CDMTCS
Research
Report
Series

Tracing Lazy Functional Languages

Jeremy Gibbons and
Keith Wansbrough
Department of Computer Science
University of Auckland

CDMTCS-006
August 1995

Centre for Discrete Mathematics and
Theoretical Computer Science

Tracing Lazy Functional Languages

Jeremy Gibbons and Keith Wansbrough

Department of Computer Science
University of Auckland
Private Bag 92019
Auckland, New Zealand

{jeremy,kwan01}@cs.auckland.ac.nz

Abstract

We argue that Ariola and Felleisen's and Maraist, Odersky and Wadler's axiomatization of the call-by-need lambda calculus forms a suitable formal basis for tracing evaluation in lazy functional languages. In particular, it allows a one-dimensional textual representation of terms, rather than requiring a two-dimensional graphical representation using arrows. We describe a program `LetTrace`*, implemented in Gofer and tracing lazy evaluation of a subset of Gofer.*

Keywords Functional programming, tracing, call-by-need, lambda calculus, lazy evaluation.

1 Introduction

One major advantage of pure, and especially lazy, functional languages over more conventional imperative languages is in not having to directly and completely specify the order of execution of a program. Leaving execution order unspecified allows the compiler or interpreter to perform transformations on the code, changing the order of execution and perhaps even executing parts of the program in parallel.

In *Proceedings of CATS'96 (Computing: The Australasian Theory Symposium)*, Melbourne, Australia, January 1996, and *Massey Functional Programming Workshop*, Palmerston North, New Zealand, February 1996.

However, when it comes to debugging a program, this feature turns into a disadvantage. In a more conventional language, one can 'trace' execution by inserting write statements in interesting places in order to monitor what is happening. More sophisticated tracing systems provide step-by-step execution of statements together with an animation consisting of the program text and a pointer to the 'current statement'. This approach does not work in a pure functional language, because there is no direct connection between the order of a program's text and the order of that program's execution. Indeed, in a lazy functional language, the same program with the same input may display different execution orders depending on what is done with the output.

Because pure functional languages are free of side-effects, the programmer cannot use the write-statement approach. Augustsson and Johnsson [5] describe the addition to a pure lazy functional language of an impure primitive trace with two arguments, which returned the first argument as result but printed the value of the second as a side-effect. They observe that 'it generally turned out to be very difficult to decipher the output from this, since quite often (due to lazy evaluation) the evaluation by trace of its arguments caused other instances of trace to be evaluated. The result was a mish-mash of output from different instances of trace.'

The essential problem is that running a purely functional program does not consist of step-by-step execution of statements, as in a conventional language, but rather consists of reducing an expression to normal form. The result of tracing, therefore, should not be the sequence of statements executed; rather, it should be the sequence of intermediate terms through which the expression is transformed.

Producing such a sequence of intermediate terms entails having a calculus rich enough to model these terms and their reductions. Plotkin [18] shows that the call-by-value lambda calculus, in which an argument to a function is evaluated before being substituted into the function body, is a good model for a call-by-value functional language, and that the call-by-name lambda calculus, in which the argument is substituted before it is evaluated, is a good model of a call-by-name language. However, there is more to a lazy functional language than just call-by-name parameter passing; in particular, multiple copies of an argument are shared, and the argument is evaluated at most once and subsequently reused if necessary. Although the call-by-name lambda calculus accurately models the semantics (the relationship between a program and its final value) of lazy evaluation, it does not accurately model the pragmatics (the evaluation process by which this relationship is achieved).

Ariola, Felleisen, Maraist, Odersky and Wadler [2, 3, 15] have developed a call-by-need lambda calculus that does accurately model the sharing of arguments in lazy evaluation; Wadler reported on the work at CATS'94 [24]. We claim that this call-by-need lambda calculus makes a good vehicle for tracing evaluation in a lazy functional language. In particular, this calculus provides a one-dimensional,

Syntactic domains:

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

Reduction rules:

(Let-I)	$(\lambda x.M)\, N \xrightarrow{\text{NAME}} \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M$
(Let-E)	$\mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M \xrightarrow{\text{NAME}} M[x := N]$

Figure 1: The call-by-name calculus

textual representation of terms, rather than requiring a two-dimensional graphical representation.

The rest of this paper is structured as follows. In Section 2 we review the call-by-name lambda calculus and introduce the call-by-need calculus; this material is not new [2, 3, 6, 8, 15, 24]. In Section 3 we describe the implementation of a program to perform evaluation in the call-by-need lambda calculus. Finally, in Section 4 we present some possible future extensions and discuss related work.

2 Lambda calculi

The call-by-need lambda calculus is most naturally presented by augmenting the standard lambda calculus syntax of variables, lambda abstractions and function applications with an extra piece of syntax, the let-expression. In order to make clear the parallels with the call-by-name calculus, we will first review the latter also augmented with let-expressions.

We write $fv(M)$ for the set of free variables of a term M. We write $M[x := N]$ for the substitution of term N for all free occurrences of variable x in term M. We assume Barendregt's *variable convention* [6]: in a term, all the bound variables are chosen to be different from the free variables.

