Tracing Lazy Functional Languages
07.483 Advanced Project in Computer Science[1]

Keith Wansbrough

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

kwan01@cs.auckland.ac.nz

October 11, 1995

[1]Project supervised by Dr. Jeremy Gibbons, jeremy@cs.auckland.ac.nz.

Abstract

The tracing of conventional imperative languages is straightforward, and tracing is probably the most common debugging technique in use today. However, attempting blindly to trace pure functional languages, especially lazy functional languages, does not work due to the fundamentally different model of execution. We claim that Ariola, Felleisen, Maraist, Odersky and Wadler's call-by-need lambda calculus provides a suitable formal model upon which to build a tracer for a lazy functional language, and we demonstrate this by exhibiting such a tracer, `LetTrace`. We also discuss a number of issues encountered in the implementation of this system, and propose a few areas for future work.

Contents

Chapter 1

Introduction

In conventional imperative languages, the program instructs the computer as to precisely what to do and in what order to do it. It is executed in a sequential, basically linear fashion. In contrast, a program in a pure functional language leaves the execution order unspecified. This allows the compiler or interpreter to perform transformations on the code, change the order of execution or even execute parts of the program in parallel.

This feature becomes a disadvantage for debugging. In a conventional imperative language, one usually has a 'source-level debugger' which steps through the program, executing one statement at a time and displaying the current state of the program at each step. Even without a source-level debugger, one is able to insert `write` statements in interesting places in the program, allowing the programmer to monitor what is happening.

In a pure functional language, one cannot 'step through' the code; the order in which the program is executed need bear no relation to the order of the program source. In fact, in a *lazy* functional language, the order in which the program is executed depends in part upon what is done with the output!

The `write`-statement approach is not available either, as pure functional languages are devoid of side-effects. Augustsson and Johnson [AJ89] experimented with adding an impure primitive `trace` to a lazy functional language (Chalmers Lazy-ML): `trace` x y would return x as result but print the value of y on the terminal as a side-effect. They note 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`.'

We see that there is some essential difficulty with tracing evaluation in a pure functional language. The difficulty is in fact that running a pure functional program consists not of a step-by-step execution of statements, but of the reduction of an expression to normal form. Hence a trace should not consist of the sequence of statements executed, but of the reduction sequence from the original expression to the result. Examples of traces of this form appear in [BW88, p. 5].

To generate this trace, we need to have an accurate model of what goes on inside the compiler or interpreter when it executes our program. In 1975, Plotkin [Plo75] showed that the call-by-value lambda calculus is a good model for the execution of call-by-value functional languages, and that the call-by-name lambda calculus is a good model for call-by-name functional languages. But modern lazy functional languages, while preserving the semantics of call-by-name, in fact operate differently, and so the call-by-name calculus will not serve our purposes here.

Recently, Ariola, Felleisen, Maraist, Odersky and Wadler [AF94, MOW94, AFM+95, MOW95] have developed a *call-by-need* lambda calculus that accurately models the behaviour of a lazy functional language evaluator. Wadler reported on this work at CATS-94. It seemed to us that their calculus would be an ideal formalism upon which to base a tracer for lazy functional languages, and in this project we demonstrate that this is indeed the case.

The rest of this report is structured as follows. In Chapter 2 we review the call-by-name and call-by-value calculi, and then present Ariola, Felleisen, Maraist, Odersky and Wadler's call-by-need calculus. We also discuss the four papers in which their calculus is presented, and note some interesting differences between them. In Chapter 3 we discuss the project itself: what we did and what motivated our approach. In Chapter 4 we discuss the results; specifically, we describe a typical user interaction with `LetTrace`, our tracing system. In Chapter 5 we discuss the details of the implementation, and a number of interesting results and discoveries arising therefrom. Finally, in Chapter 6 we list a few areas for future work. The report ends with a listing of the `LetTrace` program, in Gofer, as Appendices A through D.

Chapter 2

Lambda calculi

In 1932, Alonzo Church [Chu32, Chu41] introduced the lambda calculus as a model for computation. It has since become the preferred model of language implementors, due to its incorporation (unlike, say, the Turing Machine) of fundamental concepts such as variable and procedure call.

In this chapter, we briefly review the call-by-name (λ_{NAME}) and call-by-value (λ_{VAL}) calculi, and then proceed to introduce the call-by-need (λ_{NEED}) lambda calculus. Along the way we introduce the notion of *standard reduction.* After a discussion of the various versions of call-by-need calculi, we look at adding constructors and primitive operations to the calculus. Finally, we present a brief survey of the papers upon which this present report is based.

For a far more extensive treatment of lambda calculi, the reader is referred to Barendregt's book [Bar81].

2.1 Notation

In what follows, we write $fv(M)$ for the free variables of a term. We write $M[x := N]$ for the substitution of N for every free occurrence of the variable x in term M. The expression $M\,N$ is an *application* of the *function* (or *functor*) M to the *argument* N. The expression $\lambda x.M$ is an *abstraction*, in which x is the *variable* and M is the *body.* In the let-expression $\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N$, x is the *variable*, M the *binding* and N the *body.* We sometimes refer to a reducible subterm of a term as a *redex*, an abbreviation for *red*ucible *ex*pression. We adopt Barendregt's *variable convention* [Bar81] (but see Section 5.1), that in any term all the bound variables are distinct from all the free variables. Note that following Barendregt [Bar81], we work not with terms but with

Syntactic domains:

VARIABLES	x, y, z
VALUES	$V, W ::= x \mid \lambda x.M$
TERMS	$L, M, N ::= V \mid M\,N$

Reduction rule:

$$(\beta) \quad (\lambda x.M)\,N \xrightarrow{\text{NAME}} M[x := N]$$

Figure 2.1: The call-by-name calculus λ_{NAME}.

equivalence classes of α-renamable terms[1].

2.2 The call-by-name calculus

The call-by-name calculus is the classic pure lambda calculus as introduced by Church. It is presented in Figure 2.1. VARIABLES is a countably infinite set of variables. VALUES are terms that cannot be further reduced and can be returned as answers. TERMS, also written Λ_{NAME}, is the set of all terms in the calculus. The calculus itself is referred to as λ_{NAME}.

The one and only reduction rule, β, applies an abstraction to an application by performing a substitution of the argument of the application for every free occurrence of the variable in the body of the abstraction. This single step reduction is written $\xrightarrow{\text{NAME}}$.

We may also define the syntactic domain of *contexts*. A context is simply a term with a single hole '$[\,]$' in place of one of the subterms. We can define the contexts for our call-by-name calculus as follows:

CONTEXTS $\quad C ::= [\,] \mid \lambda x.C \mid C\,M \mid M\,C$

A context $C[\,]$ can have its hole filled by a term M; the resulting *filled context* is denoted by $C[M]$. This filled context corresponds to the obvious term, C with M substituted for the hole; if N is the resulting term, we write $C[M] \approx N$.

[1]The α rule is $\lambda x.M = \lambda y.M[x := y]$, $y \notin fv(M)$, which allows the renaming of variables as long as they are not thereby captured

$$
\begin{array}{lll}
 & & (\lambda x.x\,x)\,((\lambda y.y)\,(\lambda z.z)) \\
(\beta) & \xrightarrow{\text{NAME}} & ((\lambda y.y)\,(\lambda z.z))\,((\lambda y.y)\,(\lambda z.z)) \\
(\beta) & \xrightarrow{\text{NAME}} & (\lambda z.z)\,((\lambda z.z)\,(\lambda z.z)) \\
(\beta) & \xrightarrow{\text{NAME}} & (\lambda y.y)\,(\lambda z.z) \\
(\beta) & \xrightarrow{\text{NAME}} & \lambda z.z
\end{array}
$$

Figure 2.2: A call-by-name reduction sequence

Contexts have been introduced in order to define the *compatible closure* $\xrightarrow{\text{NAME}}$ of $\xrightarrow{\text{NAME}}$. The compatible closure is the least relation including $\xrightarrow{\text{NAME}}$ that is closed under the implication $M \xrightarrow{\text{NAME}} N \Rightarrow C[M] \xrightarrow{\text{NAME}} C[N]$.

Figure 2.2 gives a sample call-by-name reduction sequence. Notice that the argument $(\lambda y.y)\,(\lambda z.z)$ is reduced *twice*.

2.3 Standard reduction

λ_{NAME}as presented above is not deterministic. For example the term

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

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

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

and

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

But we want a deterministic, sequential model of program execution. To achieve this, we introduce the notion of *standard reduction*, as a deterministic restriction of the general notion of reduction.

Following Maraist, Odersky and Wadler [MOW95] (and not Ariola and Felleisen, see Section 2.7), we do not modify the calculus itself. Instead, we add a new syntactic domain, *evaluation contexts* ECONTEXTS, a subset of CONTEXTS:

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

We now restrict *standard* reduction $\xmapsto{\text{NAME}}$ to be the least closure of $\xrightarrow{\text{NAME}}$ under the more restricted implication $M \xmapsto{\text{NAME}} N \Rightarrow E[M] \xmapsto{\text{NAME}} E[N]$. A standard reduction step $M \overset{\text{NAME}}{\longmapsto} N$, therefore, involves finding E and K such that $M \approx E[K]$ and $E \in$ ECONTEXTS, reducing K to L via $\xrightarrow{\text{NAME}}$, and obtaining the new term $E[L] \approx N$.

This rule is known as the 'leftmost-outermost' reduction rule, as it reduces the leftmost outermost reducible subterm. It is straightforward to show that any Λ_{NAME} term has at most one reducible evaluation context[2], and so the standard reduction relation $\overset{\text{NAME}}{\longmapsto}$ is deterministic.[3]

2.4 The call-by-value calculus

Next we come to the other well-known calculus, λ_{VAL}, presented in Figure 2.3. Like λ_{NAME}, λ_{VAL} has a single reduction rule: β_V. This is very similar to β, but where β unconditionally substitutes the argument for free occurrences of the variable in the body, β_V requires that the argument be a value, i.e. not further reducible, first. This means we must also augment ECONTEXTS by an evaluation context for the argument of an application of an abstraction, to allow the reduction of this term to a value and avoid getting stuck.

Again it is possible to show that the standard reduction relation so defined is indeed deterministic[4].

The terms of λ_{VAL}, Λ_{VAL}, are the same as those of the call-by-name calculus. But the reduction relation $\xrightarrow{\text{VAL}}$ and standard reduction relation $\overset{\text{VAL}}{\longmapsto}$ are different from their call-by-name versions. To see this, define $\Omega = (\lambda x.(x\,x))\,(\lambda x.(x\,x))$ and $\mathsf{K} = \lambda x.\lambda y.x$. Now consider the term $(\mathsf{K}\,(\lambda x.x))\,\Omega$. Under call-by-name, this standard-reduces to $(\lambda y.(\lambda x.x))\,\Omega$ and then to $\lambda x.x$,

---

[2]**Proof** [Jeremy Gibbons, unpublished]. Throughout this proof, 'reducible' means reducible under $\xrightarrow{\text{NAME}}$, and a subterm N of a term M is 'markable' if there is an evaluation context $E[\,]$ such that $M \approx E[N]$. From the grammar of evaluation contexts, we see that if a strict subterm N of a term M is markable, then M is an application, whose function part is an application, whose ... whose function part is N—that is, M is a left-nested chain of applications ending at N. Thus, markable subterms of a term are on the 'left spine', and of two such subterms, one must be a subterm of the other. We show that if a subterm N of a term M is markable and reducible, then no strict subterm of N is markable and reducible; hence, a term has at most one markable and reducible subterm. Suppose N is reducible; then either N is an application whose function part is an abstraction, or N is a let-expression. If N is a let-expression, then no strict subterm of N is markable. If N is an application whose function part is an abstraction, only subterms of the function part may be markable; but the function part is an abstraction, and an abstraction is not reducible, and no strict subterm of an abstraction is markable. QED.

[3]This proof of the existence of a unique evaluation context is known as a Unique Evaluation Context theorem, and follows the idea of [FF87], where Felleisen and Friedman prove the existence of a unique sk-context for control strings of their CEK-machine.

[4]The proof is similar to that for λ_{NAME}, and is omitted.

Syntactic domains:

Variables	x, y, z
Values	$V, W ::= x \mid \lambda x.M$
Terms	$L, M, N ::= V \mid M\,N$
Contexts	$C ::= [\,] \mid \lambda x.C \mid C\,M \mid M\,C$
EContexts	$E ::= [\,] \mid E\,M \mid (\lambda x.M)\,E$

Reduction rule:

$$(\beta_V) \quad (\lambda x.M)\,V \xrightarrow{\text{VAL}} M[x := V]$$

Figure 2.3: The call-by-value calculus λ_{VAL}.

$$\begin{array}{rcl} & & (\mathsf{K}\,(\lambda x.x))\,\Omega \\ & = & ((\lambda x.\lambda y.x)\,(\lambda x.x))\,((\lambda x.(x\,x))\,(\lambda x.(x\,x))) \\ (\beta_V) & \stackrel{\text{VAL}}{\longmapsto} & (\lambda y.(\lambda x.x))\,((\lambda x.(x\,x))\,(\lambda x.(x\,x))) \\ (\beta_V) & \stackrel{\text{VAL}}{\longmapsto} & (\lambda y.(\lambda x.x))\,((\lambda x.(x\,x))\,(\lambda x.(x\,x))) \\ (\beta_V) & \stackrel{\text{VAL}}{\longmapsto} & \ldots \end{array}$$

Figure 2.4: A non-terminating reduction sequence in λ_{VAL}.

which is a value, and we terminate. Under call-by-value, however, we get the non-terminating reduction sequence illustrated in Figure 2.4 (since $\Omega \stackrel{\text{VAL}}{\longmapsto} \Omega$ and its value is required to reduce the application). In functional language terms, call-by-value is *strict*—it requires the full evaluation of an application's arguments before applying the function. This causes problems with an argument whose value is undefined (Ω, for example) that do not occur with a call-by-name calculus.

2.5 The call-by-need calculus

Recall the expression $(\lambda x.x\,x)\,((\lambda y.y)\,(\lambda z.z))$ which we considered in Section 2.2 (Figure 2.2). When this is reduced under λ_{NAME} the subexpression

$(\lambda y.y)\,(\lambda z.z)$ is evaluated twice. In λ_{VAL}, this does not occur: the argument is reduced to the value $\lambda z.z$ first, and only *then* substituted for x in $x\,x$. So the work of reducing the argument is done just once. But in the example of a call-by-value reduction sequence (Figure 2.4), we saw that sometimes a λ_{VAL} reduction sequence will not terminate, when a λ_{NAME} one will. Similarly, even if the argument of an application *does* terminate, if it is a large, expensive computation and the result is thrown away (by K, for example), the effort is wasted.

