-name lambda calculus. Consider the following reduction:

$$M \equiv ((\lambda y.y)(\lambda z.z))((\lambda y.y)(\lambda z.z)) \xrightarrow[name]{} (\lambda z.z)((\lambda y.y)(\lambda z.z)) \equiv M_1$$

and the term $N \equiv \mathsf{let}\ x\ \mathsf{be}\ (\lambda y.y)(\lambda z.z)\ \mathsf{in}\ xx$, which is larger than M (in our sharing ordering). The required reduction step in the let-calculus that simulates the

reduction step in the call-by-name calculus is

$$\mathsf{let}\ x\ \mathsf{be}\ (\lambda y.y)(\lambda z.z)\ \mathsf{in}\ xx \xrightarrow[let]{} \mathsf{let}\ x\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ \lambda z.z\ \mathsf{in}\ y)\ \mathsf{in}\ xx \equiv N_1 \ .$$

Note though, N_1 does not dominate M_1. However, there exists M_2 such that $M_1 \xrightarrow[name]{} M_2$ and $M_2 \leq N_1$:

$$M_1 \xrightarrow[name]{} M_2 \equiv (\lambda z.z)(\lambda z.z) \leq N_1 \ .$$

Since $M \leq M$, the result shows how to reconstruct a call-by-name reduction in the let-calculus.

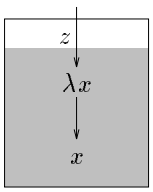
The main difficulty of the proof is that if $M \leq N$ and M is a β-redex, N is not necessarily a β_{let} redex. Suppose

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

then $M \leq N$, yet N does not contain a β_{let} redex. This is so because the language of dags captures only the sharing in a term but not its let-structure. We thus enrich dags with boxes and labeled edges (Ariola & Klop, 1996). $\mathcal{D}ag_{\Box}(M)$ is the decorated dag associated with a term 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 $\mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M$ 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). If M is x then the unshaded area will only contain the edge labelled x that leads to the shaded area containing N. Let us illustrate the extended dag notation and terminology with a number of examples.

Example 5.6

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

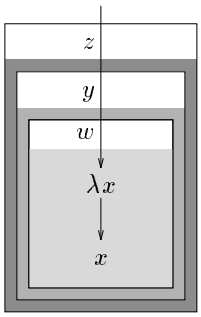


The name z associated with the root pointer is drawn outside the shaded area.

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

$$\begin{array}{l} \mathsf{let}\ z\ \mathsf{be}\ \ \mathsf{let}\ y\ \mathsf{be}\ \ \mathsf{let}\ w\ \mathsf{be}\ \lambda x.x \\ \qquad\qquad\qquad\qquad\ \ \mathsf{in}\ w \\ \qquad\qquad\ \ \mathsf{in}\ y \\ \mathsf{in}\ z \end{array}$$

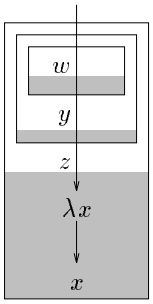
is drawn as



where the path to the λ-node must follow the label zyw and hit three walls. In contrast, in the term

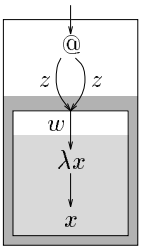
$$\begin{array}{l}\mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\\ \mathsf{in}\ \ \mathsf{let}\ y\ \mathsf{be}\ z\\ \quad\ \mathsf{in}\ \ \mathsf{let}\ w\ \mathsf{be}\ y\\ \quad\quad\ \mathsf{in}\ w\ \ ,\end{array}$$

drawn as



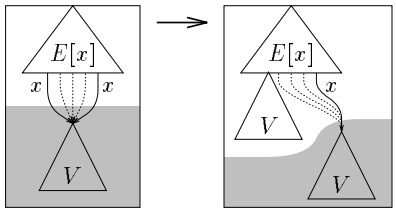
the path to the λ-node must follow the label wyz. The path penetrates three walls but also leaves two encasings.

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

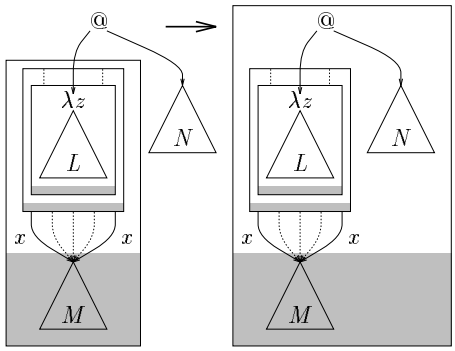


In this decorated dag, the path from the application (root) node to the λ-node has label zw, and it penetrates the wall of one box. This label and the wall are obstacles that must be eliminated in order to create a redex. In terms of our graphical language, we must eliminate the names z, w, the internal box, and pull the λ-node out of the shaded area, which is exactly the task of the $assoc_{let}$, $deref_{let}$, and $lift_{let}$ rules. Their dag-based representation in Table 1 reveals that $deref_{let}$ pulls a value

$deref_{let}$:



$lift_{let}$:



$assoc_{let}$:

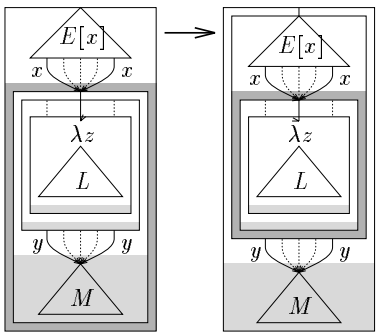


Table 1. *Rules of the let-calculus in dag-based form.*

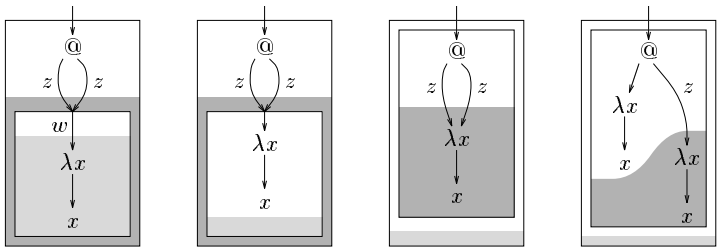


Fig. 1. Exposing the redex in let z be (let w be $\lambda x.x$ in w) in zz.