In an application $M\, N$, we call M the *function part* and N the *application part*. In a let-expression let x be M in N, we call x the *variable*, M the *definition* and N the *body*.

2.1 The call-by-name calculus

The syntactic domains and reduction rules of the call-by-name calculus are given in Figure 1. The set of variables is countably infinite. Values are those terms that cannot be further reduced. Rule Let-I introduces a let-expression, and rule Let-E

$$\begin{array}{lcl}
 & & (\lambda x.x\,x)\,((\lambda y.y)\,(\lambda z.z)) \\
(\text{Let-I}) & \xrightarrow{\text{NAME}} & \mathsf{let}\ x\ \mathsf{be}\ ((\lambda y.y)\,(\lambda z.z))\ \mathsf{in}\ x\,x \\
(\text{Let-E}) & \xrightarrow{\text{NAME}} & ((\lambda y.y)\,(\lambda z.z))\,((\lambda y.y)\,(\lambda z.z)) \\
(\text{Let-I}) & \xrightarrow{\text{NAME}} & (\mathsf{let}\ y\ \mathsf{be}\ \lambda z.z\ \mathsf{in}\ y)\,((\lambda y.y)\,(\lambda z.z)) \\
(\text{Let-E}) & \xrightarrow{\text{NAME}} & (\lambda z.z)\,((\lambda y.y)\,(\lambda z.z)) \\
(\text{Let-I}) & \xrightarrow{\text{NAME}} & \mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\,(\lambda z.z)\ \mathsf{in}\ z \\
(\text{Let-E}) & \xrightarrow{\text{NAME}} & (\lambda y.y)\,(\lambda z.z) \\
(\text{Let-I}) & \xrightarrow{\text{NAME}} & \mathsf{let}\ y\ \mathsf{be}\ \lambda z.z\ \mathsf{in}\ y \\
(\text{Let-E}) & \xrightarrow{\text{NAME}} & \lambda z.z
\end{array}$$

Figure 2: A call-by-name reduction sequence

eliminates it in favour of substitution. Note that a let-expression is here essentially just a stepping stone between an application and the corresponding substitution.

A *context* is just a term with a single *hole*—denoted $[\,]$—in place of one of the subterms. In the case of our lambda calculus with let-expressions, the contexts are defined by the grammar

$$\text{Contexts}\quad C ::= [\,] \mid \lambda x.C \mid C\,M \mid M\,C \mid \mathsf{let}\ x\ \mathsf{be}\ C\ \mathsf{in}\ M \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ C$$

We write $C[M]$ for the context $C[\,]$ with the hole filled by the term M; we call the whole thing a *filled context*. (Note that a variable free in M may become bound in $C[M]$.) A filled context may be thought of as a term with one of the subterms 'marked'. We say term N *corresponds to* filled context $C[M]$, and write $N \approx C[M]$, if N is the term obtained by forgetting the mark in $C[M]$—that is, if replacing the hole in the context $C[\,]$ by the term M yields N.

We write $\xrightarrow{\text{NAME}}$ for the compatible closure of $\xrightarrow{\text{NAME}}$ (that is, the least relation $\xrightarrow{\text{NAME}}$ that includes the original relation $\xrightarrow{\text{NAME}}$ and is closed under the implication $M \xrightarrow{\text{NAME}} N \Rightarrow C[M] \xrightarrow{\text{NAME}} C[N]$).

Figure 2 gives an example reduction sequence in the call-by-name calculus. Notice that the argument $(\lambda y.y)\,(\lambda z.z)$ gets reduced twice.

2.2 Standard reduction

The reduction rule $\xrightarrow{\text{NAME}}$ is not deterministic, since (for example) the term

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

reduces under $\xrightarrow{\text{NAME}}$ to both

$$(\mathsf{let}\ y\ \mathsf{be}\ \lambda z.z\ \mathsf{in}\ y)\,((\lambda y.y)\,(\lambda z.z))$$

and

$$((\lambda y.y)\,(\lambda z.z))\,(\mathsf{let}\ y\ \mathsf{be}\ \lambda z.z\ \mathsf{in}\ y)$$

Although the reduction rule $\xrightarrow{\text{NAME}}$ is confluent—different orders of reduction cannot yield different normal forms—there are terms with both finite and infinite reduction sequences. Thus, $\xrightarrow{\text{NAME}}$ does not provide an accurate model of deterministic program execution.

In order to provide such a model, we define a deterministic restriction of $\xrightarrow{\text{NAME}}$, called *standard reduction* and written $\overset{\text{NAME}}{\longmapsto}$. A convenient way of formulating standard reduction is to use *evaluation contexts* [8], a subset of the contexts of the calculus.

The idea behind evaluation contexts is to relate a term N to one or more filled contexts $C_i[M_i]$ (that is, $N \approx C_i[M_i]$) in such a way that at most one M_i is reducible under $\xrightarrow{\text{NAME}}$; standard reduction then reduces this particular subterm of N.