Writers of compilers and interpreters for functional languages noticed this, and in 1971 Wadsworth [Wad71] proposed the mechanism now known as call-by-need, and implemented in so-called *lazy* functional languages: only evaluate the argument of an application *when its value is required*, and 'memoize' the result so that if the argument's value is required again it need not be recalculated. (This of course requires that the language be pure: without referential transparency, there would be no guarantee that the expression would have the same value at subsequent points in time). Intuitively, it is obvious that this technique gives identical results to call-by-name. This technique is now standard for implementations of non-strict functional languages.

From a pragmatic point of view, this is excellent. Laziness, or call-by-need, speeds up the evaluation of Λ_{NAME} terms without changing the semantics. Compiler and interpreter writers can happily go away and write systems implementing this by various techniques: pointer manipulation, graph rewriting, and so on.

However, we have a problem. There is a developed calculus, λ_{NAME}, with which to understand the behaviour of programs under call-by-name, and in which to prove the validity of optimising transformations and so on. But our programs are *not* running on a call-by-name evaluator but on a call-by-*need* one. Hence their behaviour will be different. Maraist, Odersky and Wadler [MOW95] describe the danger of the gap between conceptual and implementation calculi, which "...might lead to program transformations that drastically increase the complexity of lazy functional programs.":

> Experience shows this can be a significant problem in practice. The Glasgow Haskell Compiler is written in Haskell and self-compiled; it makes extensive use of program transformations. One such transformation inadvertently introduced a loss of sharing, causing the symbol table to be rebuild *each time* an identifier was looked up. The bug was subtle enough that it was not caught until profiling tools later pinpointed the cause of the slowdown.

The obvious solution is to find a calculus that accurately models the behaviour of a lazy functional language. Recently two independent groups,

Syntactic domains:

$$\begin{array}{ll}
\textsc{Variables} & x, y, z \\
\textsc{Values} & V, W ::= x \mid \lambda x.M \\
\textsc{Terms} & L, M, N ::= V \mid M\, N \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N \\
\textsc{Contexts} & C ::= [\,] \mid \lambda x.C \mid C\, M \mid M\, C \\
 & \quad \mid \mathsf{let}\ x\ \mathsf{be}\ C\ \mathsf{in}\ N \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ C
\end{array}$$

Reduction rules:

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

Figure 2.5: The call-by-need calculus $\lambda_{\textsc{need}}$.

Ariola and Felleisen [AF94] at the University of Oregon and Rice University, Texas, and Maraist, Odersky and Wadler [MOW94, MOW95] at the Universität Karlsruhe and the University of Glasgow came up with such calculi. The two calculi are essentially the same (but see later), and the two groups have published a combined paper [AFM$^+$95].

The call-by-need calculus[5], $\lambda_{\textsc{need}}$, is detailed in Figure 2.5. We mainly follow Maraist, Odersky and Wadler's presentation [MOW95]; differences are noted below.

The basic term language $\Lambda_{\textsc{name}}$ is augmented with a new production, the let-expression. The expression $\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N$ binds the variable x to the expression M, in the expression N. We call x the *variable*, M the *binding* and N the *body*. This expression explicitly shows the sharing implicit in the lazy approach, without relying on non-source-syntax concepts such as environments or graphs: all free occurrences of x in N share the same binding M.

When the value of x is required, M is reduced to a value; due to sharing,

[5] So called (rather than the 'lazy lambda calculus') to avoid a name clash with work by Abramsky and Ong [Abr90], which unfortunately describes a call-by-*name* calculus.

the result of this evaluation is then available to *all* free occurrences of x in M—the evaluation need only be done once.

The reduction rules of λ_{NEED} are somewhat more complex than those of λ_{NAME} and λ_{VAL}. The five rules are explained as follows:

- The first rule introduces a let-expression (hence the name LET-I), translating an application of an abstraction into a corresponding binding of the variable over the scope of the abstraction body.

- The second rule, LET-V, performs substitution of the binding for free occurrences of a variable in the body of a let. Hence LET-I and LET-V together are almost equivalent to β. But they are not quite the same: LET-V only performs the substitution once the binding has been reduced to a value, and hence cannot be reduced any further. In this way, the calculus ensures that the binding is evaluated at most once—this is the laziness property.

  Note that the rule as defined here (and in [AF94, AFM+95, MOW94, MOW95]) performs a one-at-a-time substitution: *one* free occurrence of the variable has the binding substituted; the other free occurrences remain. In [MOTW95] an alternate version of the LET-V rule is given, which performs the substitution 'all-at-once'. If this is done, it is in fact possible to remove the let binding as well, as it is no longer needed. However, in a personal communication [Wadler, April 1995], Wadler says it is better to use the one-at-a-time formulation, as it "yields standard reductions closer to the operational definition of call-by-need."

- The third rule, LET-C, allows let bindings to commute with application (hence the name). Consider the term

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

  (writing I for $\lambda x.x$). We would like to associate the abstraction $\lambda w.z\,w$ with its argument I, but as written this cannot occur. The solution is to allow the let binding to commute with the application, resulting in

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

  After this transformation, reduction can proceed as expected.

- The fourth rule, LET-A, allows for another similar case. Sometimes we need to turn left-nested let bindings into right-nested ones, changing the *A*ssociation of the bindings. Consider the term

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

After introducing lets with LET-I we obtain

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

We cannot substitute for f in $f\,I\,(f\,I)$, as its binding is not yet a value. But the binding *cannot* be reduced to a value, as that would require either the substitution of $I\,I$ for z (illegal, since $I\,I$ is not a value) or the reduction of $I\,I$ first (also illegal, since we do not know we will need its value yet, and cannot until the binding of f is applied).

Hence we reassociate the bindings, to get

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

which we can readily reduce.

- Finally, we add the garbage collection rule of Felleisen and Hieb [CF90, FH92, AF94, AFM+95]. This disposes of let bindings that are no longer required: if the variable does not occur free within the body, the binding cannot be of any use.

The calculus as presented above, as with λ_{NAME}, is not deterministic. We define the set of evaluation contexts for Λ_{NEED}, and present a new set of *standard* reduction rules (Figure 2.6), as a restriction of the *general* reduction rules already presented.

Here standard values (SVALUES) exclude bare variables. Valid ANSWERS are values wrapped in zero or more bindings (a value can have no free variables, and hence these bindings are redundant and the grammar allows the system to 'see through' these extraneous bindings to the variable inside).

Valid evaluation contexts are:

- the whole term;
- the function part of an application;
- the body of a let-expression, since the body of a function should be reduced before arguments are substituted;
- and the binding, *only if the variable appears in an evaluation context;* i.e., its value is needed for further progress.

Unfortunately, this is still not enough to ensure determinism. The solution is to restrict rules LET-C and LET-A to operate only on ANSWERS; in allowing ANSWERS as answers, we must reject the LET-G rule as another

Additional syntactic domains:

$$
\begin{array}{ll}
\textsc{SValues} & V_s ::= \lambda x.M \\
\textsc{Answers} & A ::= V_s \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ A \\
\textsc{EContexts} & E, E_i ::= [\,] \mid E\,M \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ E \\
 & \quad \mid \mathsf{let}\ x\ \mathsf{be}\ E_0\ \mathsf{in}\ E_1[x]
\end{array}
$$

Standard reduction rules:

$$
\begin{array}{ll}
(\textsc{Let}\text{-}I_s) & (\lambda x.M)\,N \overset{\text{NEED}}{\longmapsto} \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M \\
(\textsc{Let}\text{-}V_s) & \mathsf{let}\ x\ \mathsf{be}\ V_s\ \mathsf{in}\ E[x] \overset{\text{NEED}}{\longmapsto} \mathsf{let}\ x\ \mathsf{be}\ V_s\ \mathsf{in}\ E[V_s] \\
(\textsc{Let}\text{-}C_s) & (\mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ A)\,N \overset{\text{NEED}}{\longmapsto} \mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ (A\,N) \\
(\textsc{Let}\text{-}A_s) & \mathsf{let}\ y\ \mathsf{be}\ (\mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ A)\ \mathsf{in}\ E[y] \\
 & \quad \overset{\text{NEED}}{\longmapsto} \mathsf{let}\ x\ \mathsf{be}\ L\ \mathsf{in}\ (\mathsf{let}\ y\ \mathsf{be}\ A\ \mathsf{in}\ E[y])
\end{array}
$$

Figure 2.6: Standard reduction in λ_{NEED}.

source of non-determinism (but see [MOW94, sec. 8] and the discussion in Section 2.7).

Defining the standard reduction $\overset{\text{NEED}}{\longmapsto}$ as before, as the compatible closure of $\overset{\text{NEED}}{\longmapsto}$ under the implication $M \overset{\text{NEED}}{\longmapsto} N \Rightarrow E[M] \overset{\text{NEED}}{\longmapsto} E[N]$, we can prove it is deterministic [MOW94, A.2], as we require.

This calculus can be proven to be observationally equivalent to the call-by-name lambda calculus λ_{NAME} on their common term language Λ_{NEED} (obviously λ_{NAME} does not apply to terms containing let-expressions originally). In fact, this proof makes up the bulk of each of the four papers [AF94, AFM+95, MOW94, MOW95]. In other words, it can be demonstrated to give the same results (*semantics*) as call-by-name.

However, its *pragmatics*—its operational behaviour—is different. As intended, it models accurately the behaviour of real, lazy implementations of pure functional languages.

It is possible to express this calculus without using let; we simply observe that the expression $\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N$ is simply syntactic sugar for $(\lambda x.N)\,M$. Performing this substitution in all the rules gives a calculus with the same semantics and effectively the same pragmatics as λ_{NEED}, but defined exclusively on the term language Λ_{NAME}. This makes it easier to compare the calculus with

λ_{NAME}, and to derive basic syntactic results. Ariola and Felleisen and Maraist, Odersky and Wadler all mention this [AF94, AFM+95, MOW94, MOW95], and Maraist, Odersky and Wadler name this **let**-less calculus $\lambda^{\ell}_{\text{NEED}}$. We discuss this calculus no further here, as it is somewhat less natural than the **let** formulation we have presented, and hence is counter to our purposes (namely, to aid in the debugging of programs by presenting a readable trace of execution to the programmer).

2.6 Adding constructors and primitives

A language without constructors (for example, numbers and pairs) and primitives (basic arithmetic operations, projections) is convenient to work with from a theoretical point of view, due to its simplicity. But the simulation of these through techniques such as Church numerals is significantly less efficient than implementing them directly, as numbers and arithmetic are generally available in hardware. For this reason, most functional languages provide constructors and primitives directly.

Figure 2.7 summarises the calculus with the addition of primitive operators and constructors. It is taken basically from [MOW95] with the addition of the ECONTEXTS production from [GW95]—the argument of a primitive must be able to be evaluated if it is not already a value. Note that operators in this scheme are unary and strict, and constructors are strict.

For more details, see [AFM+95], [MOW95] or [GW95]; we do not discuss this further here (see Section 5.3).

2.7 Sources

The call-by-need lambda calculus described above comes originally from work by Ariola and Felleisen and by Maraist, Odersky and Wadler. The two groups arrived at their results independently, and subsequently published a combined paper. We based this project primarily on four papers:

- [AF94] by Ariola and Felleisen, their technical report which appears to have been mostly written prior to their discovery of Maraist, Odersky and Wadler's work;
- [MOW94] by Maraist, Odersky and Wadler, *their* technical report, which briefly mentions [AF94];
- [AFM+95] by all five, a combined paper presented at *Principles of Programming Languages*, January 1995;

Syntactic domains:

Operators	p
Constructors	k^n (of arity n)
Values	$V, W ::= x \mid \lambda x.M \mid k^n\, V_1 \ldots V_n$
Terms	$L, M, N ::= V \mid M\,N \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N \mid p$
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: All the rules of λ_{NEED} (Figure 2.6), with the addition of:

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

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

Figure 2.7: λ_{NEED} augmented with constructors and operators.

- [MOW95] by Maraist, Odersky and Wadler, a paper submitted to the *Journal of Functional Programming* in 1995.

We also referred to a number of other papers. In this section, we mention a few notable points from the papers.

All the papers discuss the motivation for constructing a call-by-name calculus, cover the call-by-name calculus as an introduction, introduce a call-by-need calculus, give it a standard reduction system, prove it is sound and complete (or at least sketch the proofs), introduce the let-less form of the calculus, and mention some extensions (including letrec (see Section 6.4) and constructors). Except for letrec, this material has been covered in this chapter.

The major difference between the approaches of Ariola and Felleisen and Maraist, Odersky and Wadler is in how they handle standard reduction. Ariola and Felleisen restrict their calculus so that their axioms can also be read as standard reduction rules, simply by replacing the = in the axioms with what we have called $\overset{\text{NEED}}{\longmapsto}$, and obtaining the standard reduction relation (our $\overset{\text{NEED}}{\longmapsto\!\!\!\!\to}$) by taking the compatible closure. In contrast, Maraist, Odersky and Wadler choose to maintain both a calculus of *general* reductions (our $\overset{\text{NEED}}{\longrightarrow}$), and a *separate* restriction of this, a set of *standard* reduction rules (our $\overset{\text{NEED}}{\longmapsto}$). In their papers (e.g. [MOW95, sec. 1]) they point out that this means more program transformations can be expressed as equalities than in Ariola and Felleisen's calculus. In this project we have mainly followed Maraist, Odersky and Wadler.

Other differences are less significant. [MOW94, AFM+95] use the notation $\mathsf{let}\ x = M\ \mathsf{in}\ N$ for lets (indeed Wadler was still using this notation at CATS'94 [Wad94]); the clearer $\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N$ we use in this report comes from [AF94] and is used also in [MOW95]. All the papers except [MOW95] refer to λ_{let} rather than λ_{NEED}; [AF94] calls this the 'let-calculus' to distinguish it from what it calls the 'call-by-need calculus' and we (along with the other papers) call the 'let-less call-by-need calculus'. This latter is called λ_ℓ in [MOW94, AFM+95], λ_{need} in [AF94], and $\lambda^{\ell}_{\text{NEED}}$ in [MOW95].