out of the shaded area, eliminating a name on an edge; $lift_{let}$ moves a wall; and $assoc_{let}$ moves a wall that is in the shaded area. The sequence $deref_{let}$, $assoc_{let}$, $deref_{let}$ suffices to expose the redex in our example:

$$\begin{aligned}
&\mathsf{let}\ z\ \mathsf{be}\ (\mathsf{let}\ w\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ w)\ \mathsf{in}\ zz\\
&\xrightarrow[deref_{let}]{}\quad \mathsf{let}\ z\ \mathsf{be}\ (\mathsf{let}\ w\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x)\ \mathsf{in}\ zz\\
&\xrightarrow[assoc_{let}]{}\quad \mathsf{let}\ w\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ (\mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ zz)\\
&\xrightarrow[deref_{let}]{}\quad \mathsf{let}\ w\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ (\mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ (\lambda x.x)z)\ \ .
\end{aligned}$$

Figure 1 (without unreachable dags) illustrates these steps.

The language and notation for decorated dags are useful in proving the following four key lemmas.

Notation: we denote with $M \xrightarrow[\{a,l\}_{let}]{}\!\!\!\!\!\rightarrow N$ a sequence of $assoc_{let}$ and $lift_{let}$ reductions. Likewise, $M \xrightarrow[\{a,d,l\}_{let}]{}\!\!\!\!\!\rightarrow N$ denotes a sequence of $assoc_{let}$, $deref_{let}$ and $lift_{let}$ reductions.

From Table 1 it is clear that $lift_{let}$ and $assoc_{let}$ do not change the dag associated with a term, while $deref_{let}$ causes a duplication.

Lemma 5.7

(i) Given $M \in \Lambda_{\mathrm{let}}$, if $M \xrightarrow[\{a,l\}_{let}]{}\!\!\!\!\!\rightarrow N$ then $\mathcal{D}ag(M) = \mathcal{D}ag(N)$.
(ii) Given $M \in \Lambda_{\mathrm{let}}$, if $M \xrightarrow[deref_{let}]{}\!\!\!\!\!\rightarrow N$ then $N \leq M$.

The next lemma shows that adding sharing in a term does not destroy a β-redex. The β-redex can be reconstructed by using the $assoc_{let}$, $lift_{let}$ and $deref_{let}$ axioms.

Lemma 5.8
Given $M \in \Lambda, N \in \Lambda_{\mathrm{let}}$, if $M \leq N$ and $M \equiv E_n[(\lambda x.P)R]$ then there exists P', R', and $E[\,]$, such that $N \xrightarrow[\{a,d,l\}_{let}]{}\!\!\!\!\!\rightarrow E[(\lambda x.P')R']$.

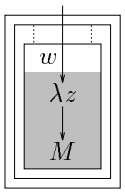
Proof
Let z be the root in $\mathcal{D}ag(M)$ of the β-redex being evaluated. Let z' and $z'_{\Box}$ be the corresponding nodes in $\mathcal{D}ag(N)$ and $\mathcal{D}ag_{\Box}(N)$, respectively. We know that the left branch of z' points to a λ-node, while in $\mathcal{D}ag_{\Box}(N)$ the path from $z'_{\Box}$ to the λ-node may contain some obstacles. The goal is to show that by using $assoc_{let}$, $deref_{let}$,

and $lift_{let}$ 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 $lift_{let}$ steps to move the walls and expose the redex.

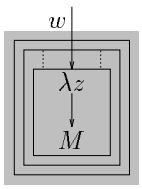
$n > 0$: By the induction hypothesis, we can remove n-1 names. Now we need to show how to eliminate the last one. There are two cases:

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



   Since w occurs in head position, an application of $deref_{let}$ 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 $assoc_{let}$ followed by a single application of $deref_{let}$ expose the λ-node.

So far we have shown that there exists P', R' and $C[\,]$ such that $N \xrightarrow[\{a,d,l\}_{let}]{}\!\!\!\!\!\rightarrow N'$ and $N' \equiv C[(\lambda x.P')R']$, with the redex $(\lambda x.P')R'$ rooted at $z'_{\Box}$ in $\mathcal{D}ag_{\Box}(N')$. Since z is the root of the leftmost-outermost redex in M, it must be that $(\lambda x.P')R'$ is needed. Therefore, $C[\,]$ must be an evaluation context. Moreover, since the reduction $N \xrightarrow[\{a,d,l\}_{let}]{}\!\!\!\!\!\rightarrow N'$ is to expose a needed redex it must be the case that the reduction is standard, that is, $N \xmapsto[\{a,d,l\}_{let}]{}\!\!\!\!\!\rightarrow N'$. □

The third lemma in our series shows that the effect of adding sharing is to turn a value into an answer.

Lemma 5.9
Given $M \in \Lambda, N \in \Lambda_{\mathrm{let}}$, if $M \leq N$ and $M \equiv \lambda x.P$ then there exists an answer A such that $N \xmapsto[\{a,d\}_{let}]{}\!\!\!\!\!\rightarrow A$.

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 $lift_{let}$ is required. □

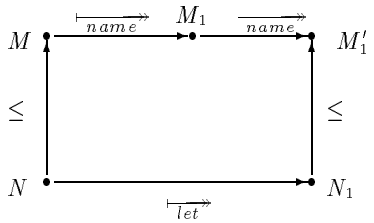
Finally, we can show that a call-by-name evaluation can be simulated in the let-calculus.

Lemma 5.10

Given a program $M \in \Lambda$ and $N \in \Lambda_{\text{let}}$, if $M \leq N$ and $M \xrightarrow[name]{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow M_1$, then $\exists M_1' \in \Lambda, N_1 \in \Lambda_{\text{let}}$ such that

$$M_1 \xrightarrow[name]{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow M_1', N \xrightarrow[let]{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow N_1 \text{ and } M_1' \leq N_1.$$

Pictorially:



Proof

By induction on the length n of the reduction $M \xrightarrow[name]{}^{n} M_1$.