For the call-by-name lambda calculus with let-expressions, the evaluation contexts are defined by the grammar

$$\text{ECONTEXTS} \quad E ::= [\,] \mid E\,M$$

We can rephrase this definition in terms of the question, "Which subterms of a term can I mark to get a filled evaluation context?" The answer to the question defines evaluation contexts:

- you can always mark the whole term;
- if the term is an application $M\,N$, you can mark in it anything you could mark in the function part M alone.

We say that term M *standard reduces* to term M', written $M \overset{\text{NAME}}{\longmapsto} M'$, iff $M \approx E[N]$, $M' \approx E[N']$ and $N \xrightarrow{\text{NAME}} N'$. It is not hard to prove the *Unique Evaluation Context Lemma* for the call-by-name calculus:

Lemma 1 *For a given term N, there is at most one filled evaluation context $C[M]$ such that $N \approx C[M]$ and M is reducible under $\xrightarrow{\text{NAME}}$.*

Hence, a term has at most one standard redex, and the standard reduction $\overset{\text{NAME}}{\longmapsto}$ is a partial function.

Closed terms in the calculus correspond to programs, and execution of a program corresponds to repeatedly reducing the standard redex until the whole term becomes irreducible (has no standard redex), at which point it will be a value.

$$
\begin{array}{ll}
(\textsc{Let-I}) & (\lambda x.M)\,N \xrightarrow{\text{NEED}} \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M \\
(\textsc{Let-V}) & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ C[x] \xrightarrow{\text{NEED}} \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ C[V] \\
(\textsc{Let-A}) & \mathsf{let}\ y\ \mathsf{be}\ (\mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ M)\ \mathsf{in}\ N \xrightarrow{\text{NEED}} \mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ (\mathsf{let}\ y\ \mathsf{be}\ M\ \mathsf{in}\ N) \\
(\textsc{Let-C}) & (\mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ M)\,N \xrightarrow{\text{NEED}} \mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ (M\,N) \\
(\textsc{Let-G}) & \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N \xrightarrow{\text{NEED}} N \quad \mathsf{if}\ x \neq fv(N)
\end{array}
$$

Figure 3: Call-by-need reduction rules

2.3 The call-by-need calculus

As we have seen, reduction in the call-by-name calculus does not correspond to lazy evaluation, because the argument to a function may get reduced more than once. The call-by-need[1] calculus [2, 3, 15] resolves this discrepancy.

The four syntactic domains Variables, Values, Terms and Contexts of the call-by-need calculus are the same as for call-by-name. The reduction rules, given in Figure 3, are however a little more complex.

- The first rule, Let-I (for 'let introduction'), is the same as for the call-by-name calculus (indeed, we presented the call-by-name calculus in an unconventional way just to make this so).

- The second rule, Let-V, corresponds to the Let-E rule of the call-by-name calculus. However, it only applies when the definition is (or has been reduced to) a value—hence the name. In this way, an argument is not substituted into a function body until it has been reduced to a value, and hence is reduced at most once. Note that this rule replaces only one occurrence of the variable; in general, replacing all occurrences will involve several applications of the rule, and then an application of the Let-G rule (see below) to remove the now-redundant let binding. Maraist *et al.* [14] give a rule closer to the call-by-name Let-E rule, replacing all free occurrences of the variable and removing the let binding all at once.

- The third rule, Let-A, turns left-nested let bindings into right-nested ones. To see why it is needed, consider for example the term

$$(\lambda f.f\,I\,(f\,I))\,((\lambda z.\lambda w.z\,w)\,(I\,I))$$

[1]It would be nice to call it the 'lazy lambda calculus', but unfortunately that name has already been taken by Abramsky and Ong [1]—ironically, for a call-by-name rather than a call-by-need calculus.

(writing I for $\lambda x.x$). After two applications of LET-I this reduces to

$$\mathsf{let}\ f\ \mathsf{be}\ (\mathsf{let}\ z\ \mathsf{be}\ I\,I\ \mathsf{in}\ \lambda w.z\,w)\ \mathsf{in}\ f\,I\,(f\,I)$$

We cannot substitute the definition of f into the body of the expression, because that would duplicate the unreduced $I\,I$ in the definition of z. Nor can we reduce the $I\,I$, because we do not know yet that we need it. The solution is to reassociate the bindings using the LET-A rule (hence the name), yielding

$$\mathsf{let}\ z\ \mathsf{be}\ I\,I\ \mathsf{in}\ \mathsf{let}\ f\ \mathsf{be}\ \lambda w.z\,w\ \mathsf{in}\ f\,I\,(f\,I)$$

Now the redex $I\,I$ will be reduced just once, despite f appearing twice in the result.

- The fourth rule, LET-C, is needed for a similar reason. Consider for example the term

  $$(\mathsf{let}\ z\ \mathsf{be}\ I\,I\ \mathsf{in}\ \lambda w.z\,w)\,I$$

  We want to associate the abstraction $\lambda w.z\,w$ with its argument I, but the two are not adjacent. The solution in this case is to allow let bindings to commute (hence the name) with application, giving

  $$\mathsf{let}\ z\ \mathsf{be}\ I\,I\ \mathsf{in}\ (\lambda w.z\,w)\,I$$

- The final rule, LET-G (for 'garbage collection'), discards redundant let bindings.