The proof techniques used in the papers are different. Ariola and Felleisen [AF94, sec. 4] have a graphical proof of the completeness of λ_{NEED} using decorated directed acyclic graphs. This is the proof used in the combined paper [AFM+95]. Maraist, Odersky and Wadler's proof [MOW94, sec. 5] is different, and does not involve the use of decorated DAGs.

Ariola and Felleisen in [AF94] consider two other calculi, a fully lazy calculus and a 'lenient' calculus, in relation to λ_{NEED}. These are used as examples of the application of the call-by-need calculus. They also discuss some potential further applications of the calculus. In their extensions section they mention the need for *lazy* constructors.

Maraist, Odersky and Wadler in [MOW94] present a natural operational semantics for call-by-need, and show its correspondence with their standard reduction system.

The two groups have different approaches to LET-G. Maraist, Odersky and Wadler include it in their calculus but throw it out for their standard reduction system; since Ariola and Felleisen do not distinguish between the calculus and the standard reduction system, they have to exclude it. However they do suggest work on garbage collection as an extension, and present the garbage collection rule with the suggestion that a rigorous treatment of garbage collection would be useful. Maraist, Odersky and Wadler in [MOW94, sec. 8] explain that adding LET-G to the standard reduction system is just like adding a garbage collector to a language: it can operate at any time and as much as it wants without affecting the result. This is our justification for including LET-G in our system despite it not being 'standard'. They also discuss in the same section a similar optimisation, 'indirection shortening'.

A number of other papers were also referred to.

The seminal paper for this type of work (constructing a calculus to represent the behaviour of a real implementation or system) is [Plo75]. In it

Plotkin introduces λ_{VAL} and λ_{NAME}, the calculi we presented in Sections 2.4 and 2.2 respectively, and relates them to a subset of ISWIM (a real call-by-value language) and a variant of his own, a call-by-name language. It is interesting to note that he includes constants in his treatment, unlike the call-by-need papers we have discussed. He does not however include letrec. After the proofs (similar in spirit to those of Ariola, Felleisen, Maraist, Odersky and Wadler), he goes on to provide translations of call-by-name into call-by-value and vice versa, by means of continuation-passing (cps). A cps transformation has the advantage that the resulting code is calling-style independent: it will behave the same under call-by-name or call-by-value.

[OLT94] describes a cps transformation for call-by-need. [FF87, sec. 3.2] contains a Unique Evaluation Context lemma, as required for call-by-need to prove that the standard reduction system is indeed standard. [MOTW95] is notable for suggesting the use of the *linear* lambda calculus to help model call-by-need; it also presents a typed call-by-need calculus. Finally, we referred to [Jon94] as an example of how a real lazy functional language (Gofer) operates.

Chapter 3

LetTrace

The excellent match of the call-by-need calculus λ_{NEED} to the reality of modern lazy functional languages immediately suggested using it as the basis of a tracing system for the latter.

The source-level style of the calculus (representing sharing by let-expressions, corresponding to `let`-expressions in Gofer for example) suggested that the translation required between the source, the calculus and the user would be minimal (compare this with the graphs used in [BW88, sec. 6.2.2], for example, and the pointer system they represent). The ease of writing a term involving sharing as a single line of text rather than as a two-dimensional graph was also apparent.

We decided to use the calculus directly as an evaluator, rather than attempting to reinterpret the internal behaviour of an existing implementation of a lazy functional language in terms of the calculus. By taking this approach we were able to use the calculus in an extremely direct way, which would not have been possible otherwise. We were also able to ignore much of the complexity of a real functional language, and focus on the essential core of the language.

The `LetTrace` system, then, is a complete stand-alone system. We chose to allow the user to input Λ_{NEED} terms using a syntax very similar to but not identical with a certain subset of Gofer. The clarity of the relationship between the code of `LetTrace` and the statement of the calculus (compare the definitions of `evctxl` and `reducl` in Appendix A with Figure 2.6, see Section 5.2) resulting from the decision to implement `LetTrace` as a stand-alone system suggests that the right choice was made.

`LetTrace` is coded in Gofer, itself a lazy functional language, and a future goal is that it be able to trace its own execution. This is a major reason for the decision to allow as input a subset of Gofer, rather than using or inventing some other syntax.

The calculus we eventually implemented turns out to be fractionally different from the calculus detailed in Chapter 2. We do not believe the differences are significant. Details are given in Sections 5.2 and 5.3. We also found it necessary to make some changes to the calculus with regard to variable renaming and substitution; these are detailed in Section 5.1. But overall, `LetTrace` directly implements a calculus very close to λ_{NEED}.

Chapter 4

Results

`LetTrace` contains at its core an evaluator based on λ_{NEED}. However, a means must be provided for the user to supply Λ_{NEED} terms, and a means must also be provided for the user to view the sequence of terms in the reduction. As stated above, we chose to allow the user to input terms in a slightly modified subset of Gofer syntax (see Section 5.5). For the output we originally performed no translation, and simply printed out the Λ_{NEED} terms using a straightforward ASCII notation. However the tracer actually provides more information than just the sequence of terms, and the encoding of this additional information in ASCII became unwieldy (not to mention the difficulty of interpreting terms whose representations wrapped around the screen several times). So it was decided to use a graphical output system, XFig. Note that this is intended merely as a prototype—a production version of `LetTrace` would have an interactive graphical user interface. This is discussed further in Sections 5.4 and 6.1.

A typical user interaction with `LetTrace` proceeds as follows. First, the user types their source in `LetTrace` (we overload the name of the system to also denote the language it accepts, as is customary with computer languages). The convention is to use the extension `.ltr`. The source files, as described in Section 5.5, are presented in a literate form: source lines are introduced with a `>`, and all other lines are considered comment lines. A sample source file is presented in Figure 4.1, defining a function `intsum` which computes $\sum_1^n i$ given n as argument.

Next, the user chooses a term to reduce within the environment defined by their source file. For our example, we choose the term `intsum 3`.

This term must now be encoded in `LetTrace`'s internal syntax; the source location information is set to line 9999 arbitrarily. `intsum 3` is encoded as shown in Figure 4.2. Note that this expression is placed within a call to `convert` (Section 5.7). This script is a literate Gofer script, and it is read

```
intsum.ltr, sample source file for LetTrace.

Keith Wansbrough 1995-07-11..1995-10-05.

First the Y combinator, which encapsulates recursion
in lambda calculus.

> y = \f -> ( (\xa -> (f (xa xa))) (\xb -> (f (xb xb))) )

Now a simple trial function: intsum, defined as follows:

                  n
                 ---
                 \
     intsum n  =  >  i
                 /
                 ---
                  1

We can define this recursively as:

     intsum 0     = 0
     intsum (n+1) = (n+1) + intsum ((n+1)-1)

or in C notation as

     intsum n = (Iszero n) ? 0 : (Add n (intsum (Pred n)))

In LetTrace, we write:

> intsum' = \g -> (\n -> (((Iszero n) 0)
>                                      ((Add n) (g (Pred n)))))

> intsum = y intsum'
```

Figure 4.1: `intsum.ltr`, a sample `LetTrace` source file.

```
intsum.lgs, a sample script encoding intsum 3
  for input to LetTrace.

Keith Wansbrough 1995-10-05..1995-10-05.

myterm is the encoding of intsum 3 in the environment defined
by intsum.ltr.

> myterm = (convert (parsefile "intsum.ltr") (t,tr))
>     where t  = (App (Var (V "intsum")) (Con (Delta 3)))
>           tr = termToTrmRnge t (R (P 9999 1) (P 9999 1))
```

Figure 4.2: `intsum.lgs`, a script to encode `intsum 3`.

into the Gofer system after reading the `LetTrace` source files[1].

Once the `LetTrace` system and the script to build the input term are loaded into the Gofer system, we run `LetTrace` on the term by entering the command

```
writefigfile "intsum.fig" (evaltofig (evaluate stdredl myterm))
```

This will write a file `intsum.fig` containing XFig code for a figure representing the reduction sequence[2].

Next, we fire up XFig and load in the file `intsum.fig`. This figure will be very large—the reason we used XFig rather than, say, outputting PostScript or LaTeX `picture` code was primarily the fact that XFig has scroll bars, whereas most pieces of paper don't. A portion of `intsum.fig` is displayed in Figure 4.3; the whole trace is displayed (greatly reduced) in Figure 4.4. It is possible to see each step in the reduction, which rule is applied, to what evaluation context (indicated by the boxes), and where each subterm moves to in the next step (the arrows). The boxes underline the head of the evaluation context chosen, as in [AF94, sec. 2]. Note in Figure 4.4 how it is possible to see the four steps in the recursion.

[1]In retrospect, it would have been nice to have built a wrapper for the call to `convert` and `termToTrmRnge`.

[2]A wrapper for this would have been nice also.

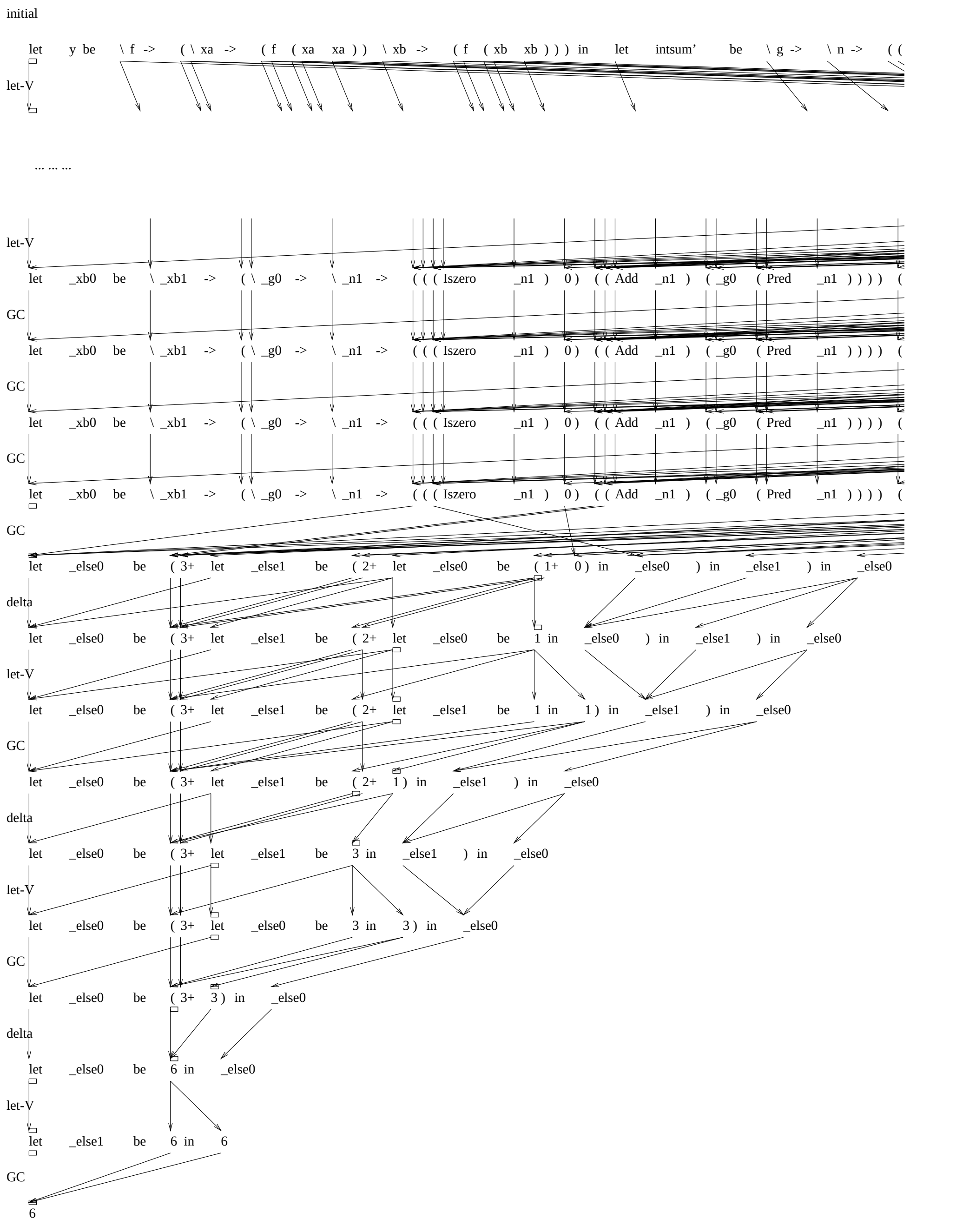

Figure 4.3: A portion of the trace of `intsum 3`.