$n = 1$: Let $M \equiv E_n[(\lambda x.P)R]$, and let z be the root in $\mathcal{D}ag(M)$ of the β-redex in the hole of $E_n[\,]$. From Lemma 5.8, there exists P', R' and $E[\,]$:

$$N \xrightarrow[let]{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow N' \text{ and } N' \equiv E[(\lambda x.P')R'] \ .$$

Let z' be the root in $\mathcal{D}ag(N')$ of the β_{let}-redex. From Lemma 5.7 and the fact that M does not contain any sharing we have $M \leq N'$. Thus:

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

and

$$N' \xrightarrow[\beta_{let}]{} N_1 \equiv E[\mathsf{let}\ x\ \mathsf{be}\ R'\ \mathsf{in}\ P'] \ .$$

If there exists a node z_1 in $\mathcal{D}ag(M)$, 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 $M_1', M_1 \xrightarrow[name]{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow M_1'$, by doing a complete development of $\mathcal{F}$. We have: $M_1' \leq N_1$.

$n > 1$: Let $M \xrightarrow[name]{}^{n-1} M'$ and $M' \xrightarrow[name]{} M_1$. By the induction hypothesis, $\exists N_1 \in \Lambda_{\text{let}}, M'' \in \Lambda$,

$$N \xrightarrow[let]{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow N_1, M' \xrightarrow[name]{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow M'' \text{ and } M'' \leq N_1 \ .$$

From Lemma 2.3 $\exists M_2$,

$$M'' \xrightarrow[name]{}^{0/1} M_2 \text{ and } M_1 \xrightarrow[name]{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow M_2 \ .$$

If $M_2 \equiv M''$ we are done. Otherwise, by the induction hypothesis $\exists N_1' \in \Lambda_{\text{let}}, M_2' \in \Lambda$,

$$N_1 \xrightarrow[let]{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow N_1', M_2 \xrightarrow[name]{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow M_2' \text{ and } M_2' \leq N_1' \ .$$

□

Putting all the lemmas together we can prove that if a call-by-name interpreter stops in a lambda abstraction then there exists a reduction in the let-calculus to an answer.[‡]

Theorem 5.11
Given a program $M \in \Lambda$, $M \xmapsto[name]{}\!\!\!\!\!\rightarrow \lambda x.N \Longrightarrow M \xmapsto[let]{}\!\!\!\!\!\rightarrow A$.

Proof
Since $M \leq M$, from Lemma 5.10, there exists $M_1 \in \Lambda_{\mathrm{let}}, N_1 \in \Lambda$ such that:

$$M \xmapsto[let]{}\!\!\!\!\!\rightarrow M_1 \text{ and } \lambda x.N \xrightarrow[name]{}\!\!\!\!\!\rightarrow \lambda x.N_1$$

where $\lambda x.N_1 \leq M_1$. The result then follows from Lemma 5.9. □

We are now ready to show that the call-by-need parameter passing technique is a correct implementation of call-by-name.

Theorem 5.12
Given a program M, $\mathtt{Eval}_{\mathsf{name}}(M) = \mathsf{closure}$ if and only if $\mathtt{Eval}_{\mathsf{need}}(M) = \mathsf{closure}$.

Proof
The if-direction follows from Theorem 5.2. The other direction follows from Theorem 5.11 and the fact that in the call-by-need calculus the let-construct is syntactic sugar for function application. □

6 Fully lazy calculus

The call-by-need calculus captures the sharing of the evaluation of arguments. However, some implementations share even more computations than just those of arguments. For example, according to our calculus, the program

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

will evaluate the redex II twice. This is what happens also in the interpreter of Henderson and Morris (1976) and in the G-machine (Peyton Jones & Salkild, 1989). However, the redex II will be evaluated only once following the combinator machine of Turner (1979), supercombinator approach of Hughes (1982) and Wadsworth's interpreter (1971). We refer to these implementations as *fully lazy*.

Wadsworth (1971) was the first one to provide a fully lazy interpreter. He observed that the redex II should not be copied, since no occurrence of the bound variable w occurs in II. This can be captured by lifting the redex II as follows:

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

If we perform such a lifting, the redex II is only performed once:

$$(\lambda f.fI(fI))((\lambda z.(\lambda w.zw))(II)) \xrightarrow[assoc]{} (\lambda z.(\lambda f.fI(fI))(\lambda w.zw))(II) \xrightarrow[deref]{}$$
$$(\lambda z.(\lambda f.(\lambda w.zw)(fI))(\lambda w.zw))(II) \longrightarrow (\lambda z.(\lambda f.(\lambda w.zw)(fI))(\lambda w.zw))((\lambda z.I)I) \quad .$$

A redex like II is called a *maximal free expression* (mfe) of $(\lambda w.(II)w)$.

[‡] Indeed, the following stronger result also holds. The call-by-need reduction always involves lesser number of β-steps than the call-by-name reduction that it simulates.

Definition 6.1
A subterm N of a lambda abstraction M is a free expression of M if all free variables of N are free in M, and N is not a variable or a lambda abstraction. N is said to be a maximal free expression of M if M does not contain any other free expression that properly contains N.

Notation: $\vec{z}$ and $\vec{N}$ stand for $z_1, \cdots, z_n$ and $N_1, \cdots, N_n$, respectively. $\mathtt{mfe}(M, \vec{N})$ is true if $N_1, \cdots, N_n$ are mfe's of M. We use V_{mfe} to denote a value that does not contain mfe's.