The grammar of contexts for the call-by-need calculus is the same as for the call-by-name calculus; however, the *evaluation* contexts are different. As with the call-by-name calculus, the evaluation contexts include the whole term, and the function part of an application:

$$E, E' ::= [\,] \mid E\,M$$

Function bodies should be reduced before arguments are substituted, so also the body of a let-expression is an evaluation context:

$$E, E' ::= \cdots \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ E$$

Finally, if the term is a let-expression whose body is an evaluation context filled with the defined variable, then the definition of the variable is needed for further progress, and so must be reduced:

$$E, E' ::= \cdots \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ E'[x]$$

Thus, we define the evaluation contexts for the call-by-need calculus by

$$
\begin{array}{ll}
(\textsc{Let-I}) & (\lambda x.M)\,N \xrightarrow{\text{NEED-S}} \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M \\
(\textsc{Let-V}) & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[x] \xrightarrow{\text{NEED-S}} \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[V] \\
(\textsc{Let-A}) & \mathsf{let}\ y\ \mathsf{be}\ (\mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ A)\ \mathsf{in}\ E[y] \\
 & \quad \xrightarrow{\text{NEED-S}} \mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ (\mathsf{let}\ y\ \mathsf{be}\ A\ \mathsf{in}\ E[y]) \\
(\textsc{Let-C}) & (\mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ A)\,N \xrightarrow{\text{NEED-S}} \mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ A\,N
\end{array}
$$

Figure 4: Standard call-by-need reduction

ECONTEXTS $E, E' ::= [\,] \mid E\,M \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ E \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ E'[x]$

Again, we can rephrase this definition in terms of the question, "Which subterms of a term can I mark to get a filled evaluation context, in the call-by-need calculus?" The answer, defining evaluation contexts, is:

- you can always mark the whole term;
- if the term is an application $M\,N$, you can mark in it anything you could mark in the function part M alone;
- if the term is a let-expression $\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N$, you can mark in it anything you could mark in the body N alone;
- if the term is a let-expression $\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N$, and you could mark an occurrence of the variable x in the body N alone, then you can mark in the whole anything you could mark in the definition M alone.

Unfortunately, this definition of evaluation contexts does not provide a Unique Evaluation Context Lemma for the call-by-need calculus. For example, in the term

$$\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ W\ \mathsf{in}\ x\,y$$

both let-expressions are markable and reducible under $\xrightarrow{\text{NEED}}$; we could substitute either x or y first.

We resolve this by directly defining a deterministic restriction of $\xrightarrow{\text{NEED}}$. We introduce a new syntactic category, ANSWERS:

ANSWERS $A ::= V \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ A$

Thus, an answer is a value (that is, a variable or an abstraction) wrapped up in zero or more let bindings. We then define the restriction $\xrightarrow{\text{NEED-S}}$ of $\xrightarrow{\text{NEED}}$, as in Figure 4. Now the rules LET-V and LET-A only apply if an occurrence of the variable is markable in the body (and so the definition of the variable is needed in order to

Syntactic domains:

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

Reduction rules:

δ-V $\quad p\, V \xrightarrow{\text{NEED-S}} \delta(p, V) \quad$ if $\delta(p, V)$ defined

δ-A $\quad p\,(\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N) \xrightarrow{\text{NEED-S}} \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ p\, N$

Figure 5: Adding constructors and primitive operators

make progress), and Let-A only applies if further the definition of y is an answer; the Let-C rule only applies if the body of the let-expression is an answer.

It is now possible—but tedious—to prove the Unique Evaluation Context Lemma for the call-by-need calculus [3, Lemma 4.2]:

Lemma 2 *For a given term N, there is at most one filled evaluation context $C[M]$ such that $N \approx C[M]$ and M is reducible under $\xrightarrow{\text{NEED-S}}$.*

Hence, standard reduction $\overset{\text{NEED}}{\longmapsto}$, the closure of the relation $\xrightarrow{\text{NEED-S}}$ under the implication $M \overset{\text{NEED}}{\longmapsto} N \Rightarrow E[M] \overset{\text{NEED}}{\longmapsto} E[N]$, is a partial function.

2.4 Constructors and primitive operators

Most 'real' functional languages augment the pure lambda calculus with constructors (for example, numbers and pairs) and primitives (for example, addition and projections). Of course, these features can be simulated in the pure lambda calculus, but providing them directly improves clarity and efficiency.

The changes to the calculus are shown in Figure 5. There are two new syntactic domains, Operators p and Constructors k^n of arity $n \geq 0$. Operators are values[2], and a constructor k^n together with n values $V_1, \ldots, V_n$ is itself a value. (The components of a constructor must be values, in order that copying the whole construct is safe. A constructor k^1 can be applied to a non-value M using a let-expression: $\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ k^1\, x$.) There are two new reduction rules. Rule δ-V applies a primitive operator, and is defined in terms of partial function δ from

[2]Maraist [13] observes that it is insufficient to have operators as terms, as in [3, 15], because then, for example, $\mathsf{let}\ x\ \mathsf{be}\ p\ \mathsf{in}\ x$ is irreducible.

operators and values to terms; this function may be arbitrary, except that it may not 'look inside' lambda abstractions (see [3] for more details). Note that this rule makes operators unary and strict; multiary operators can be simulated using currying. The second rule, δ-A, allows a let-expression as an argument to an operator to be pulled outside the application of that operator.