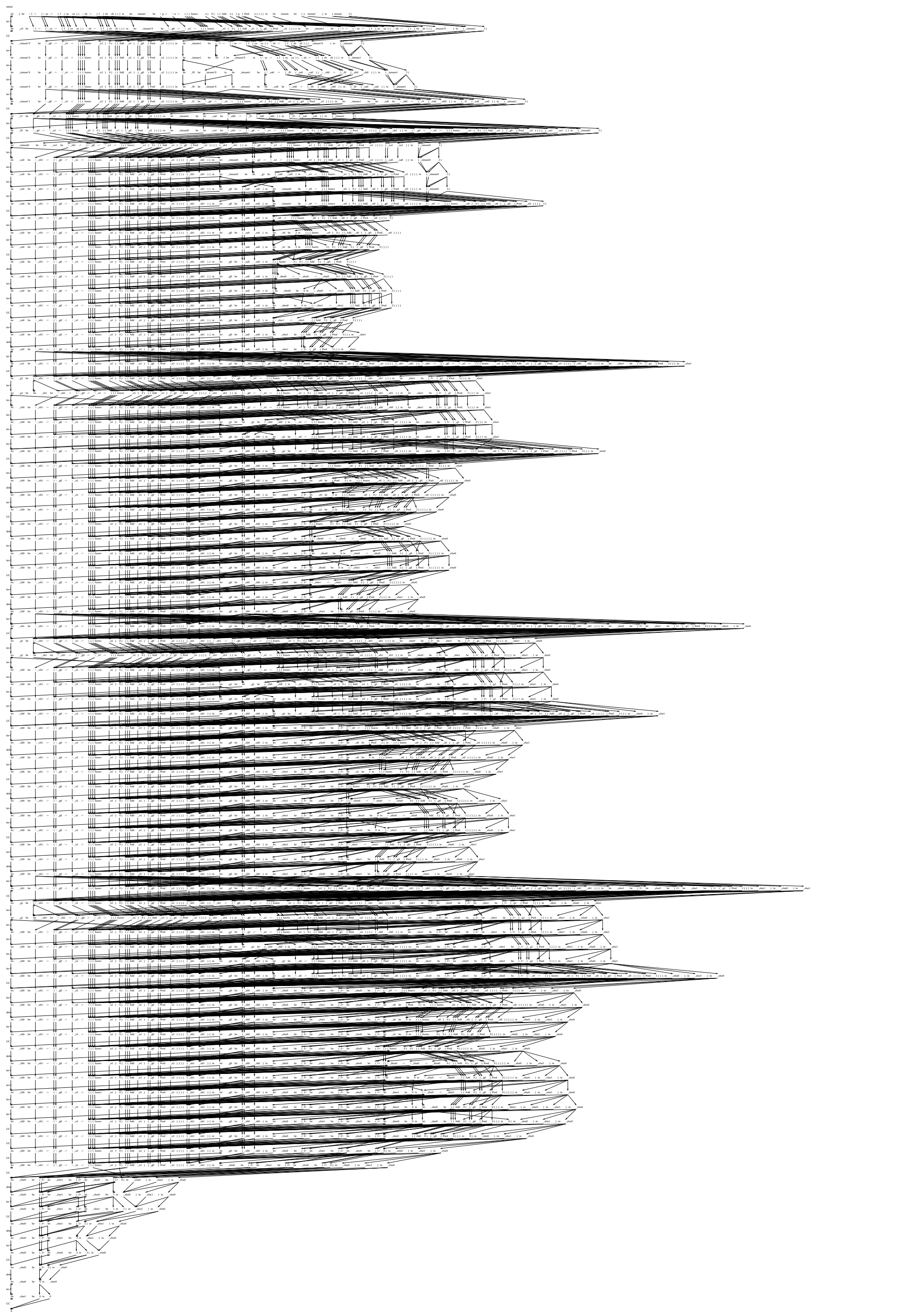

Figure 4.4: The complete trace of `intsum 3`, reduced.

However, it cannot be denied that this figure is somewhat confusing, not to mention excessively large! It must be stressed that this output is merely a prototype viewer, dictated more by ease of coding than anything else. *Most* (but not all) of the information supplied by the `LetTrace` system core is displayed in this figure; however there are much better ways of visualising this data. This is discussed further in Section 5.4. We merely note here that, for example, the `LetTrace` system internally generates and preserves data structures relating every portion of every term to an explicit location in the source file, and that this information would ideally be used to enable the user to view their own source code, as in a conventional debugger, rather than the terms of Λ_{NEED} displayed in this figure. And instead of arrows representing code motion, some form of animation would seem to be indicated.

However, we believe the `LetTrace` system shows the potential of Ariola, Felleisen, Maraist, Odersky and Wadler's call-by-need calculus λ_{NEED}; and hopefully much of it will form a basis for further work in this area. All the basic information is provided by the system, and the basic work of the evaluator is already there; all that is required is a better visualisation layer.

Chapter 5

Internals

In this chapter we consider a number of interesting details of the implementation of `LetTrace`.

5.1 Renaming and substitution

Conventionally when presenting a calculus, one uses (as we have done) Barendregt's *variable convention* [Bar81]: in a term, all bound variables are chosen to be distinct from all the free variables. This makes the statement of the calculus much cleaner and simpler, because one need not worry about variable capture: no rearrangement of the term can cause a free variable to become bound. All the papers this present work was based on [AF94, MOW94, AFM+95, MOW95] make this assumption.

Unfortunately, this is an unrealistic assumption in practice, for two reasons. The first is that real programs do not in general conform; the second and more important is that literal application of the reduction rules can break the convention. Consider for example the term $(\lambda y.(\lambda x.y\,x))\,(\lambda x.z)$. This term satisfies the variable convention. But two standard reduction steps in λ_{NEED} give us $\mathsf{let}\ y\ \mathsf{be}\ \lambda x.z\ \mathsf{in}\ \lambda x.(\lambda x.z)\,x$ which violates it: the subterm $(\lambda x.z)\,x$ contains x both free and bound.

There are at least two solutions to this problem. Either we preprocess the user's program to ensure the variable convention is adhered to (by renaming variables appropriately) and then repeat this preprocessing step after every reduction step; or alternatively we abandon the convention and rephrase the reduction rules more carefully to avoid variable capture. We choose the latter course.

Recall Figure 2.5. We replace this calculus with the one in Figure 5.1.

We see that LET-I remains the same. There is no way a free variable can

(LET-I)	$(\lambda x.M)\,N \xrightarrow{\text{NEED}}$ let x be N in M	
(LET-V)	let x be V in $C[x] \xrightarrow{\text{NEED}}$ let x' be V in $C[x][x := V]$	
(LET-C)	(let x be L in $M)\,N \xrightarrow{\text{NEED}}$ let x' be L in $(M[x := x']\,N)$	
(LET-A)	let y be (let x be L in M) in N $\xrightarrow{\text{NEED}}$ let x' be L in (let y be $M[x := x']$ in N)	
(LET-G)	let x be M in $N \xrightarrow{\text{NEED}} N$	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 5.1: Call-by-need reduction rules with variable substitution.

be captured by this rule.

LET-V, however, has the potential for variable capture. If the variable x is free in V, then the substitution of V into $C[x]$ will introduce occurrences of x in the scope of the let, and these will incorrectly become bound. To avoid this, we choose a variable x' distinct from all free variables in V, and substitute x' for x in $C[x]$ and in the let. However, since we then perform an all-at-once substitution, replacing all occurrences of x', we can save ourselves time and instead of doing $C[x][x := x'][x' := V]$ we simply do $C[x][x := V]$, and the only remaining occurrence of x' is in the let.

The x' we choose must also be distinct from all free variables in $C[x]$ (except that it is allowed to be x itself if x is free—in this case renaming is unnecessary). This is to avoid the situation in which we replace all occurrences of x by y, say, in $C[x]$, when there is already a y free in $C[x]$. If we do this, then any later substitution for y will also (incorrectly) alter the ys that were free before the renaming. So for LET-V we must choose x' distinct from all free variables in both V and $C[x]$, except that we do not need to exclude x itself if it is free in $C[x]$ unless it is also free in V.

In practice, it is easier to simply exclude $fv(V) \cup fv(C[x])$.

Similarly, in LET-A, if x is free in N it will incorrectly become bound after the rearrangement. Also, if we choose a new x' distinct from all free variables in N but free in M, then we will clash with that variable. So we choose x' distinct from all free variables in both M and N.

Exactly the same reasoning applies to LET-C. And for LET-G it is not difficult to see that there is no danger of variable capture.

Variable renaming as required by the above causes problems. Recall that the intention of this system is to aid in debugging. Random variable renaming (to sequence numbers for example) makes it difficult or impossible for the user to follow what is going on, and so it would be helpful if some assistance was provided to the user.

We chose to achieve this by basing the name of the new variable x' on the name of the old variable x: the variable `foo` is replaced by the first of `_foo0`, `_foo1`, ... which is suitable. If the variable is already of the form `_bar`n, the new variable is chosen from `_bar0`, `_bar1`, ...[1]. This means that logically the 'same' variable retains a similar name throughout the reduction sequence, but we maintain the required distinctness of variables.

5.2 Standard reduction

As discussed in Section 2.3, the notion of standard reduction is the notion of a single, standard order in which reductions will occur. This is ensured by restricting the calculus in certain ways, and when this is done there will be at most one *evaluation context* or subterm to be reduced in any term, with no evaluation context exactly when the term is irreducible.

The standard reduction rules have already been given (Figure 2.6). In this section we discuss how they have been implemented in `LetTrace`.

The calculus we have implemented is slightly different from that described in Figure 2.6. There are two main differences. The first is that we chose to use the all-at-once version of the LET-V rule (see Section 2.5) rather than the one-at-a-time version. This was done primarily for ease of implementation. The second is the inclusion of the garbage collection rule LET-G. This was discussed in Section 2.7: [MOW94, sec. 8] explains that LET-G does not affect the eventual result of the evaluation, despite the loss of standardisation. The other differences, in variable renaming and constants, are detailed in sections of their own: Sections 5.1 and 5.3, respectively.

We decided to split the task of implementation into two. The source (see appendix A) has two functions that deal with reductions: `evctxl` and `reducl`. In both cases, the `-l` suffix (for **let**) indicates λ_{NEED}.[2] `evctxl` generates a list of all possible evaluation contexts, and passes them to `reducl`, which attempts to reduce them. By the Unique Evaluation Context theorem, at most one evaluation context is reducible, and hence the order of their enumeration does not matter.

[1] Perhaps `_bar`n, `_bar`$(n+1)$, ... would have been better.

[2] Having a suffix identifying the reduction system is intentional—this makes it possible to implement multiple reduction systems (call-by-name, call-by-value and call-by-name for example) simultaneously in `LetTrace` and compare reductions.

`evctxl` is a simple recursion. It takes a term, and returns a list of evaluation contexts. The list consists of the whole term, followed by (for applications) the evaluation contexts of the functor and for the argument if the functor is constant, or (for **let**s) the evaluation contexts of the body and if the variable appears in the hole of an evaluation context in the body, the evaluation contexts of the variable binding.

`reducl` simply lists the rules of the standardised restriction of the calculus. It attempts to reduce the outermost part of the term it is passed, returning the transformed term and the rule applied if it succeeds, and a flag if it fails.

It can be seen how the definition of `evctxl` matches closely the definition of ECONTEXTS, and the definition of `reducl` matches closely the standard reduction rules of the calculus. This is intentional—it should be apparent how to implement different variants of the calculus, merely by changing these two functions (and giving them new suffixes) to match the new rules.

These two functions are tied together by a third function, `stdredl`. This function is only tagged `-l` because it has calls to `evctxl` and `reducl` hard-coded; it itself contains no λ_{NEED}-specific code.

`stdredl` takes a term, and returns a quadruple $\langle r, E[M], E, E[N]\rangle$, where r is the rule applied or a flag indicating no reduction was possible, $E[M]$ is the original term, E is the evaluation context used, and $E[N]$ is the resulting term. `stdredl` is a list comprehension on the list generated by `evctxl`, filtered by trying `reducl` on each evaluation context. The first (in fact, the only) item in this list is chosen, if it exists.

5.3 Constants

Maraist, Odersky and Wadler have suggested how to add constants to λ_{NEED}, and we presented this in Section 2.6. This extension handles constructors of arbitrary arity, and primitive operators operating on general VALUES (i.e. variables, abstractions or constructed elements). `LetTrace` implements only a subset of these: constants (nullary constructors), and primitives operating on constants, and in fact places them both in the same class, CONSTANTS.

The calculus in Section 2.6 contains two rules, δ-V and δ-A. `LetTrace` implements a single rule, DELTA (see Figure 5.2). This single rule, along with the modification to ECONTEXTS allowing a primitive's argument to be evaluated, is sufficient to allow `LetTrace`'s simple subset of constructors and operators to work correctly.[3]

[3]Our DELTA is basically δ-V; the reason we don't need δ-A is that our only constructors are nullary, and so a constructed element can have no free variables. Hence the **let**-

Syntactic domains:

Constants	k, k'
Values	$V, W ::= x \mid \lambda x.M \mid k$
Terms	$L, M, N ::= V \mid M\,N \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N$
EContexts	$E, E' ::= [\,] \mid E\,M \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ E$
	$\mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ E'[x] \mid k\,E$

Reduction rules: All the rules of λ_{NEED} (Figure 5.1), with the addition of:

$$\text{Delta} \quad k\,k' \xmapsto{\text{NEED}} \delta(k, k') \quad \text{if } \delta(k, k') \text{ defined.}$$

Figure 5.2: `LetTrace` reduction rules for constants.

5.4 Viewing the trace

In Section 5.6 we discuss the information `LetTrace` keeps on the source location of every subterm in the term being reduced. `LetTrace` also knows the term, the evaluation context used (i.e. the redex that was reduced) and the rule used to reduce it. All this information is output in the form of a list of quadruples, as described briefly in Section 5.2 and more fully in the listing of the function `evaluate`, Appendix A.

While these quadruples contain all the information we want from the tracer, they are not in a particularly user-friendly format. Even though this project was intended to be merely a proof of concept rather than a production system, this format was not considered adequate. Instead, we decided to process this into a static figure displaying the sequence of reductions (loosely in the style of [BW88, e.g. p. 5]).

After considering a number of options, we decided to use XFig to handle our output. XFig is freely available, has a simple ASCII file format that is easily generated mechanically, and has sufficient graphical capability for our purposes. It also has scroll bars in its user interface: we briefly considered PostScript and other paper-based options; however the size of the output precluded these, since the average trace is much to big to fit on a single sheet of paper (visibly, at least: see Figure 4.4). The interactive interface of XFig

expression in the left-hand side of δ-A will be garbage-collected out of the way (Maraist, Odersky and Wadler [MOW95, sec. 8] point out the necessity for retaining Let-G here; this is additional support for its inclusion).

allows the user to scroll around their trace and zoom in on points of interest, and even clip out parts of the trace to print on paper if they so choose.

For comparison, the `LetTrace` code (Appendix A) still contains the function `evaluatel`, which attempts to pretty-print a reduction in ASCII; typically on an 80-column terminal a single term will wrap around three times or more, rendering this output unusable.

Instead, there is a module `genfig` (Appendix D) which generates XFig code from the quadruple list emitted by `evaluate`. Details can be found in the listing; basically we define a function `evaltofig` that takes the list and returns a list of XFig objects; this is then converted to an XFig file by `writefigfile`.

The figure is built up by placing the initial term on the page and remembering the position of every subterm, then repeatedly placing the next term, remembering the position of every subterm, and drawing arrows between the terms. The positioning of the arrows is determined by using the subterm range information as tags, and matching these. Subterms with the same range information will have arrows drawn between them. The fact that subterm range information is not quite unique explains why the output (e.g. Figure 4.3) contains more arrows than would be suggested by Figure 5.4.

Evaluation contexts are indicated by a small box under the left end of the context; it was originally intended that this box extend the full length of the evaluation context. However this is a non-trivial exercise, as not enough information is at present stored to compute the right-hand edge coordinate of the box. Problems such as these convinced us that XFig was not the best way to proceed. Some other ideas for visualising the output are detailed in section 6.1; MetaPost would seem to be the next step on from XFig.