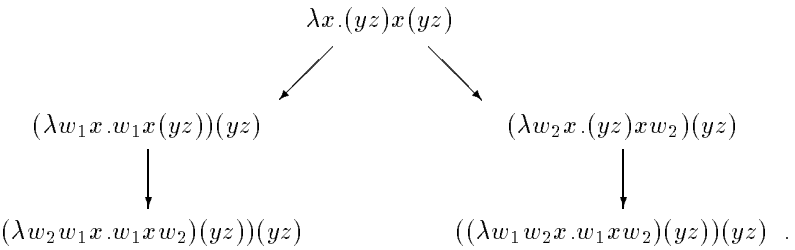
If we now restrict the axiom *deref* of the call-by-need calculus so that it only duplicates values that do not contain mfe's, we are guaranteed that no mfe is reduced twice. We call the new *deref* axiom $deref_{wad}$:

$$(\lambda x.E[x])V_{mfe} = (\lambda x.E[V_{mfe}])V_{mfe} \quad .$$

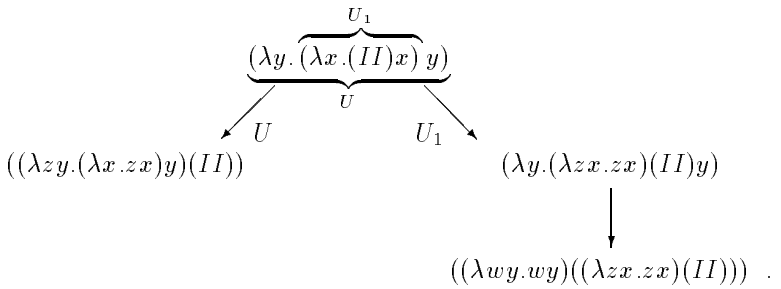
To accommodate this restriction, we must also introduce an axiom that permits the reduction of a value to a value without mfe's so that the $deref_{wad}$ axiom applies. This suggests the *mfe* axiom:

$$(\lambda y.C[\vec{N}]) = ((\lambda \vec{z} y.C[\vec{z}])\vec{N}) \qquad \mathtt{mfe}(\lambda y.C[\vec{N}], \vec{N}) \text{ and } \nexists N, \mathtt{mfe}(\lambda y.C[\vec{z}], N) \quad .$$

The proviso guarantees that all mfe's are extracted at once, which avoids the following non-confluence trap:



Unfortunately, the *mfe* axiom still interferes with itself:



Since the goal of lifting mfe's is to avoid some extra copying, it makes sense to perform redex U because the λy-abstraction can be put in a context that associates a name to it. However, that is not the case for the λx-abstraction. Therefore, there is no need of reducing redex U_1; no copying of the body $(II)x$ can be demanded at this point. This suggests the following axiom as a replacement of *mfe*:

$$(\lambda x.E[x])(\lambda y.C[\vec{N}]) = (\lambda x.E[x])((\lambda \vec{z} y.C[\vec{z}])\vec{N}) \qquad \mathtt{mfe}(\lambda y.C[\vec{N}], \vec{N}) \text{ and } \nexists N, \mathtt{mfe}(\lambda y.C[\vec{z}], N) \quad .$$

In other words, we allow lifting of mfe's only if there is a danger of duplication, and demand that mfe's are lifted to a well-defined point. We summarize the fully lazy calculus in the following definition.

Definition 6.2
[The Wadsworth's calculus λ_{wad}] The following axioms define Wadsworth's calculus:

$$\begin{array}{rcll}
(\lambda x.E[x])V_{mfe} & = & (\lambda x.E[V_{mfe}])V_{mfe} & deref_{wad} \\
(\lambda x.E[x])(\lambda y.C[\vec{N}]) & = & (\lambda x.E[x])((\lambda \vec{z} y.C[\vec{z}])\vec{N}) & mfe_{wad} \\
 & & \quad \texttt{mfe}(\lambda y.C[\vec{N}], \vec{N}) \text{ and} & \\
 & & \quad \nexists N, \texttt{mfe}(\lambda y.C[\vec{z}], N) & \\
(\lambda x.A)MN & = & (\lambda x.AN)M & lift \\
(\lambda x.E[x])((\lambda y.A)M) & = & (\lambda y.(\lambda x.E[x])A)M & assoc
\end{array}$$

$\xrightarrow[wad]{}$ and $\xrightarrow[wad]{}\!\!\!\!\!\rightarrow$ denote the one-step and multiple steps reductions, respectively.

Like the call-by-need calculus, Wadsworth's calculus still enjoys two key properties: there are no critical pairs, and it is left-linear. As such, λ_{wad} satisfies the Parallel Move Lemma and is confluent. Therefore, as in Section 4, we can define a standard reduction, written as $\xmapsto[wad]{}\!\!\!\!\!\rightarrow$, and prove that the standard reduction leads to an answer if there exists one. The standard reduction defines Wadsworth's interpreter.

It is interesting to analyse the effects of the call-by-need axioms on terms that do not contain mfe's. As shown next, no new mfe's are created by applying the *deref* axiom.

Lemma 6.3
If M does not contain mfe's and $M \xrightarrow[deref]{} N$ then N also does not contain mfe's.

Proof
Let $M \equiv C[(\lambda x.E[x])V] \xrightarrow[deref]{} N \equiv C[(\lambda x.E[V])V]$. M does not contain mfe's, therefore the value V can actually be denoted by V_{mfe}. Since values are not mfe's, by structural induction on the definition of evaluation context we can show that if $E[x]$ does not contain mfe's then the same holds for $E[V_{mfe}]$. The result then follows trivially. □

In contrast, the *assoc* and the *lift* axioms introduce new mfe's. Take the following reduction

$$M \equiv (\lambda x.x)((\lambda y.\lambda z.z)N) \xrightarrow[assoc]{} (\lambda y.(\lambda x.x)\lambda z.z)N \equiv M_1 \ , \tag{3}$$

the term $(\lambda x.x)(\lambda z.z)$ is a new mfe of $(\lambda y.(\lambda x.x)\lambda z.z)$. The same happens for the *lift* axiom:

$$M \equiv (\lambda x.\lambda z.z)N_1N_2 \xrightarrow[lift]{} (\lambda x.(\lambda z.z)N_2)N_1 \equiv M_1 \ , \tag{4}$$

$(\lambda z.z)N_2$ is a new mfe of $(\lambda x.(\lambda z.z)N_2)$. Note that the creation of new mfe's only happens when the *assoc* and *lift* redexes are of the form $(\lambda x.E[x])((\lambda y.V)M)$ and $(\lambda x.V)MN$, respectively.

Lemma 6.4
Let M be free of mfe's:
(i) if $M \xrightarrow[assoc]{} N$ and N contains a mfe then it must be that $M \equiv C[(\lambda x.E[x])((\lambda y.V)P)]$.
(ii) if $M \xrightarrow[lift]{} N$ and N contains a mfe then it must be that $M \equiv C[(\lambda x.V)PQ]$.