Notice that the δ-V rule only applies when the function δ is defined. For example, consider the calculus augmented with the nullary constructors *True* and *False* and the operator *Not*, with δ given by

$$\begin{aligned} \delta(\mathit{Not}, \mathit{True}) &= \mathit{False} \\ \delta(\mathit{Not}, \mathit{False}) &= \mathit{True} \end{aligned}$$

and undefined elsewhere. In the term

$$\mathsf{let}\ x\ \mathsf{be}\ \mathit{True}\ \mathsf{in}\ \mathit{Not}\, x$$

the subterm $\mathit{Not}\, x$ is markable, but should not be reducible (because $\delta(\mathit{Not}, x)$ is undefined). Instead, we need to add $p\,E$ as another kind of evaluation context[3]; then variable x is markable in $\mathit{Not}\, x$, so we may substitute its definition *True* as expected:

$$\begin{array}{lll} & & \mathsf{let}\ x\ \mathsf{be}\ \mathit{True}\ \mathsf{in}\ \mathit{Not}\, x \\ (\textsc{Let}\text{-V}) & \overset{\text{NEED}}{\longmapsto} & \mathsf{let}\ x\ \mathsf{be}\ \mathit{True}\ \mathsf{in}\ \mathit{Not}\ \mathit{True} \\ (\textsc{Let}\text{-G}) & \overset{\text{NEED}}{\longmapsto} & \mathit{Not}\ \mathit{True} \\ (\delta\text{-V}) & \overset{\text{NEED}}{\longmapsto} & \mathit{False} \end{array}$$

We can extend the calculus to include integers and addition by defining the constructors Delta_0, Delta_1, $\mathit{Delta}_2, \ldots$, the operators *Add* and Plus_0, Plus_1, $\mathit{Plus}_2, \ldots$, and the primitive evaluation rule

$$\begin{aligned} \delta(\mathit{Add}, \mathit{Delta}_i) &= \mathit{Plus}_i \\ \delta(\mathit{Plus}_i, \mathit{Delta}_j) &= \mathit{Delta}_{i+j} \end{aligned}$$

Then, for example, we have the reduction sequence in Figure 6. Notice that the $1+1$ gets evaluated only once; only two additions are performed overall.

3 LetTrace

We have implemented the call-by-need lambda calculus as a Gofer program called `LetTrace`. This program takes a term in the lambda calculus and produces a trace of its evaluation. The intention is that it be used to produce reduction sequences in

[3]This extra kind of evaluation context is missing from [3], and so in that paper the term let x be *True* in $\mathit{Not}\, x$ is irreducible.

$$
\begin{array}{lll}
 & & \textsf{let}\ x\ \textsf{be}\ (\mathit{Add}\,\mathit{Delta}_1)\,\mathit{Delta}_1\ \textsf{in}\ (\mathit{Add}\,x)\,x \\
(\delta\text{-V}) & \xrightarrow{\text{NEED}} & \textsf{let}\ x\ \textsf{be}\ \mathit{Plus}_1\,\mathit{Delta}_1\ \textsf{in}\ (\mathit{Add}\,x)\,x \\
(\delta\text{-V}) & \xrightarrow{\text{NEED}} & \textsf{let}\ x\ \textsf{be}\ \mathit{Delta}_2\ \textsf{in}\ (\mathit{Add}\,x)\,x \\
(\text{Let-V}) & \xrightarrow{\text{NEED}} & \textsf{let}\ x\ \textsf{be}\ \mathit{Delta}_2\ \textsf{in}\ (\mathit{Add}\,\mathit{Delta}_2)\,x \\
(\delta\text{-V}) & \xrightarrow{\text{NEED}} & \textsf{let}\ x\ \textsf{be}\ \mathit{Delta}_2\ \textsf{in}\ \mathit{Plus}_2\,x \\
(\text{Let-V}) & \xrightarrow{\text{NEED}} & \textsf{let}\ x\ \textsf{be}\ \mathit{Delta}_2\ \textsf{in}\ \mathit{Plus}_2\,\mathit{Delta}_2 \\
(\text{Let-G}) & \xrightarrow{\text{NEED}} & \mathit{Plus}_2\,\mathit{Delta}_2 \\
(\delta\text{-V}) & \xrightarrow{\text{NEED}} & \mathit{Delta}_4
\end{array}
$$

Figure 6: A reduction sequence with constructors and operators

the style of Bird and Wadler [7] to help explain lazy evaluation. Actually, we want to improve on Bird and Wadler; when it comes to sharing (for example, [7, p143]), they resort to drawing arrows, whereas we may use let-expressions. For example, `LetTrace` produces the trace in Figure 7 from the expression

$$(\lambda x.((\mathit{Add}\,x)\,x))\,((\mathit{Add}\,\mathit{Delta}_1)\,\mathit{Delta}_1)$$

whereas Bird and Wadler would have to use a series of diagrams with arrows, such as the diagram in Figure 8.