5.5 Reading Gofer syntax

To be of any use, it is important that the system accept a conventional syntax, rather than requiring direct entry of lambda calculus terms. We chose to use a subset of Gofer syntax.

The subset chosen is described more precisely in the source itself (appendices B and C); the grammar is given in Figure 5.3. It can be seen that neither pattern matching nor arguments are supported; all arguments have to be coded explicitly with lambda expressions. Constants are implemented differently from Gofer: integers can be written literally, but other constants are written as identifiers with initial capital letters. The supported constants are `Succ`, `Pred`, `Iszero` and `Add`.

$\langle \mathit{Program} \rangle$	::=	`{` $\langle \mathit{Definitions} \rangle$	
$\langle \mathit{Definitions} \rangle$	::=	$\langle \mathit{Definition} \rangle$ `;` $\langle \mathit{Definitions} \rangle$	
	|	$\langle \mathit{Definition} \rangle$ `}`	
$\langle \mathit{Definition} \rangle$	::=	$\langle \mathit{identifier} \rangle$ `=` $\langle \mathit{Term} \rangle$	
$\langle \mathit{Term} \rangle$	::=	$\langle \mathit{identifier} \rangle$	(variable)
	|	$\langle \mathit{Identifier} \rangle$	(constant)
	|	$\langle \mathit{intConstant} \rangle$	(integer constant)
	|	`\` $\langle \mathit{identifier} \rangle$ `->` $\langle \mathit{Term} \rangle$	(abstraction)
	|	$\langle \mathit{Term} \rangle$ $\langle \mathit{Term} \rangle$	(application)
	|	`(` $\langle \mathit{Term} \rangle$ `)`	(parentheses)

Notes

- Function application associates to the left; i.e. `f g h` is parsed as `((f g) h)`, not `(f (g h))`.
- An initial-lowercase identifier `foo` is parsed as an $\langle \mathit{identifier} \rangle$, i.e. a variable; an initial-uppercase identifier `Foo` is parsed as an $\langle \mathit{Identifier} \rangle$, i.e. a constant.
- The recognised $\langle \mathit{Identifier} \rangle$s are `Succ`, `Pred`, `Iszero` and `Add`, representing the functions successor $(\lambda x.x + 1)$, predecessor $(\lambda x.x - 1)$, zero test $(\lambda x.\lambda y.\lambda z.\mathsf{if}\ x = 0\ \mathsf{then}\ y\ \mathsf{else}\ z)$ and addition $(\lambda x.\lambda y.x + y)$, respectively.

Figure 5.3: The grammar recognised by `LetTrace`.

We split the task into two: the lexer (appendix B) and the parser (appendix C).

The lexer is based on that supplied with the Gofer system as `Demos/Utils/lexer.gs`. This original file was a great help, as it already contained most of the code required to lexically analyse Gofer source (along with a minimal parser which we did not use).

However, some modifications were required. First, we wanted `LetTrace` to support the weak form of literate programming supported by the Gofer system. Secondly, we needed to tokenise the output (the original `lexer.gs` returned a list of strings; we wanted a list of tokens). Thirdly, we needed to keep information on the position of each token in the source file. This information is used later by the lexer to apply the 'layout rule' (see below), and is also used by the rest of the `LetTrace` system to keep track of the relationship between the lambda term and the source code (see Section 5.6). And finally, we needed to implement the 'layout rule' that allows the programmer to dispense with many braces and indicate scope by layout instead.

- Literate programming was relatively easy to implement. We simply wrote a function `tangle`, which takes a literate script and returns an 'illiterate' one, in which the comment lines have been blanked out and the `>` characters have been replaced by spaces. If the rules about adjacency of code and comment have been violated, an error is reported.

  We could have handled this in the lexer itself (all the position information required is available), but we decided it would be easier and less prone to error if we separated it off into a separate function rather than hacking `lex`.

- To simplify the parser, we decided to tokenise the source within the lexer; i.e. we required the lexer to return a list of tokens, rather than a list of strings. After defining a type `Token`, based on appendix A of the Gofer documentation [Jon91], we wrote the function `tokenise` which takes a list of strings and maps it to a list of tokens. The conversion is fairly straightforward—the only mildly complex parts were the integer and real number conversion, and the recognition of the various character escapes.

- Token position was handled by introducing a new 'positioned string' datatype, `PosdString`, which is a string tagged with a row and column in the source file. The lexer was modified to take a positioned string and return positioned strings; the row and column information is carried through.

The lexer operates by returning the first token found, along with the remainder of the input. Care was taken to position the remainder appropriately further on from the original, by adding the length of the token to the column number. When a newline is read, the row is incremented by one and the column returns to 1. When a tab is read, the function `nextTab` computes the next tab stop.

Another datatype, `PosdToken`, was also introduced, being a token tagged with a row and column. This is handled in the obvious way.

- The 'layout rule' allows the programmer to dispense with many braces and indicate scope by layout instead. The rules according the Gofer documentation [Jon91, ch. 13] are as follows:

    1. An opening brace `{` is inserted in front of the first token at the beginning of a file or following one of the keywords `where`, `let` or `of`, unless that token is itself an opening brace.
    2. A `;` token is inserted in front of the first token in any subsequent line with exactly the same indentation as the token in front of which the opening brace was inserted.
    3. The layout rule ends and a `}` token is inserted in front of the first token in a subsequent line whose indentation is strictly less than that of the token in front of which the opening brace was inserted.
    4. A closing brace `}` will also be inserted at any point where an otherwise unexpected token is encountered. This part of the rule makes it possible to use expressions such as:

       ```
       let a = fact 12 in a+a
       ```

       without needing to use the layout characters explicitly as in:

       ```
       let {a = fact 12} in a+a
       ```

    5. Lines containing only whitespace (blanks and tabs) and comments do not affect the use of the layout rule.
    6. For the purposes of determining the indentation of each line in a file, tab stops are assumed to be placed every 8 characters, with the leftmost tab stop in column 9. Each tab character inserts one or more spaces as necessary to move to the next tab stop.
    7. The indentation of the end of file token is zero.

We implement all of these except for rule 4. The reason this rule cannot easily be implemented is that it relies on the recognition of an 'unexpected token'. But this is the lexer, and lexer knows nothing of which tokens the parser is or is not expecting. Hence we cannot implement this rule entirely within the lexer.

However, even implementing only *most* of the layout rule is very helpful to the user.

If the lexer finds no valid parse, or more than one parse, an error is returned.

5.6 Relating the trace to the program source

Conventional debuggers for imperative languages offer 'source-level debugging', where the user can single-step through the execution of the code, and observe not assembler or some other internal representation but their own original source. We would like to allow something analogous within our system.

Section 5.5 describes how we associate a range in the source file with every subtree of the term. But how can we preserve this information through reductions?

The answer is to consider the reductions possible, and work out for each one what happens to each code fragment or subterm. Figure 5.4 shows this. Each subterm in the resulting term has to take a value from somewhere in the original redex; in most cases it is obvious which subterm corresponds to which, but in some cases there is no obvious subterm to match to. In these cases, we pick the entire redex, as the narrowest possible subterm including all the source involved in the new subterm. Subterms outside the redex remain unchanged as expected.

Implementing all this means that at all times, every subterm can be related to a specific portion of the user's source code. In the present system (see Section 5.4), this information is used only to trace the motion of subterms within the term as reduction proceeds; ultimately, however, it should be used directly as a basis for a more user-friendly and interactive interface. Two possibilities spring to mind:

- The user could be presented with the current Λ_{NEED} term, and when a subterm is double-clicked, their source window could open with the corresponding range highlighted.

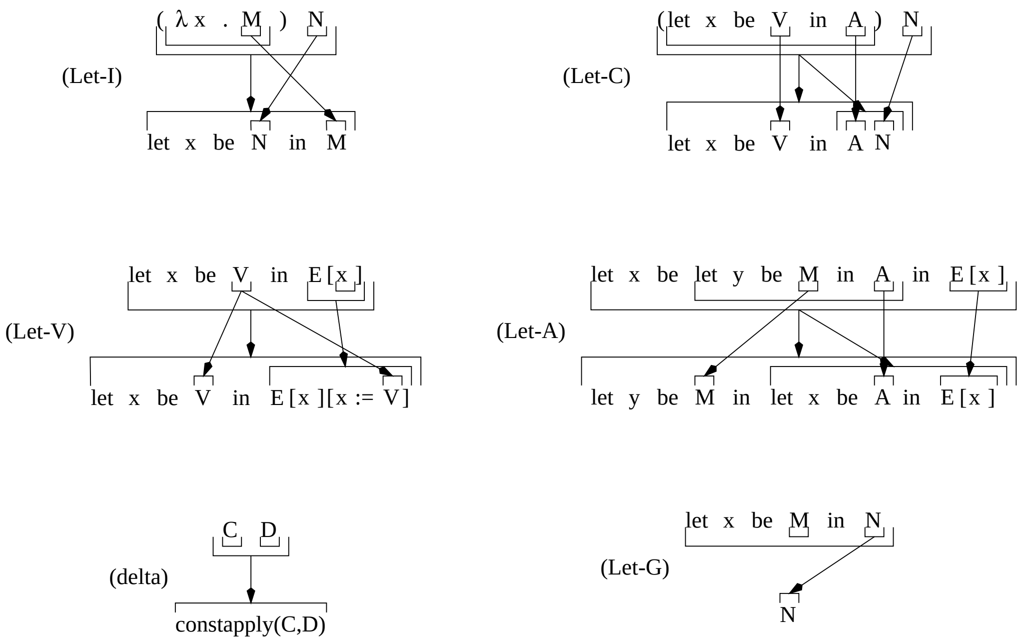

Figure 5.4: The behaviour of subterms during reductions.

- Even better, the user could be presented with a screen comprised largely of their own source, but transformed where necessary into a source-level reflection of subterms of the reduced Λ_{NEED} term.

These would both be dramatic improvements over the current system, and would make use of the information that is currently not being used to its full potential.

5.7 Handling environments

As can be seen from the grammar (Figure 5.3), a user's program consists of a list of definitions, each of which binds an identifier to a term. In Gofer, the semantics of the program says that each term is in an environment containing its own identifier along with all the identifiers for all other definitions at this (top) level. This means that implicit recursion is possible: both the self-recursive function

```
fact = \n -> (Iszero n) 1 (times n (fact (Pred n)))
```

and the mutually recursive functions

```
even = \n -> (Iszero n) 1 (odd (Pred n))
```

```
odd = \n -> (Iszero n) 0 (even (Pred n))
```

are legal.

Unfortunately, the call-by-need calculus we are using in this paper cannot handle this. A binding is represented by a let, and the scope of the let is only the *body* of the let: not the binding, and not the bindings or bodies of any other lets outside it. What we need is what is elsewhere called a letrec, with the semantics as described above. A letrec would bind a list of identifiers to a list of terms, with the scope of each identifier being the body *and every binding.* In other words, letrec x_1 be B_1, x_2 be B_2, ... in M binds x_1 to B_1 and x_2 to B_2, in M, B_1 and B_2. (note M represents the term the user enters and wishes to be reduced in the given environment).

So our examples above could be expressed as

$$\mathsf{letrec}\, \mathit{fact}\ \mathsf{be}\ \lambda n.(\mathrm{Iszero}\, n)\, 1\, (\mathit{times}\, n\, (\mathit{fact}\, (\mathrm{Pred}\, n)))\ \mathsf{in}\ \ldots$$

and

$$\begin{aligned} \mathsf{letrec}\quad & \mathit{even}\ \mathsf{be}\ \lambda n.(\mathrm{Iszero}\, n)\, 1\, (\mathit{odd}\, (\mathrm{Pred}\, n)),\\ & \mathit{odd}\ \mathsf{be}\ \lambda n.(\mathrm{Iszero}\, n)\, 0\, (\mathit{even}\, (\mathrm{Pred}\, n))\mathsf{in}\ldots. \end{aligned}$$

This is the ideal, and is potential future work. But for the present we needed an appropriate representation of the source program as a Λ_{NEED} term. We chose to represent

$$\mathsf{letrec}\ x_1\ \mathsf{be}\ B_1,\ x_2\ \mathsf{be}\ B_2,\ \ldots\ \mathsf{in}\ M$$

as

$$\mathsf{let}\ x_1\ \mathsf{be}\ B_1\ \mathsf{in}\ \mathsf{let}\ x_2\ \mathsf{be}\ B_2\ \mathsf{in}\ \ldots M.$$

This representation allows neither self-recursion (as in `fact`) nor mutual recursion (as in `even` and `odd`); but it does allow referring to other terms, *as long as there are no forward references.* This handles many situations; where recursion is required we must use an explicit recursion operator such as the fixpoint operator Y, as we did in the `intsum` example (Figure 4.1) (naming the operator `y`). The encoding of the environment and user-entered term in this manner is handled by the function `convert`; see Appendix C.

Chapter 6

The Future

There are a number of areas deserving of further study. This chapter attempts to list a few of these.

6.1 Visualisation

As has already been mentioned, the output of `LetTrace` leaves something to be desired. There are a number of techniques by which it could be improved.

One suggestion, due to Jeremy Gibbons, is using a 'ladder notation' style for the reduction. An example of this is given in Figure 6.1. Here we avoid rewriting parts of the term that do not change, and the parts that *do* change are written below each other. This means that instead of a figure full of extraneous clutter, we get a sparse figure containing only the interesting parts (at least in theory). However, while this technique avoids rewriting anything outside the redex, it is not clear how to avoid rewriting subterms *within* a redex.

The environment-wrapping system mentioned in Section 5.7 means the term is initially huge, and it is several garbage collection steps until the unused terms are cleared away. Furthermore, the definitions are superfluous until they are used, which may be many reductions away. It would seem like a good idea to find some way of 'tucking away' the outer environment until it is needed. Perhaps a term like

$$\mathsf{let}\ x_1\ \mathsf{be}\ B_1\ \mathsf{in}\ \mathsf{let}\ x_2\ \mathsf{be}\ B_2\ \mathsf{in}\ \ldots M$$

could be represented as

$$[M]_{x_1,x_2,\ldots}$$

until the values of x_1, x_2, ... are required by the reduction of (subterms of) M.