Proof

(i) Let us assume $M \equiv C[(\lambda x.E[x])((\lambda y.A)P)] \longrightarrow C[(\lambda y.(\lambda x.E[x])A)P] \equiv N$, with A not a value. For $(\lambda x.E[x])A$ to be a mfe of $(\lambda y.(\lambda x.E[x])A)$, the bound variable y cannot occur free in A. Since A is not a value, it means that A must be a mfe of $\lambda y.A$. Since M is free of mfe's we reached a contradiction. Therefore, A must be a value.
(ii) Same as the above.
□

The mfe's created by either *assoc* or *lift* are benign mfe's, that is, no need of lifting them will ever occur during the evaluation of a program. In fact, referring to equation (3), the evaluation of M_1 will dereference x and then stop. Analogously, referring to equation (4), the evaluation of M_1 must reduce N_2 to a value, and if that succeeds the evaluation will stop after a *deref* step. In other words, if the program does not contain any mfe's, no mfe_{wad} step occurs at run-time.

Theorem 6.5
If M does not contain mfe's then there is no N, $M \xmapsto[need]{} N$ and $N \equiv E[U]$, with U a mfe_{wad} redex.

Proof
Let us assume there exists a term N, $M \xmapsto[need]{} N$ and N contains a mfe_{wad} redex U. Since M does not contain mfe's it means that the mfe's lifted by redex U must have been created during the evaluation of M. That is, there must be terms M_1 and M_2 such that

$$M \xmapsto[need]{} M_1 \longmapsto M_2 \xmapsto[need]{} N \ ,$$

with M_2 the first term in the evaluation that contains a mfe that is later on lifted by redex U. By Lemma 6.3 the reduction from M_1 to M_2 must either be an *assoc* or a *lift* step. We analyse the two cases:

assoc: By the previous lemma, M_1 is $E[(\lambda x.E_1[x])((\lambda y.V)P)]$, and M_2 is $E[(\lambda y.(\lambda x.E_1[x])V)P]$. We show by structural induction on the definition of the evaluation context E that the lifting of the new mfe $(\lambda x.E_1[x])V$ will never be demanded.

- E **is** $[\ ]$**:** That is, M_2 is $(\lambda y.(\lambda x.E_1[x])V)P$. Trivial, since the λy-abstraction will never have a name associated to it.
- E **is** E_2Q **or** $(\lambda z.E_2)Q$**:** Follows from the induction hypothesis.
- E **is** $(\lambda z.E_2[z])E_3$**:** That is, $M_2 \equiv (\lambda z.E_2[z])E_3[((\lambda y.(\lambda x.E_1[x])V)P)]$. By induction hypothesis, $(\lambda x.E_1[x])V$ does not get lifted by evaluating $E_3[((\lambda y.(\lambda x.E_1[x])V)P)]$. Thus, the only way it might be lifted is if it

is demanded by the outside lambda, that is, by $(\lambda z.E_2[z])$. For that to happen, $E_3[((\lambda y.(\lambda x.E_1[x])V)P)]$ must first be evaluated to a value. During that evaluation the application $(\lambda x.E_1[x])V$ will have to disappear, and thus the possibility of lifting it vanishes.

lift: As in the previous case.

Since M does not contain mfe's and the new mfe's are not lifted out, it must be the case that such an N cannot be found. □

Corollary 6.6
If M does not contain mfe's and $M \xrightarrow[wad]{}{}^{n} N$ then $M \xrightarrow[need]{}{}^{n} N$.

From the above corollary laziness and fully laziness coincide for programs that do not contain mfe's. Moreover, the corollary suggests that we could lift all the mfe's of a program at compile-time and then use our call-by-need interpreter. The lifting of mfe's at compile-time is the essence of the sophisticated lambda lifting algorithms pioneered by John Hughes (1982). Since we cannot predict at compile time whether or not a lambda abstraction will be shared, we have to resort to the *mfe* axiom. It is strongly normalizing, which implies that we can mfe-normalize the program before evaluating. However, as described earlier, it is non-confluent. To encompass this problem the *mfe* rule is applied following a specific strategy, namely we apply it in an inside-out manner, starting from the innermost lambda of a term. For example, the reduction $(\lambda y.(\lambda x.(II)x)y \xrightarrow[mfe]{} (\lambda zy.(\lambda x.zx)y)(II)$ is disallowed because II should be lifted from the λx-abstraction first. If we denote by $\mathtt{MFE}(M)$ the final term obtained by applying the *mfe* rule in an inside-out manner, we conjecture the soundness and the completeness of the lifting process:

Conjecture 6.7
Given a program M, $\mathtt{Eval}_{\mathtt{need}}(M) = \mathtt{Eval}_{\mathtt{need}}(\mathtt{MFE}(M))$.

Unfortunately, applying the lifting of mfe's at compile time is not as efficient as Wadsworth's interpreter:

$$M \xrightarrow[wad]{}{}^{n} A \not\Longrightarrow \mathtt{MFE}(M) \xrightarrow[need]{}{}^{\leq n} A' \ .$$

And yet worse, lifting mfe's can even slow down a call-by-need interpreter. For example,

$$M \equiv (\lambda f.fI)(\lambda w.(II)w) \xrightarrow[need]{}{}^{4} A \ .$$

Whereas,

$$\mathtt{MFE}(M) \equiv (\lambda f.fI)((\lambda z.\lambda w.zw)(II)) \xrightarrow[need]{}{}^{6} A' \ .$$

We are currently in the process of investigating how different notions of mfe's impact efficiency. That is, how to best approximate Wadsworth's interpreter without too much run-time overhead.

7 Reasoning about a lenient language

The $\mathtt{Eval}_{\mathtt{need}}$ interpreter defines a possible implementation of non-strictness. In this section, we investigate another evaluator based on a notion of parallel reduction.