In this section we briefly describe the most interesting parts of the implementation. For further details, see [25].

3.1 Renaming and substitution

The first problem we encountered is that Barendregt's variable convention is an unrealistic assumption in practice. For starters, we cannot assume that a user's program will satisfy it. More seriously, literal application of the reduction rules can break the convention. For example, the term $(\lambda y.(\lambda x.y\,x))\,(\lambda x.z)$, which satisfies the variable convention, standard reduces in two steps to

$$\textsf{let}\ y\ \textsf{be}\ \lambda x.z\ \textsf{in}\ \lambda x.(\lambda x.z)\,x$$

in which the subterm $(\lambda x.z)\,x$ contains x both free and bound.

One solution would be to post-process the result of each reduction step, renaming variables if necessary to re-establish the variable convention. Another solution—the one we adopted in `LetTrace`—is to abandon the convention, and rephrase the reduction rules more carefully to avoid inadvertent variable capture. Our rephrased rules are shown in Figure 9. The Let-I and Let-G rules remain the same, as they

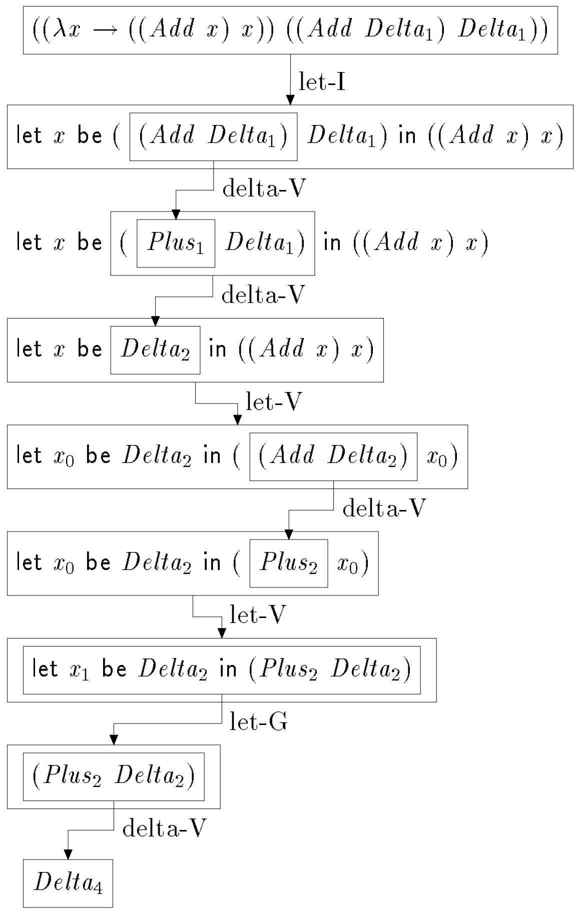
((λx → ((Add x) x)) ((Add Delta1) Delta1))
let-I
let x be ((Add Delta1) Delta1) in ((Add x) x)
delta-V
let x be (Plus1 Delta1) in ((Add x) x)
delta-V
let x be Delta2 in ((Add x) x)
let-V
let x0 be Delta2 in ((Add Delta2) x0)
delta-V
let x0 be Delta2 in (Plus2 x0)
let-V
let x1 be Delta2 in (Plus2 Delta2)
let-G
(Plus2 Delta2)
delta-V
Delta4

Figure 7: An example trace

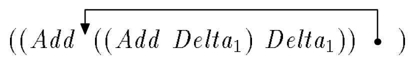

Figure 8: Bird and Wadler's representation of shared expressions

(Let-I)	$(\lambda x.M)\,N \xrightarrow{\text{NEED}} \textsf{let } x \textsf{ be } N \textsf{ in } M$
(Let-V)	$\textsf{let } x \textsf{ be } V \textsf{ in } C[x] \xrightarrow{\text{NEED}} \textsf{let } x' \textsf{ be } V \textsf{ in } C[x := x'][V]$
(Let-A)	$\textsf{let } y \textsf{ be } (\textsf{let } x \textsf{ be } L \textsf{ in } M) \textsf{ in } N$
	$\xrightarrow{\text{NEED}} \textsf{let } x' \textsf{ be } L \textsf{ in } (\textsf{let } y \textsf{ be } M[x := x'] \textsf{ in } N)$
(Let-C)	$(\textsf{let } x \textsf{ be } L \textsf{ in } M)\,N \xrightarrow{\text{NEED}} \textsf{let } x' \textsf{ be } L \textsf{ in } (M[x := x']\,N)$
(Let-G)	$\textsf{let } x \textsf{ be } M \textsf{ in } N \xrightarrow{\text{NEED}} N \quad \text{if } x \neq fv(N)$
where	x' is not free in V or $C[x]$ for Let-V,
	x' is not free in M or N for Let-A and Let-C.