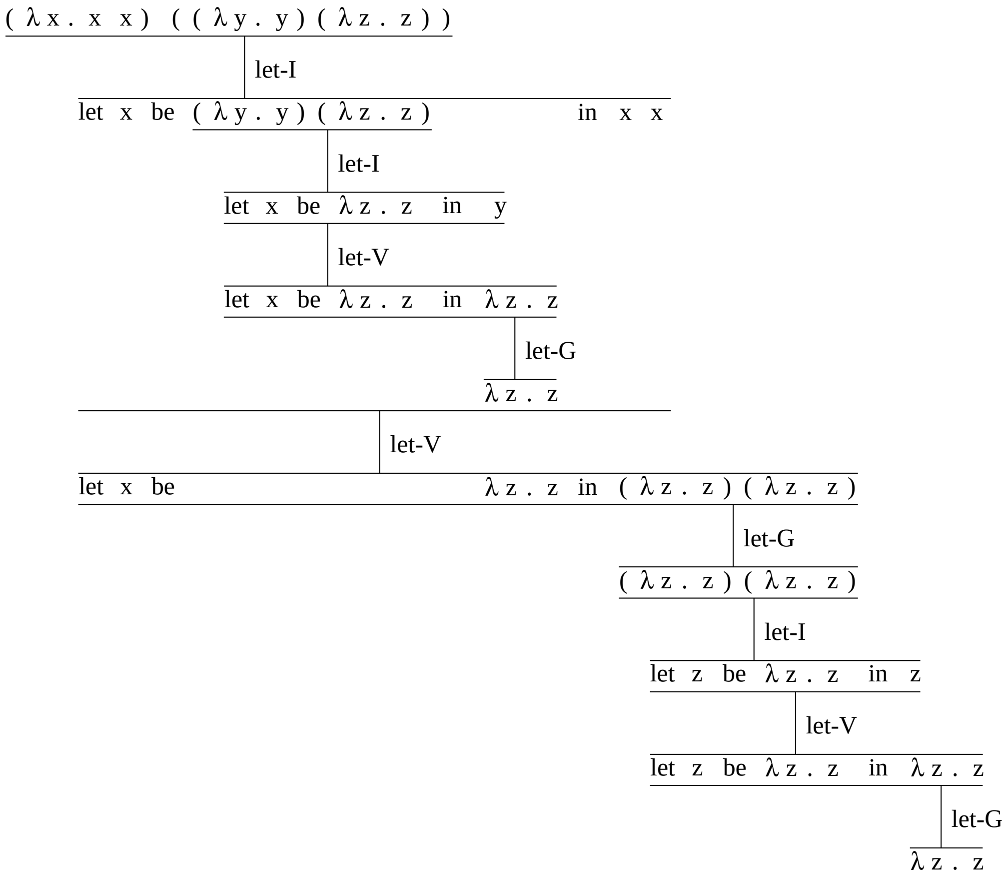

Figure 6.1: Ladder notation.

There should be a way to use the source position information; perhaps the original source could be written near the Λ_{NEED} terms, showing what is happening to the user's program. At present, this information is used only to track the motion of subterms.

Ideally, with any notation we use it would be possible to fit a typical reduction sequence onto several sheets of A4 paper in portrait format; i.e. no term should be wider than the width of a sheet. This is obviously not the case for our current XFig system.

Even coding the current XFig trace visualisation system was quite tricky. With any more complex scheme, arranging for arrows to meet up appropriately, everything else to line up and so on would become an impossible nightmare. Jeremy Gibbons suggested the use of MetaPost, a program that generates PostScript figures from source written in a declarative language optimised for figure construction. MetaPost is based on MetaFont, the system designed by Don Knuth to create characters for TEX, and is described in detail in [Hob]. MetaPost allows the visualisation system to say merely 'this arrow should point to this object', without requiring the system (as at present) to figure out exactly where 'this object' is on the page before drawing the arrow.

From our experience with writing `genfig.lgs`, the module that generates the XFig source, we would recommend *against* implementing the visualisation system in Gofer. It seems difficult to move the position information around the system and manipulate it efficiently in a pure functional language. Also (at least with version 2.30) using I/O in Gofer has severe efficiency problems: it is not possible to append to a file without opening it, appending to it and closing it, and so a string containing the entire file must first be built and then written out. The building of this string by appending shorter strings is inefficient, as the append function must traverse the entire left string: hence the $O(1)$ 'generate the next line of the figure' becomes $O(n)$ 'generate it and append it to the current output string'; the whole algorithm becomes $O(n^2)$. `LetTrace` is quite capable of outputting all the raw information necessary; it would probably be best to then post-process this with, say, a C++ program.

All these ideas may be heading in the wrong direction. A more useful visual representation of what is going on may well involve an interactive graphical user interface—some form of animation system. This would show the last term and the next term, to which it reduces, dynamically, and demonstrate by means of some animation how one reduces to the other and where all the subterms move to. On request by the user, the display would step to the next reduction step; a reverse step could also be requested. The resulting display could well resemble looking through a narrow horizontal window onto one of the systems described above; however it would have the advantage of being much more usable.

With an interactive system, alternative solutions for the problem of excess clutter can be explored. Imagine having the 'environment' (i.e., nested lets) residing in a small control which when double-clicked opens up to reveal the environment, or even displays the environment in a separate window. Imagine compressing all subterms not directly involved in the current reduction into small handles, which again when double-clicked expand appropriately. In this way, the system would allow the user to make their own decisions as to what is or is not important in the trace, rather having the system make the decisions for the user and risk getting them wrong.

A possible system in which to implement this would be Cerno-II [FHM94], a visualisation tool for Snart programs written in Snart over Prolog on the Apple Macintosh. `LetTrace` could either be used as is, taking the output from a run of `evaluate` and using it to drive the interactive visualisation tool; or `LetTrace` could be modified to perform the reduction steps on demand—this would obviously be better for debugging non-terminating reductions, as well as being more efficient for most other cases.

6.2 Syntactic sugar

The language on which `LetTrace` is based is almost entirely unsugared—the only 'sugar' construct is the provision for definitions of terms in an environment: sugar for nested lets (or ideally letrecs, see Section 6.4). However, real functional languages such as Gofer have a large amount of sugar added: so much so that the relationship of the average Gofer program to Church's lambda calculus is not at all obvious to the uninitiated.

This means that to be useful, a tracing system must be able to deal with sugar. There are two issues here: first, the system must know how to translate the sugar into Λ_{NEED}; and second, the system must then be able to relate its internal Λ_{NEED} terms to what the user originally wrote in their program: the system must be able to convert back from Λ_{NEED} to the original sugared constructs.

The first issue is fairly straightforward. The translations are obviously known to compiler writers, and are in fact widely known even outside this group. For example, in Bird and Wadler's introductory textbook for functional languages [BW88, p. 63] they present the translation rules for list comprehension in terms of the straightforward functions *map* and *filter*. So we could without any serious difficulty read in Gofer programs and translate their sugar into Λ_{NEED} terms.

The second issue is unfortunately nowhere near as easy. Although we have been unable to investigate it, we believe the solution is our system of maintaining source positions along with every subterm in the code. From these source positions, we should be able to recreate the source code (i.e. the sugared construct) corresponding to the subterm simply by displaying the indicated section of the file! Flags would probably be necessary to detect if significant changes had occurred inside the subterm that would cause it no longer to match the source; however we believe this technique is viable.

But there is another problem. This technique is no longer applicable after the translation of the construct has been used in a reduction. Suddenly the user will be presented with a bare Λ_{NEED} term they have never seen before, with the system claiming they wrote it. Understandably, they are likely to complain. The solution to this would seem to entail some sort of system to recognise subterm patterns that can be reduced to a sugared form, and to present these to the user. This is a much harder problem.

Alternatively, the sugared constructs can be added to the calculus. But given the size of the proofs for λ_{NEED} as it stands (see, for example, [AF94]) *without* any more constructs than absolutely necessary, this would seem not to be a viable alternative.

6.3 Source-level tracing

We have already (Section 5.6) mentioned the correlation of the user's source to the internal `LetTrace` terms, and in Sections 6.1 and 6.2 we discussed some possible uses for the information. Another idea, relating to the animation system described in Section 6.1, would be to enable the user to double-click on the viewed term and have a window come up with the corresponding source highlighted. This is possible now with the information provided with the `LetTrace` system. It is possible that this could be extended to eventually provide a system that in some useful way is analogous to the conventional imperative source-level debuggers mentioned in Chapter 1.

6.4 Recursion

As noted in Section 5.7, the let-expression does not correspond to what we need to handle environments correctly, nor to constructs such as `where` clauses in Gofer. What is needed here is a way to model implicit recursion: a construct that binds a list of identifiers to a list of bindings, with the scope of the identifiers *including the bindings* (even the identifiers' own bindings). This is commonly known as a letrec ('recursive let').

In their papers [AF94, AFM+95], Ariola and Felleisen describe an extension of λ_{NEED} to include a letrec construct. It is not clear whether they have completed the proof of the soundness and completeness of this calculus. It would be an obvious extension to modify `LetTrace` to use these rules rather than those of λ_{NEED}, resulting in a system that accurately modelled the true behaviour of environments and `where` clauses. The only reason this has not been done is the rather ugly appearance of the rules as given in [AF94] and [AFM+95].

6.5 General constructors

In Section 5.3 we described how `LetTrace` implements a small subset of constructors and primitive operators: constants and primitives operating on constants. It would be useful and presumably straightforward to extend `LetTrace` with the general constructors and primitives described in Section 2.6.

6.6 Types

Another feature of many real functional languages including Gofer not implemented in `LetTrace` is typing. The Λ_{NEED} calculus is untyped. Maraist, Odersky and Wadler [MOW95, pp. 26–27] discuss this briefly in connection with the *linear* lambda calculus of Girard [Gir87], in [MOTW95]. Note that with types, at least under the standard Hindley-Milner type inference system, it is not possible to express the fixed-point combinator Y we use for explicit recursion (as in the example in Chapter 4). This means that we *must* implement letrecs in a system with types.

6.7 Tidying up

As has been mentioned elsewhere, there is some tidying up of this implementation to be done. Specifically, in Section 5.2 we noted that the calculus implemented by `LetTrace` is in fact slightly different from λ_{NEED}; either this could be fixed or the differences more convincingly explained away. Also, constructors and primitives have not been implemented in the full generality of the calculus in Section 2.6. This would be useful.

6.8 Relationship to existing implementations

We have constructed, as we said in Chapter 3, a complete stand-alone system for tracing the execution of (ultimately all) Gofer code. But there are two obvious problems with a stand-alone system. The first problem is the duplication of effort. Every language feature of Gofer is implemented in Gofer already; we must re-implement it in `LetTrace`. The second problem is related; it is that with two distinct systems we can never be entirely sure that they perform identically. How do we know the behaviour simulated by `LetTrace` is indeed the same as the actual behaviour of the same program when executed on Gofer version 2.30, say?

From these concerns, it would seem worthwhile investigating means of integrating this debugging system with the existing implementation of Gofer. This would involve digging around the internal data structures of Gofer, which are presented reasonably clearly in Jones' paper [Jon94]. It is not clear to the author at this time how to relate these to the call-by-need lambda calculus λ_{NEED} presented in this paper, except that if λ_{NEED} really does accurately model the behaviour of languages like Gofer, it should be possible (at least in theory).

Acknowledgements

Chapters 1 and 2 draw heavily from [GW95], which was largely written by Jeremy Gibbons. Philip Wadler's presentation at CATS'94 was the major inspiration for this project. The original idea for the project, and an initial core of `LetTrace` (since greatly modified), are due to Jeremy Gibbons, who has also supervised my work. Thanks are also due to the Department of Computer Science of the University of Auckland.

Bibliography

[Abr90] Samson Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[AF94] 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.

[AFM+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Principles of Programming Languages*, 1995.

[AJ89] Lennart Augustsson and Thomas Johnsson. The Chalmers Lazy-ML compiler. *Computer Journal*, 32(2):127–141, 1989.

[Bar81] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1981.

[BW88] Richard S. Bird and Philip L. Wadler. *An Introduction to Functional Programming*. Prentice-Hall, 1988.

[CF90] E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *Proc. ACM Conference on Principles of Programming Languages*, 1990.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 2(33):346–366, 1932 and 2(34):839–864, 1933.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.

[FF87] 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.

[FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 1992.

[FHM94] S. Fenwick, J. Hosking, and W. Mugridge. Cerno-II: A program visualisation system. Computer Science Report 87, Department of Computer Science, University of Auckland, February 1994.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Sciences*, pages 1–102, 1987.

[GW95] Jeremy Gibbons and Keith Wansbrough. Tracing lazy functional languages. Submitted to Computing: the Australian Theory Symposium 1996, August 1995.

[Hob] John D. Hobby. A user's manual for MetaPost. Computing Science Technical Report 162, AT&T Bell Laboratories.

[Jon91] Mark P. Jones. *Gofer Functional Programming Environment, Version 2.20.* `ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer`, September 1991.

[Jon94] Mark P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Department of Computer Science, Yale University, May 1994.

[MOTW95] 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.

[MOW94] 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.

[MOW95] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. Submitted to the Journal of Functional Programming, 1995.

[OLT94] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, (7):57–81, 1994.

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

[Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

[Wad94] Philip Wadler. A call-by-need lambda calculus. Presentation at CATS'94, Sydney, December 1994.

Appendix A

`lambda.lgs`: The evaluator

This file is `lambda.lgs`, definitions for call-by-need lambda calculus, from Gordon Plotkin's TCS 1975 paper and Ariola, Felleisen, Maraist, Odersky, Wadler, *A Call-By-Need Lambda Calculus*, POPL 1995.

Modified by Keith Wansbrough 1995-03-20...1995-06-03. New major version from 1995-06-13. Modified by Keith Wansbrough 1995-06-13...1995-08-11. Changed to a TEX-ish format by Keith Wansbrough 1995-09-25...1995-19-26.

History

Originally written by Jeremy Gibbons `jeremy@cs.auckland.ac.nz`.

Worked on for 07.483 Advanced Project in Computer Science, under Jeremy Gibbons, by Keith Wansbrough `kwan01@cs.auckland.ac.nz`.