Instead of delaying the evaluation of the argument until its value is needed, the argument can be evaluated in parallel with the body of the function. For example, we could reduce $(\lambda f.(II)f)(II)$ to $(\lambda f.(\lambda z.I)If)((\lambda z.I)I)$ in one step. In other words, if we had processors waiting for work, we could execute any redex we want independently of the need. One extreme is to compute all redexes in a term. This is usually referred in the literature as the *Gross-Knuth* evaluation strategy, and we denote it by $(\xrightarrow[GK]{})$. Due to the Parallel Move Lemma of the call-by-need calculus, the *Gross-Knuth* evaluation strategy defines a function on terms.

Definition 7.1
$M \xrightarrow[GK]{} N$ if and only if $M \xrightarrow[cdv\ \mathcal{F}]{} N$, with $\mathcal{F}$ the set of all redexes in M. $\xrightarrow[GK]{*}$ is the transitive reflexive closure of $\xrightarrow[GK]{}$.

For simplicity let us assume $II \longrightarrow I$. The following reduction is then a complete development with respect to the underlined redexes in M,

$$M \equiv (\underline{\lambda}x.x(\underline{II}))(\lambda y.\underline{II}) \longrightarrow (\lambda x.(\lambda y.\underline{II})(\underline{II}))(\lambda y.\underline{II}) \longrightarrow (\lambda x.(\lambda y.I)(\underline{II}))(\lambda y.\underline{II})$$
$$\longrightarrow (\lambda x.(\lambda y.I)I)(\lambda y.\underline{II}) \longrightarrow (\lambda x.(\lambda y.I)I)(\lambda y.I) \quad .$$

Thus, we would say $M \xrightarrow[GK]{} (\lambda x.(\lambda y.I)I)(\lambda y.I)$.

Lemma 7.2
$\xrightarrow[GK]{*}$ is a cofinal strategy: for all terms M, N such that $M \xrightarrow[need]{*} N$ there exists M_1 such that $M \xrightarrow[GK]{*} M_1$ and $N \xrightarrow[need]{*} M_1$.

Proof
By induction on the number n of reduction steps of $M \xrightarrow[need]{*} N$.

$n = 1$: Directly from the Parallel Move Lemma.
$n > 1$: Let $M \longrightarrow^{n-1} M_1 \longrightarrow N$. By induction hypothesis, $\exists M_1'$, $M \xrightarrow[GK]{*} M_1'$ and $M_1 \xrightarrow[need]{*} M_1'$. Let us assume $M_1 \xrightarrow{U} N$. From the Parallel Move Lemma:

$$M_1' \xrightarrow[cdv\ \mathcal{F}_k]{} N' \text{ and } N \xrightarrow[need]{*} N' \quad ,$$

where $\mathcal{F}_k$ is the set of descendants of U with respect to the reduction $M_1 \xrightarrow[need]{*} M_1'$. If the size of $\mathcal{F}_k$ is zero, then the result holds trivially. Since $\mathcal{F}_k$ is a subset of all the redexes in M_1', the result follows again from the Parallel Move Lemma.

□

The Gross-Knuth strategy suggests a lockstep parallelism, that is, we first find all redexes in a program, we reduce them and then start over. However, real implementations allow more freedom than that. Even more, redexes inside a lambda are not reduced until the function is applied. For example, in the term

$$(\lambda x.\underbrace{(II)}_{U}(\lambda y.\underbrace{II}_{U_1}))M \quad ,$$

we would like to allow the reduction of U but not of U_1, because U_1 occurs inside a lambda abstraction which is not applied. We call this evaluation strategy *lenient*,

that is, the only control mechanism is lambda abstraction (this is the evaluation strategy of the language Id (Nikhil, 1991)). We first introduce the notion of *lenient evaluation context*:

$$E_{lenient} ::= [\,] \mid E_{lenient} M \mid M E_{lenient} \mid (\lambda x.E_{lenient})M \quad .$$

The lenient strategy is then defined as:

Definition 7.3
$M \xrightarrow[lenient]{} N$ if and only if M is not an answer and $M \xrightarrow[cdv\ \mathcal{F}]{} N$, with $\mathcal{F}$ a set of redexes U in M such that $M \equiv E_{lenient}[U]$.

Note that differently from the Gross-Knuth strategy we do not have to select all redexes in a term. For example,

$$(\lambda x.(II)w(II))(\lambda y.II) \xrightarrow[lenient]{} (\lambda x.Iw(II))(\lambda y.II) \quad ,$$

but

$$(\lambda x.(II)w(II))(\lambda y.II) \not\xrightarrow[GK]{} (\lambda x.Iw(II))(\lambda y.II) \quad .$$

The lenient strategy is non-deterministic but confluent.

Theorem 7.4
Given a program M, $M \xrightarrow[need]{}\!\!\!\!\!\rightarrow A$ if and only if there exists an answer A' such that $M \xrightarrow[lenient]{}\!\!\!\!\!\rightarrow A' \xrightarrow[need]{}\!\!\!\!\!\rightarrow A$.

Proof
The only if direction is trivial. The other direction follows from the cofinality of the Gross-Knuth strategy (Lemma 7.2) and the fact that if $M \xrightarrow[GK]{}\!\!\!\!\!\rightarrow A$ then there exists an answer A' such that $M \xrightarrow[lenient]{}\!\!\!\!\!\rightarrow A' \xrightarrow[need]{}\!\!\!\!\!\rightarrow A$.

□

By defining a lenient evaluator as:

$$\mathtt{Eval}_{\mathsf{lenient}}(M) = \mathsf{closure} \text{ if and only if } M \xrightarrow[lenient]{}\!\!\!\!\!\rightarrow A \quad ,$$

we obtain that the lenient strategy is another possible implementation of non-strictness.

Corollary 7.5
Given a program M, $\mathtt{Eval}_{\mathsf{need}}(M) = \mathsf{closure}$ if and only if $\mathtt{Eval}_{\mathsf{lenient}}(M) = \mathsf{closure}$.

The above corollary indicates that laziness should not be confused with non-strictness, as it is often done in the literature. Specifically, non-strictness is a property of a language's semantics while laziness is an implementation technique.