Figure 9: Call-by-need reduction rules with variable substitution.

cannot capture a free variable. The Let-V rule, however, has the potential for variable capture. If the variable x is free in V, then substitution of V into $C[x]$ would introduce occurrences of x in the scope of the let, and these would incorrectly be bound. To avoid this, we choose a variable x' distinct from all free variables in V and $C[x]$, and substitute x' for x in $C[x]$ and in the let. We then replace the variable filling the hole in the context by the value V. The Let-A and Let-C rules also need modification. We substitute for the variable x a variable x' not free in M or N.

Any solution to the problem requires variables to be renamed, which is potentially confusing to the poor student trying to trace an ill-understood program. We have therefore tried to be as helpful as possible in choosing new variable names; a variable x will be renamed to x_0—or to x with some other subscript, if x_0 is not a fresh name. (It would be even more helpful if we renamed variables only when necessary, but this makes the calculus less clear [22]).

3.2 Factoring standard reduction

We factored standard reduction into two phases: computing all the filled evaluation contexts of a term, and taking the first—indeed, the only—one with a marked subterm reducible under $\xrightarrow{\text{NAME-S}}$. (Of course, lazy evaluation means that once the 'right' evaluation context has been found, no others will be computed.) This leads to a pleasingly clear program.

Consequently, `LetTrace` can readily be modified to use a different calculus, simply by changing these two functions to match the evaluation contexts and reduction rules of the new calculus.

3.3 Viewing a trace

`LetTrace` is currently non-interactive; it generates a text file with one reduction per line, as in Figure 6. However, `LetTrace` maintains more information than is shown in this figure. In particular, tracing also yields information about where the standard redex is, and what it reduces to. We have used this information to produce the prettier trace in Figure 7. This trace was produced entirely automatically; `LetTrace` can also create a program in John Hobby's METAPOST [10] language, a declarative language for generating PostScript output.

We have also experimented with generating input for `xfig`. This is not yet complete (we only mark the start of the standard redex, and the start of its reduction), and the result is not so pretty. However, the output is available directly, rather than having to be post-processed, and `xfig`'s scroll-bars make it quite good for browsing a tall wide picture.

Obviously, all of these interfaces could be improved. It would be nice to have interactive tracing, whereby the user could choose (for example) to skip display of some reduction steps. At the very least, it would be nice to have more control over the output, perhaps collapsing sections horizontally (within a term) and vertically (between terms).

3.4 Reading Gofer syntax

We have adapted a lexer supplied with the Gofer system (which was based on the definitions in the Haskell report [11]) to note additionally the position in the source file of each token, and to handle a literate syntax.

The parser used is based on Wadler's monadic parser [23], and follows the grammar found in Appendix A of the Gofer documentation [12] with only slight variations.

This allows the programmer to present input to `LetTrace` in (a subset of) literate Haskell or Gofer syntax, rather than directly as lambda expressions. Note, however, that the lets in the calculus are not letrecs, so definitions in the file cannot be implicitly recursive—though they may explicitly define and use the Y combinator—and earlier definitions cannot depend on later ones.

We have not yet provided conversion in the other direction, displaying the trace in high-level Gofer syntax, though clearly that would be a nice touch.

3.5 Relating the trace to the program

As mentioned above, the lexer records the position in the input file of each token in the input program. We have a version of `LetTrace` that maintains the start and end position in the input file of each subterm in the term being reduced. This position information is passed around in the right way through the various substitutions and rearrangements that arise during reduction.The propagation of this information is fairly straightforward; when it is not obvious to which subterm in the original evaluation context a particular resulting subterm relates, the entire evaluation context is chosen.

The use of these rules ensures that at every step, every subterm in the current term corresponds to a known location in the source code. This information could be used to draw arrows between matching parts of successive terms; with a more elaborate user interface it could be used to perform some type of animation, or to allow a user's double-click to take them to the relevant source code for a particular subterm from any point in the reduction sequence.

4 Extensions and related work

We have already mentioned some areas—for example, various aspects of the user interface—that require further work. In this section we briefly cover some more speculative directions. We also discuss related work.

4.1 Recursion

Our let-expressions are not recursive; the variable is bound in the body but not in the definition. We can implement recursion with the Y combinator. However, the Y combinator does not quite accurately model sharing with lazy data constructors [3].

This problem would be solved by extending the calculus to include also a letrec construct. Ariola *et al.* [3] give the axioms for such a construct. However, they are significantly more complex than the axioms for a non-recursive let.

4.2 Source-level tracing

Currently, `LetTrace` translates (a subset of) literate Gofer syntax into the lambda calculus, and never translates back again. An obvious use of the position information that we carry around for every subterm of the source program is to present to the user their original source code, rather than its rendition in the lambda calculus. We need to work out when we can present the original source unchanged, when we must present a reduced lambda-calculus version, and perhaps when we can rephrase reductions at the lambda-calculus level in terms of the original source.

4.3 Parallel evaluation

Our `LetTrace` system could be adapted without too much difficulty to trace parallel evaluation of non-overlapping redexes, given the right 'parallel lambda calculus' on which to base it.