1995-03-20:	Added `y`, `times2` (buggy `times2`).
1995-03-25:	Added 'let' construct, call-by-need stuff (`-l`)
1995-04-02:	Bug fix. Remove fancy `'difference'`. Tidy `add`.
	Changed to consistent variable names.
	Rewrote `reducel` according to my chart.
1995-04-03:	Added definition of `intsum` using `Y`.
1995-04-05:	Include `reducel` chart as comment. Add reduction tracing.
1995-04-17:	Add logging commands `saveto` and `teelines`.
	Add bindings.
1995-06-03:	Add new version of reduction, using explicit evaluation
	contexts. Augmented `Term` with `Hole`, for internal use.
	Added `Eq` to `Term`, `Rule`, `Constant`. Changed lets to be
	displayed as `be` rather than `=`.
1995-06-13:	Remove bindings. Delete all non-call-by-need stuff.
1995-06-14:	Fix let-V so it doesn't need GC to stop infinite loop.
1995-06-20:	Create new datatype `EvCtx`, separate from `Term`.
	Fix let-V bug. Fix let-A bug.
1995-06-24:	Move `saveto` and `teelines` to `log.lgs`.
1995-06-29:	Comment out example terms.
1995-06-30:	Change variables from `(V Int)` to `(V String)`.
	Changed printing of lambda expressions to match input syntax.
	Start adding generated symbols.
1995-07-11:	Change all occurrences of `freshv` to `genfroms`, i.e.,
	generate sane variable names. Fix another let-A bug.
1995-08-08:	Speedup modification of `evctxl`.
	Add `TrmRnge` definition. Change `reducl`, add `substTTR`.
	Carry all changes right through up to evaluate.
1995-08-11:	Make `Range` and `Position` `Ord` as well as `Eq`.

Known Bugs

None!

To Do

None.

Program

We represent variables by strings.

```
> data Variable = V String
> instance Text Variable
```

```
>    where showsPrec _ (V v) = showString v
> instance Eq Variable where (V x == V y) = (x==y)
> instance Ord Variable where (V x <= V y) = (x<=y)
```

`freshv` is an infinite set of variables.

```
> freshv :: Set Variable
> freshv = [ V ("_" ++ show v) | v <- [0..] ]
```

`genv tag` is an infinite set of variables with the given tag.

```
> genv :: String -> Set Variable
> genv s = [ V ("_" ++ s ++ show v) | v <- [0..] ]
```

`isgen v` is true if `v` is an internal variable, i.e. generated by `genv`.

```
> isgen :: Variable -> Bool
> isgen (V (c:cs)) = c=='_'
```

`gengen v` is an infinite set of variables with the same tag as the `genv`'d variable `v`.

```
> gengen :: Variable -> Set Variable
> gengen (V ('_':cs)) = genv (takeWhile (not . isDigit) cs)
```

`genfrom v` is an infinite set of variables with the same tag as `v` if `v` is `genv`'d, otherwise with the identifier of `v` as the tag.

```
> genfrom :: Variable -> Set Variable
> genfrom v@(V id)  = if isgen v then gengen v else genv id
```

A particular collection of constants. `constapply` is a partial function, and yields only closed terms. Note the behaviour of `Iszero`, generating variables for the two continuations with names based on `then` and `else`.

```
> data Constant = Succ | Pred | Iszero | Add | Plus Int | Delta Int
> instance Text Constant
>    where showsPrec _ Succ = showString "Succ"
>          showsPrec _ Pred = showString "Pred"
>          showsPrec _ Iszero = showString "Iszero"
>          showsPrec _ Add = showString "Add"
>          showsPrec _ (Plus n) = shows n . showString "+"
>          showsPrec _ (Delta n) = shows n
> instance Eq Constant
>    where (==) Succ Succ = True
>          (==) Pred Pred = True
>          (==) Iszero Iszero = True
>          (==) Add Add = True
>          (==) (Plus n) (Plus n') = (n == n')
>          (==) (Delta n) (Delta n') = (n == n')
>          (==) _ _ = False
> constapply Succ (Delta n) = Con (Delta (n+1))
> constapply Pred (Delta (n+1)) = Con (Delta n)
> constapply Iszero (Delta 0) = Abs x (Abs y (Var x))
>    where x = choose (genv "then")
```

```
>         y = choose (genv "else")
> constapply Iszero (Delta (n+1)) = Abs x (Abs y (Var y))
>    where x = choose (genv "then")
>          y = choose (genv "else")
> constapply Add (Delta n) = Con (Plus n)
> constapply (Plus n) (Delta m) = Con (Delta (n+m))
> constapply _ _ = error "Illegal application of constants"
```

`Term`s take one of five forms (including lets for call-by-need).

```
> data Term = Var Variable
>           | Con Constant
>           | Abs Variable Term
>           | App Term Term
>           | Let Variable Term Term
> instance Text Term
>    where showsPrec _ (Var v) = shows v
>          showsPrec _ (Con c) = shows c
>          showsPrec _ (Abs x m) = showString "(\\" . shows x .
>                showString " -> " . shows m . showChar ')'
>          showsPrec _ (App m n) = showChar '(' . shows m .
>                showChar ' ' . shows n . showChar ')'
>          showsPrec _ (Let v m n) = showString "(let " . shows v .
>                showString " be " . shows m . showString " in " .
>                shows n . showChar ')'
> instance Eq Term
>    where (Var v      == Var v')       = (v == v')
>          (Con c      == Con c')       = (c == c')
>          (Abs x m    == Abs x' m')    = (x == x' && m == m')
>          (App m n    == App m' n')    = (m == m' && n == n')
>          (Let v m n  == Let v' m' n') = (v == v' && m == m' && n == n')
>          (_          == _)            = False
```

Discriminators.

```
> isvar (Var _) = True
> isvar _ = False
> iscon (Con _) = True
> iscon _ = False
> isabs (Abs _ _) = True
> isabs _ = False
> isapp (App _ _) = True
> isapp _ = False
> islet (Let _ _ _) = True
> islet _ = False
```

`EvCtx`s (evaluation contexts) are a parallel data structure to `Terms`. Each `EvCtx` applies to a particular `Term` and has a parallel structure, except that eventually an `EvCtx` has an `ECHole` where a `Term` has a subtree. The mapping between `EvCtx`' and `Term`'s constructors should be obvious.

```
> data EvCtx = ECVar | ECCon | ECAbs EvCtx | ECApp EvCtx EvCtx
>               | ECLet EvCtx EvCtx | ECHole
```

`termToEvCtx` converts a term to an `EvCtx` with no hole.

```
> termToEvCtx :: Term -> EvCtx
> termToEvCtx (Var _)      =  ECVar
> termToEvCtx (Con _)      =  ECCon
> termToEvCtx (Abs _ m)    =  ECAbs (termToEvCtx m)
> termToEvCtx (App m n)    =  ECApp (termToEvCtx m) (termToEvCtx n)
> termToEvCtx (Let _ m n)  =  ECLet (termToEvCtx m) (termToEvCtx n)
```

A `Position` is a row and a column in the source file; a `Range` is a start and finish `Position`.

```
> data Position = P Int Int
> data Range = R Position Position

> instance Eq Position
>    where (P v h) == (P v' h') = (v,h) == (v',h')
> instance Ord Position
>    where (P v h) <= (P v' h') = (v,h) <= (v',h')

> instance Eq Range
>    where (R p1 p2) == (R p1' p2') = (p1,p2) == (p1',p2')
> instance Ord Range
>    where (R p1 p2) <= (R p1' p2') = (p1,p2) <= (p1',p2')
```

`TrmRnges` (term ranges) are a parallel data structure to `Terms`. Each `TrmRnge` applies to a particular `Term` and has a parallel structure; it stores the `Range` (i.e. source text location) for each subtree of the `Term`.

```
> data TrmRnge = TRVar Range | TRCon Range | TRAbs TrmRnge Range
>                  | TRApp TrmRnge TrmRnge Range
>                  | TRLet TrmRnge TrmRnge Range

> instance Eq TrmRnge
>    where (TRVar r      == TRVar r')       = (r == r')
>          (TRCon r      == TRCon r')       = (r == r')
>          (TRAbs m r    == TRAbs m' r')    = (m == m' && r == r')
>          (TRApp m n r  == TRApp m' n' r') = (m == m' && n == n' && r == r')
>          (TRLet m n r  == TRLet m' n' r') = (m == m' && n == n' && r == r')
>          (_          == _)            = False
```

Function to extract the range from the root node of a `TrmRnge`:

```
> rootRange :: TrmRnge -> Range
> rootRange (TRVar r) = r
> rootRange (TRCon r) = r
> rootRange (TRAbs _ r) = r
> rootRange (TRApp _ _ r) = r
> rootRange (TRLet _ _ r) = r
```

Function to generate `TrmRnge` from `Term`, at a given range. This is a hack, used by `convert` in `parser.lgs` to generate the term entered by the user. Because all the `rs` are the same, no useful information will be contained in the generated `TrmRnge`.

```
> termToTrmRnge :: Term -> Range -> TrmRnge
> termToTrmRnge (Var _)     r = TRVar r
> termToTrmRnge (Con _)     r = TRCon r
> termToTrmRnge (Abs _ m)   r = TRAbs (termToTrmRnge m r) r
> termToTrmRnge (App m n)   r = TRApp (termToTrmRnge m r) (termToTrmRnge n r) r
> termToTrmRnge (Let _ m n) r = TRLet (termToTrmRnge m r) (termToTrmRnge n r) r
```

Values are variables, constants and abstractions; lambda applications and lets are non-values.

```
> isval x = not ((isapp x) || (islet x))
```

`freevars` returns the set of free variables of a term, $fv(M)$. A variable has itself as a free variable. A constant has no free variables. An abstraction binds one of the body's free variables. An application has the free variables of each of its two subterms. And a let is an application of an abstraction, and so the variable is bound in the body but not in the binding—the free variables are those in the body except the bound one, along with those in the binding.

```
> freevars (Var x) = singleton x
> freevars (Con a) = emptyset
> freevars (Abs x m) = freevars m \\ singleton x
> freevars (App m n) = freevars m `union` freevars n
> freevars (Let x m n) = freevars m `union` (freevars n \\ singleton x)
```

`` n `subst` (x,m) `` substitutes `m` for every free occurrence of `x` in `n`, that is, $n[x := m]$.

```
> (Var y) `subst` (x,m) = m, if x==y
>                       = Var y, otherwise
> (Con a) `subst` (x,m) = Con a
> (App n n') `subst` (x,m) = App (n `subst` (x,m)) (n' `subst` (x,m))
> (Abs y n) `subst` (x,m) = Abs z ((n `subst` (y,Var z)) `subst` (x,m))
>      where z = choose ((genfrom y) `difference`
>                  ((freevars m `union` freevars n) `union` singleton x))
> (Let y n n') `subst` (x,m) = Let w (n `subst` (x,m))
>                                  (( n' `subst` (y,Var w)) `subst` (x,m))
>      where w = choose ((genfrom y) `difference`
>                  ((freevars m `union` freevars n') `union` singleton x))
```

`` n `substTTR (x,m) `` substitutes `m` for every free occurrence of `x` in `n`, that is, $n[x := m]$, where `n` and `m` are pairs `(Term,TrmRnge)`. It is equivalent to `` `subst` ``, but on `TrmRnge`s rather than `Term`s.

```
> substTTR :: (Term,TrmRnge) -> (Variable,(Term,TrmRnge)) -> (Term,TrmRnge)

> (Var y,TRVar r) `substTTR` (x,(m,trm)) = (m,trm), if x==y
>                        = (Var y,TRVar r), otherwise

> (Con a,TRCon r) `substTTR` (x,(m,trm)) = (Con a,TRCon r)

> (App n n',TRApp trn trn' r) `substTTR` (x,(m,trm)) =
>      let (ns,trns)   = (n,trn)   `substTTR` (x,(m,trm))
>          (n's,trn's) = (n',trn') `substTTR` (x,(m,trm)) in
>      (App ns n's,TRApp trns trn's r)
```

```
> (Abs y n,TRAbs trn r) `substTTR` (x,(m,trm)) =
>      let (ns,trns) = ((n `subst` (y,Var z)),trn) `substTTR` (x,(m,trm)) in
>        (Abs z ns,TRAbs trns r)
>      where z = choose ((genfrom y) `difference`
>                   ((freevars m `union` freevars n) `union` singleton x))

> (Let y n n',TRLet trn trn' r) `substTTR` (x,(m,trm)) =
>      let (ns,trns)   = (n,trn) `substTTR` (x,(m,trm))
>          (n's,trn's) = ((n' `subst` (y,Var w)),trn') `substTTR` (x,(m,trm)) in
>        (Let w ns n's,TRLet trns trn's r)
>      where w = choose ((genfrom y) `difference`
>                   ((freevars m `union` freevars n') `union` singleton x))
```

Now a type for rules (so we can trace which rules have been applied). `Alpha` and `Beta` are for use simulating Λ or other calculi. `Start` and `None` are flags used by various other functions.

```
> data Rule = Start | LetI | LetV | LetC | LetA | GC | Delt
>             | Alpha | Beta | None
> instance Text Rule
>    where showsPrec _ Start = showString "initial"
>          showsPrec _ LetI  = showString "let-I"
>          showsPrec _ LetV  = showString "let-V"
>          showsPrec _ LetC  = showString "let-C"
>          showsPrec _ LetA  = showString "let-A"
>          showsPrec _ GC    = showString "GC"
>          showsPrec _ Delt  = showString "delta"
>          showsPrec _ Alpha = showString "alpha"
>          showsPrec _ Beta  = showString "beta"
>          showsPrec _ None  = showString "<none>"  --never happens
> instance Eq Rule
>    where (==) Start Start = True
>          (==) LetI LetI = True
>          (==) LetV LetV = True
>          (==) LetC LetC = True
>          (==) LetA LetA = True
>          (==) GC GC = True
>          (==) Delt Delt = True
>          (==) Alpha Alpha = True
>          (==) Beta Beta = True
>          (==) None None = True
>          (==) _ _ = False
```

A display routine for evaluation contexts. `showCtx` takes a term and an evaluation context, and displays them.

```
> showCtx (Var v) (ECVar) = shows v
> showCtx (Con c) (ECCon) = shows c
> showCtx (Abs x m) (ECAbs m') = showString "(\\" . shows x
>      . showString " -> " . showCtx m m' . showChar ')'
> showCtx (App m n) (ECApp m' n') = showChar '(' . showCtx m m' . showChar ' '
>      . showCtx n n' . showChar ')'
> showCtx (Let v m n) (ECLet m' n') = showString "(let " . shows v
>      . showString " be " . showCtx m m' . showString " in " . showCtx n n'
>      . showChar ')'
> showCtx m (ECHole) = showString "[" . shows m . showString "]"
```

Implementation of sets (as sorted duplicate-free lists). Only works for `Ordered` element types. Can't use `merge` for union, because we want to remove duplicates.