8 Extensions

Lazy implementations of the lambda calculus provide at least two pragmatic extensions of the pure theory. The first extension is a set of constructors for forming complex data values. The second important extension is a form for specifying mutually recursive computations (and values). We deal with each of these extensions in turn.

Lazy constructors Most non-strict functional languages provide lazy data constructors. For example, lazy cons only evaluates its arguments when there is a demand for their respective values (Friedman & Wise, 1976). The evaluation rules for cons can be inferred from the usual encoding of cons and the related destructure operations as functions (Barendregt, 1984): $\mathsf{cons} = (\lambda x_1 x_2 p.p x_1 x_2)$, $\mathsf{car} = \lambda p.p(\lambda x_1 x_2.x_1)$, and $\mathsf{cdr} = \lambda p.p(\lambda x_1 x_2.x_2)$.

Recursion A deficiency of the call-by-need calculus is its treatment of recursive or cyclic values. Following tradition, it relies on the Y combinator to implement recursion. In the absence of data constructors, this solution is not a problem. 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 a lazy evaluator will only allocate *one cell* and will represent M as a cyclic structure.

To cope with recursion, we transform Ariola and Klop's (1994; 1996) call-by-name calculus with cycles to a call-by-need calculus with cycles. The first step is to add a letrec construct of the form

$$\mathsf{letrec}\ x_1\ \mathsf{be}\ N_1, \cdots,\ x_n\ \mathsf{be}\ N_n\ \mathsf{in}\ M$$

to the syntax. No ordering among the bindings is assumed. The set of evaluation contexts is the natural adaptation of the set of evaluation contexts for the call-by-need calculus. The only difference is the presence of $D[x, x_n]$, which stands for a set of declarations containing bindings of the form "x be $E[x_1], x_1$ be $E[x_2], \cdots, x_{n-1}$ be $E[x_n]$". As shown in the following definition, the set of values V also contains a new constant $\bullet$, which arises due to cycles. Specifically, we consider a set of declarations of the shape $D[x, x]$ to be equivalent to $\bullet$. The constant $\bullet$ represents a program that provably diverges. It is not an answer and does not contain a redex.

Definition 8.1
[The Call-by-Need Letrec Calculus] The following clauses define the syntax and basic axioms of the call-by-need Letrec calculus.

Syntax:

Expressions (Λ):	M	::=	$x \mid \lambda x.M \mid MM \mid \mathsf{letrec}\ D\ \mathsf{in}\ M$
Declaration:	D	::=	$x_1\ \mathsf{be}\ M_1, \cdots, x_n\ \mathsf{be}\ M_n$
Values:	V	::=	$\lambda x.M \mid \bullet$
Answers:	A	::=	$V \mid \mathsf{letrec}\ D\ \mathsf{in}\ A$
Evaluation Contexts:	E	::=	$[\,] \mid EM \mid \mathsf{letrec}\ D\ \mathsf{in}\ E \mid$
			$\mathsf{letrec}\ D,\ x\ \mathsf{be}\ E\ \mathsf{in}\ E[x] \mid$
			$\mathsf{letrec}\ x_n\ \mathsf{be}\ E, D[x, x_n]\ \mathsf{in}\ E[x]$
Dependencies:	$D[x, x_n]$	::=	$x\ \mathsf{be}\ E[x_1], \cdots, x_{n-1}\ \mathsf{be}\ E[x_n],\ D$

Axioms:

β_{need} :
$(\lambda x.M)N = \mathsf{letrec}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M$
lift :
$(\mathsf{letrec}\ D\ \mathsf{in}\ A)N = \mathsf{letrec}\ D\ \mathsf{in}\ AN$
deref :
$\mathsf{letrec}\ D,\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[x] = \mathsf{letrec}\ D,\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[V]$
$deref_{env}$:
$\mathsf{letrec}\ x_n\ \mathsf{be}\ V,\ D[x, x_n]\ \mathsf{in}\ E[x] = \mathsf{letrec}\ x_n\ \mathsf{be}\ V,\ D[x, V]\ \mathsf{in}\ E[x]$
assoc
$\mathsf{letrec}\ D',\ x\ \mathsf{be}\ (\mathsf{letrec}\ D\ \mathsf{in}\ A)\ \mathsf{in}\ E[x] = \mathsf{letrec}\ D',\ D,\ x\ \mathsf{be}\ A\ \mathsf{in}\ E[x]$
$assoc_{env}$
$\mathsf{letrec}\ x_n\ \mathsf{be}\ (\mathsf{letrec}\ D\ \mathsf{in}\ A),\ D[x, x_n]\ \mathsf{in}\ E[x] = \mathsf{letrec}\ D,\ x_n\ \mathsf{be}\ A,\ D[x, x_n]\ \mathsf{in}\ E[x]$

By applying a similar reasoning as in Section 4 it is possible to show that the extended call-by-need calculus is an orthogonal system, and as such it is confluent and enjoys the Parallel Move Lemma. We are in the process of showing soundness and completeness of the above calculus with respect to Ariola and Klop's calculus (1994; 1996).

9 Applications

The call-by-need calculus has a number of potential applications. Its 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 is usually implemented using assignments. Crank (1990) briefly discusses a rewriting semantics of call-by-need based on Felleisen and Hieb's (1992) λ-calculus with assignment. We believe that the call-by-need calculus is the correct basis for proving this implementation technique correct with a simple bi-simulation theorem for the respective standard reductions.

Call-by-need and cps conversion Okasaki et al. (1994) 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 (Plotkin, 1975). Plotkin's proof techniques should immediately apply. Since this transformation appears to be used in several implementations of lazy languages [Odersky: personal communication, June 1994] 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, the idea of garbage collection can easily be expressed for the call-by-need calculus by adapting the garbage collection rule for reference cells of Felleisen and

Hieb (1992; 1990):

$$(\lambda x.N)M = N \qquad \text{if } x \notin FV(N) \ .$$

We expect that the work on garbage collection in functional languages by Morrisett et al. (1995) will apply to call-by-need languages and will strengthen the calculus and its utility for reasoning about the implementations of lazy languages.

Acknowledgement The first author wishes to thank Stefan Blom, Amr Sabry, Miley Semmelroth, J.W. Klop and H. Barendregt for numerous helpful discussions.