4.4 Related work

The work most closely related to ours is that of the Calculator Project at Queen Mary and Westfield College and the Open University; this involved `MiraCalc` [9, 20], essentially a tracer for the lazy functional language Miranda[4]. `MiraCalc` uses a one-dimensional textual representation of terms, like `LetTrace`, but actually uses call-by-name rather than call-by-need reduction, and so does not really trace lazy evaluation. Auguston and Reinfelds [4] describe a similar system for tracing 'lazy evaluation' in Miranda, but again they use call-by-name rather than call-by-need reduction.

Raffalli [19] describes a tracer for pure lambda calculus (with no primitives), allowing call-by-name and a kind of lazy evaluation ('call-by-name with some sharing'). However, this system performs computation under lambda abstractions, so does not model lazy evaluation as in most functional languages.

Several authors [17, 21, 16, 26] describe a 'transformational' approach to tracing, whereby the program to be traced is transformed to return a pair consisting of the original result and some 'trace information'. However, this approach comes adrift because of lazy evaluation; the trace information usually concerns partially evaluated arguments, and forcing evaluation to record their values changes the behaviour of the program being traced.

Acknowledgements

The implementation described here is due mostly to Keith Wansbrough. The authors would like to thank Phil Wadler, whose presentation at CATS'94 inspired the work reported here, and the anonymous referees for their helpful comments.

References

[1] Samson Abramsky. The lazy lambda calculus. In David A. Turner (editor), *Research Topics in Functional Programming*, Chapter 4. Addison-Wesley, 1990.

[2] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. Technical Report CIS-TR-94-23, Computer Science Dept, University of Oregon, October 1994.

[3] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky and Philip Wadler. A call-by-need lambda calculus. In *Principles of Programming Languages*, 1995.

[4] Mikhail Auguston and Juris Reinfelds. A visual Miranda machine. In *5th Annual Working Conference on Software Engineering Education*, Dunedin, November 1994. IEEE. Extended abstract.

[5] Lennart Augustsson and Thomas Johnsson. The Chalmers Lazy-ML compiler. *Computer Journal*, Volume 32, Number 2, pages 127–141, 1989.

[4] Miranda is a trademark of Research Software Ltd.

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

[7] Richard S. Bird and Philip L. Wadler. *An Introduction to Functional Programming.* Prentice-Hall, 1988.

[8] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda calculus. In M. Wirsing (editor), *Formal Description of Programming Concepts III*, pages 193–219. Elsevier North-Holland, 1987.

[9] Doug Goldson. A symbolic calculator for non-strict functional programs. *Computer Journal*, Volume 37, Number 3, pages 178–187, 1994.

[10] J. D. Hobby. A METAFONT-like system with PostScript output. *Tugboat, the TEX User's Group Newsletter*, Volume 10, Number 4, pages 505–512, December 1989.

[11] P. Hudak, S. Peyton Jones and P. Wadler. Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, Volume 27, Number 5, May 1992.

[12] Mark P. Jones. *Gofer Manual.* Department of Computer Science, Yale University, 1991.

[13] John Maraist. Private email communication, 30 August 1995.

[14] John Maraist, Martin Odersky, David Turner and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus (extended abstract). In *11th International Conference on the Mathematical Foundations of Programming Semantics*, April 1995.

[15] John Maraist, Martin Odersky and Philip Wadler. The call-by-need lambda calculus (unabridged). Technical Report 28/94, Fakultät für Informatik, Universität Karlsruhe, October 1994.

[16] Lee Naish and Tim Barbour. Towards a portable lazy functional declarative debugger. Technical Report 95/27, Department of Computer Science, University of Melbourne, 1995.

[17] John T. O'Donnell and Cordelia V. Hall. Debugging in applicative languages. *Lisp and Symbolic Computation*, Volume 1, pages 113–145, 1988.

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

[19] Christophe Raffalli. A normaliser for pure λ-calculus. Chalmers University of Technology and Logic Team of Paris VII, August 1995.

[20] Steve Reeves, Doug Goldson, Pat Fung, Tim O'Shea, Mike Hopkins and Richard Bornat. The Calculator Project: Formal reasoning about programs. In *5th Annual Working Conference on Software Engineering Education*, Dunedin, November 1994. IEEE.

[21] Jan Sparud. Towards a Haskell debugger. In *Functional Programming Languages and Computer Architecture*, 1995.

[22] Allen Stoughton. Substitution revisited. *Theoretical Computer Science*, Volume 59, pages 317–325, 1988.

[23] Philip Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, 1990. Later version to appear in Mathematical Structures in Computer Science.

[24] Philip Wadler. A call-by-need lambda calculus. Presentation at CATS'94, Sydney, December 1994.

[25] Keith Wansbrough. Tracing lazy functional languages. Graduate project report, Department of Computer Science, University of Auckland, 1995.

[26] Richard Watson. Debugging techniques for functional languages. In Neil Leslie and Nigel Perry (editors), *Proceedings of the Massey Functional Programming Workshop*, 1994.