References

Abadi, M., Cardelli, L., Curien, P.-L., & Lévy, J.-J. (1991). Explicit substitutions. *Journal of Functional Programming*, **4**(1), 375–416.

Abramsky, S. (1990). The lazy lambda calculus. Turner, D. (ed), *Declarative Programming*. Addison-Wesley.

Ariola, Z. M., & Arvind. (1995). Properties of a first-order functional language with sharing. *Theoretical Computer Science*, **146**, 69–108.

Ariola, Z. M., & Klop, J. W. (1994). Cyclic lambda graph rewriting. *Pages 416–425 of: Proc. Ninth Symposium on Logic in Computer Science (LICS'94), Paris, France.*

Ariola, Z. M., & Klop, J. W. (1996). *Lambda calculus with explicit recursion.* Tech. rept. CIS-TR-96-04. Department of computer and information science, University of Oregon.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., & Wadler, P. (1995). The call-by-need lambda calculus. *Pages 233–246 of: Proc. ACM Conference on Principles of Programming Languages.*

Arvind, Kathail, V., & Pingali, K. (1984). Sharing of computation in functional language implementations. *Pages 5.1–5.12 of: Proc. International Workshop on High-Level Computer Architecture.*

Barendregt, H. P. (1984). *The lambda calculus: Its syntax and semantics.* North-Holland, Amsterdam.

Bloo, Roel, & Rose, Kristoffer Høgsbro. (1995). Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. *Pages 62–72 of: CSN '95 – Computer Science in the Netherlands.*

Crank, E., & Felleisen, M. (1990). Parameter-passing and the lambda calculus. *Pages 233–244 of: Proc. ACM Conference on Principles of Programming Languages.*

Felleisen, M., & Friedman, D. P. (1986). Control operators, the SECD-machine, and the lambda-calculus. *Pages 193–217 of: 3rd Working Conference on the Formal Description of Programming Concepts.*

Felleisen, M., & Friedman, D. P. (1989). A syntactic theory of sequential state. *Theoretical Computer Science*, **69**(3), 243–287.

Felleisen, M., & Hieb, R. (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, **102**, 235–271.

Field, J. (1990). On laziness and optimality in lambda interpreters: Tools for specification and analysis. *Pages 1–15 of: Proc. ACM Conference on Principles of Programming Languages, San Francisco.*

Friedman, D. P., & Wise, D. S. (1976). *Cons* should not evaluate its arguments. *Pages 257–284 of: Proc. International Conference on Automata, Languages and Programming.*

Gonthier, G., Abadi, M., & Lévy, J.-J. (1992). The geometry of optimal lambda reduction. *Pages 15–26 of: Proc. ACM Conference on Principles of Programming Languages.*

Henderson, P., & Morris, J.H. (1976). A lazy evaluator. *Pages 95–103 of: Proc. ACM Conference on Principles of Programming Languages.*

Huet, G., & Lévy, L.-J. (1991). Computations in orthogonal rewriting systems 1 and 2. *Pages 395–443 of: Computational logic. Essays in Honor of Alan Robinson. Ed. J.-L. Lassez & G.D. Plotkin.*

Huet, G.P. (1980). Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM*, **27**(4), 797–821.

Hughes, R. J. M. (1982). Super-combinators: a new implementation method for applicative languages. *Pages 1–10 of: Proc. ACM Symposium on Lisp and Functional Programming.*

Kathail, V. K. (1990). *Optimal interpreters for lambda-calculus based funtional languages.* Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, MIT.

Klop, J. W. (1992). Term rewriting systems. *Pages 1–116 of:* Abramsky, S., Gabbay, D., & Maibaum, T. (eds), *Handbook of Logic in Computer Science*, vol. II. Oxford University Press.

Lamping, J. (1990). An algorithm for optimal lambda calculus reduction. *Pages 16–30 of: Proc. ACM Conference on Principles of Programming Languages, San Francisco.*

Launchbury, J. (1993). A natural semantics for lazy evaluation. *Pages 144–154 of: Proc. ACM Conference on Principles of Programming Languages.*

Lescanne, P. (1994). From $\lambda\sigma$ to λv a journey through calculi of explicit substitutions. *Pages 60–69 of: Proc. ACM Conference on Principles of Programming Languages, Portland.*

Maraist, J., Odersky, M., & Wadler, P. (1994). *The call-by-need lambda calculus (unabridged).* Tech. rept. 28/94. Universität Karlsruhe, Fakultät für Informatik.

Maranget, L. (1991). Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. *Pages 255–269 of: Proc. ACM Conference on Principles of Programming Languages, Orlando, Florida.*

Morrisett, G., Felleisen, M., & Harper, R. (1995). Modeling memory management. *Pages 66–77 of: Proc. ACM Conference on Functional Programming Languages and Computer Architecture.*

Nikhil, R.S. (1991). *Id (version 90.1) reference manual.* Tech. rept. CSG Memo 284-2. MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.

Okasaki, C., Lee, P., & Tarditi, D. (1994). Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, **7**(1), 57–81.

Ong, C.-H.L. (1988). *The lazy calculus: An investigation in the foundations of functional programming.* Ph.D. thesis, Imperial College, London.

Peyton Jones, S. L. (1987). *The implementation of functional programming languages.* International Series in Computer Science. Prentice Hall.

Peyton Jones, S. L., & Salkild, J. (1989). The spineless tagless g-machine. *Pages 184–201 of: Proc. ACM Conference on Functional Programming Languages and Computer Architecture.*

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

Purushothaman, S., & Seaman, J. (1992). An adequate operational semantics of sharing in lazy evaluation. *Pages 435–450 of: Proc. 4th European Symposium on Programming, Springer Verlag LNCS 582.*

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

van Oostrom, V. (1994). *Confluence for Abstract and Higher-Order Rewriting.* Ph.D. thesis, Vrije Universiteit.

Wadsworth, C. (1971). *Semantics and pragmatics of the lambda-calculus.* Ph.D. thesis, University of Oxford.

Yoshida, N. (1993). Optimal reduction in weak-λ-calculus with shared environments. *Pages 243–252 of: Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Copenhagen.*