`xs 'difference' ys` won't work for infinite `ys`.

```
> type Set a = [a]
> emptyset = []
> singleton x = [x]
> [] `union` ys = ys
> xs `union` [] = xs
> (x:xs) `union` (y:ys) = x : (xs `union` (y:ys)), if x<y
>                       = x : (xs `union` ys), if x==y
>                       = y : ((x:xs) `union` ys), if x>y
> xs `difference` ys = xs \\ ys
> x `memberof` [] = False
> x `memberof` (y:ys) = False, if x<y
>                     = True, if x==y
>                     = x `memberof` ys, if x>y
```

`choose` selects an element from the set; `chooses n` selects `n` elements from the set.

```
> chooses n xs = take n xs
> choose (x:xs) = x
```

A generator for evaluation contexts. `evctxl m` is a list of call-by-let evaluation contexts `(e,h)`, where `e` is the context and `h` is the hole.

We have:

- Always, the whole term.
- For applications, the functor (recursively); — and if the functor is constant, the argument.
- For lets, the body; — and if the variable appears in the hole of an evaluation context in the body, the variable binding.

```
> evctxl :: (Term,TrmRnge) -> [(EvCtx,(Term,TrmRnge))]
> evctxl (t,tr) = [(ECHole,(t,tr))] ++ case t of
>        (App m n)   ->
>           let tTECn = termToEvCtx n
>               (TRApp trm trn r) = tr in
>             [((ECApp w tTECn),m') | (w,m') <- evctxl (m,trm)]
>               ++ if iscon m then
>                    let tTECm = termToEvCtx m in
>                      [((ECApp tTECm w),n') | (w,n') <- evctxl (n,trn)]
>                  else
>                    []
>        (Let x m n) ->
>           let tTECm = termToEvCtx m
>               (TRLet trm trn r) = tr in
>             [((ECLet tTECm w),n') | (w,n') <- evctxl (n,trn)]
>               ++ if (Var x) `elem` (map (fst.snd) (evctxl (n,trn))) then
>                    let tTECn = termToEvCtx n in
>                      [((ECLet w tTECn),m') | (w,m') <- evctxl (m,trm)]
>                  else
>                    []
>        otherwise   ->
>           []
```

The function `reducl` performs a single lambda-let reduction on the outermost part of its argument `m`, and returns a pair `(r,n)` where `r` is the rule used and `n` is `m` reduced, or `r`=`None` and `n`=`m`.

```
> reducl :: (Term,TrmRnge) -> (Rule,(Term,TrmRnge))

> reducl (App m n,TRApp trm trn r)
>    | iscon m && iscon n = let (Con a,Con b) = (m,n)
>                               d = constapply a b in
>                             (Delt,(d,termToTrmRnge d r))
>    | isabs m = let (Abs x m') = m   --let-I
>                    (TRAbs trm' rm) = trm in
>                  (LetI,(Let x n m',TRLet trn trm' r))
>    | islet m = let (Let x m' n') = m   --let-C
>                    (TRLet trm' trn' rm) = trm in
>         let y = choose ((genfrom x) `difference`
>                    (freevars n' `union` freevars n)) in
>           (LetC,(Let y m' (App (n' `subst` (x,Var y)) n),
>                  TRLet trm' (TRApp trn' trn r) r))

> reducl (Let x m n,TRLet trm trn r)
>    | not (x `elem` freevars n) = (GC,(n,trn)) --Felleisen and Hieb's GC rule
>                                               --must be listed *first*
>    | isval m && in_hole x (n,trn) =   --let-V, all-at-once version
>         let (n',trn') = ((n,trn) `substTTR` (x,(m,trm))) in
>         let y = choose ((genfrom x) `difference`  -- assumption!  see below
>                   (freevars m `union` freevars n)) in
>           (LetV,(Let y m n',TRLet trm trn' r))
>    | islet m && in_hole x (n,trn) = let (Let y m' n') = m  --let-A
>                                         (TRLet trm' trn' r') = trm in
>         let z = choose ((genfrom y) `difference`
>                   (freevars n' `union` freevars n)) in
>           (LetA,(Let z m' (Let x (n' `subst` (y,Var z)) n),
>                  TRLet trm' (TRLet trn' trn r) r))
>   where in_hole x m = not (null [h | (_,(h,_))<-evctxl m, h == (Var x)])
```

Here the let-V rule assumes that since *all* occurrences of `x` in `n` are replaced with `m`, we don't need to do ``(n `subst` (x,(Var y)))`` to preserve captures of `x` by the `Let x` ....

```
> reducl (m,trm) = (None,(m,trm))
```

The function `stdredl` performs a single standard call-by-let reduction of its argument, and returns `(r,m,w,n)`, where `r` is the rule applied or `None` if no reduction could be performed, `m` is the original term, `w` is the evaluation context used, and `n` is the resulting term. If `r`=`None`, then `n`=`m` and `w`=the entire term.

```
> stdredl :: (Term,TrmRnge) -> (Rule,(Term,TrmRnge),EvCtx,(Term,TrmRnge))
> stdredl m = let reds = [(r,m,w,(fill m w n')) | (w,n) <- evctxl m,
>                                                 (r,n') = reducl n,
>                                                 r /= None]
>             in if null reds then (None,m,ECHole,m) else head reds
```

The function `fill m w n` fills the hole in `m` indicated by `w` with the term `m`.

```
> fill :: (Term,TrmRnge) -> EvCtx -> (Term,TrmRnge) -> (Term,TrmRnge)
> fill _                          ECHole        h
>        =  h
> fill (App m n,TRApp trm trn r)   (ECApp m' n')  h
>        = let (fm,trfm) = fill (m,trm) m' h
>              (fn,trfn) = fill (n,trn) n' h in
>           (App fm fn,TRApp trfm trfn r)
> fill (Abs v m,TRAbs trm r)       (ECAbs m')     h
>        = let (fm,trfm) = fill (m,trm) m' h in
>           (Abs v fm,TRAbs trfm r)
> fill (Let v m n,TRLet trm trn r) (ECLet m' n')  h
>        = let (fm,trfm) = fill (m,trm) m' h
>              (fn,trfn) = fill (n,trn) n' h in
>            (Let v fm fn,TRLet trfm trfn r)
> fill m                          _             _
>        =  m
```

Given a one-step reduction and a term, reduce that term as far as possible and return a list of the reduction steps.

```
> evaluate :: ((Term,TrmRnge) -> (Rule,(Term,TrmRnge),EvCtx,(Term,TrmRnge)))
>          -> (Term,TrmRnge)
>          -> [(Rule,(Term,TrmRnge),EvCtx,(Term,TrmRnge))]
> evaluate r m = eval' r (Start,m,ECHole,m)
>    where eval' r (None,_,_,_) = []
>          eval' r (s,m,w,n) = (s,m,w,n) : eval' r (r n)
```

Do the same as `evaluate`, but restricted to a maximum `n` reductions.

```
> evaluate' :: ((Term,TrmRnge) -> (Rule,(Term,TrmRnge),EvCtx,(Term,TrmRnge)))
>           -> Int
>           -> (Term,TrmRnge)
>           -> [(Rule,(Term,TrmRnge),EvCtx,(Term,TrmRnge))]
> evaluate' r n = take n . evaluate r
```

Pretty-print a single reduction step.

```
> showStep (r,m,w,n) = showCtx (fst m) w "" ++ "\n"
>    ++ "       |==" ++ show r ++ "==>  " ++ showCtx (fst n) w ""
```

Generate and pretty-print a reduction sequence.

```
> evaluatel    = layn . map showStep . evaluate  stdredl
> evaluatel' n = layn . map showStep . evaluate' stdredl n
```

END

Appendix B

`lexer.lgs`: The lexer

This file is `lexer.lgs`, and was originally from the Gofer distribution as `Demos/Utils/lexer.gs`. It is now a lexer for LetTrace source files, which are intended to be a (very restricted) subset of Gofer.

Note that this file requires too many type variables for the PC Gofer version 2.28 to handle. Needs at least the 386Gofer version.

Modified by Keith Wansbrough 1995-06-23...1995-06-26. LaTeX-ified by Keith Wansbrough 1995-09-25...1995-09-26.

History

1995-06-23:	Initial copy.
1995-06-24:	Convert this source file to a literate script. Add `tangle` to (primitively) handle literate scripts. Modify lexer to return position of each token in input file.
1995-06-26:	Modify `tangle` to preserve position information (i.e. make lines blank rather than remove them, and replace initial `>` by ␣ rather than by nothing.) Add `llexall`, a literate lexical analyser (formed by composing the new `tangle` with `lexall`). Patch `lex` in a few places to handle more of Gofer (I hope it is *all* now!) Add tokeniser. Handle tabs correctly. Handle layout rule (except case 4).

Known Bugs

None.

To Do

None.

Program

`PosdString` is a string located at a position: (string,row,col) (where row and col are 1-origin).

```
> type PosdString = (String,Int,Int)
```

`posddropWhile` is like `dropWhile` except that it maintains the source position (row and column).

```
> posddropWhile p ((x:xs),r,c)
>         | p x          = case x of '\n' -> posddropWhile p (xs,r+1,1)
>                                    '\t' -> posddropWhile p (xs,r,nextTab c)
>                                    _    -> posddropWhile p (xs,r,c+1)
>         | otherwise    = ((x:xs),r,c)
>              where nextTab c = (((c-1)/8)+1)*8 + 1
> posddropWhile p ([],r,c) = ([],r,c)
```

`lex` is the lexer. It takes a positioned string and returns a list of alternative parses, each parse being a pair of (positioned token, positioned remainder). This block of code comes from the original `lexer.gs`, with a few modifications to include positions.

```
> lex                   :: PosdString -> [(PosdString,PosdString)]

> lex ("",r,c)                  | c==1       = [(("",r,0),("",r,c))]
>                               | otherwise  = [(("",r+1,0),("",r,c))]
>                                  -- note: EOF always column 0

> lex ((ch:s),r,c) | isSpace ch = lex (posddropWhile isSpace ((ch:s),r,c))

> lex (('-':'-':s),r,c)         = case dropWhile (/= '\n') s of
>                                '\n':t -> lex (t,r+1,1)
>                                _      -> [] -- unterminated end-of-line
>                                             -- comment

> lex (('{':'-':s),r,c)         = lexNest lex (s,r,c+2)
>            where
>              lexNest f (('-':'}':s),r,c) = f (s,r,c+2)
>              lexNest f (('{':'-':s),r,c) = lexNest (lexNest f) (s,r,c+2)
>              lexNest f (('\n':s),r,c)    = lexNest f (s,r+1,1)
>              lexNest f ((_:s),r,c)       = lexNest f (s,r,c+1)
>              lexNest _ ("",_,_)          = [] -- unterminated
>                                                -- nested comment
```

```
> lex (('<':'-':s),r,c)        = [(("<-",r,c),(s,r,c+2))]

> lex (('\'':s),r,c)    = [(('\'':ch++"'",r,c), (t,tr,tc+1))
>                          | ((ch,_,_),('\'':t,tr,tc))  <- lexLitChar (s,r,c+1),
>                                                ch /= "'"                ]

> lex (('"':s),r,c)     =
>          [(('"':str,r,c), (t,tr,tc))
>            | ((str,_,_),(t,tr,tc)) <- lexString (s,r,c+1)]
>                where
>                lexString (('"':s),r,c) = [(("\"",r,c),(s,r,c+1))]
>                lexString (s,r,c) = [((ch++str,r,c), (u,ur,uc))
>                    | ((ch,_,_),(t,tr,tc))  <- lexStrItem (s,r,c),
>                      ((str,_,_),(u,ur,uc)) <- lexString (t,tr,tc)  ]
>
>                lexStrItem (('\\':'&':s),r,c) = [(("\\&",r,c),(s,r,c+2))]
>                lexStrItem (('\\':ch:s),r,c) | isSpace ch
>                    = [(("\\&",r,c),(t,tr,tc+1))
>                             | ('\\':t,tr,tc) <-
>                                       [posddropWhile isSpace (ch:s,r,c+1)]]
>                lexStrItem s           = lexLitChar s

> lex ((c:s),v,h) | isSingle c        = [((([c],v,h),(s,v,h+1))]
>         | isSym1 c    = [((c:sym,v,h),(t,v,h+(length sym)+1))
>                             | (sym,t) <- [span isSym s]]
>         | isAlpha c   = [((c:nam,v,h),(t,v,h+(length nam)+1))
>                             | (nam,t) <- [span isIdChar s]]
>         | isDigit c   = [((c:ds++fe,v,h),(t,v,h+(length ds)+(length fe)+1))
>                             | (ds,s)  <- [span isDigit s],
>                                          (fe,t)  <- lexFracExp s ]
>         | otherwise   = []    -- bad character
>              where
>              isSingle c  =  c `elem` ",;()[]{}_"
>              isSym1 c    =  c `elem` "-~" || isSym c
>              isSym c     =  c `elem` "!@#$%&*+./<=>?\\^|:`"
>              isIdChar c  =  isAlphanum c || c `elem` "_'"
>
>              lexFracExp ('.':c1:s)  | isDigit c1
>                          = [('.':ds++e,u) | (ds,t) <- lexDigits (c1:s),
>                                             (e,u)  <- lexExp t    ]
>              lexFracExp s  = [("",s)]
>
>              lexExp (e:s) | e `elem` "eE"
>                      = [(e:c:ds,u) | (c:t)  <- [s], c `elem` "+-",
>                                                (ds,u) <- lexDigits t] ++
>                        [(e:ds,t)   | (ds,t) <- lexDigits s]
>              lexExp s = [("",s)]

> lexDigits             :: String -> [(String,String)]
> lexDigits             =  nonnull isDigit

> nonnull               :: (Char -> Bool) -> (String -> [(String,String)])
> nonnull p s           =  [(cs,t) | (cs@(_:_),t) <- [span p s]]

> lexLitChar            :: PosdString -> [(PosdString,PosdString)]
> lexLitChar (('\\':s),v,h)   =  [(('\\':esc,v,h), (t,v1,h1))
>                                 | ((esc,_,_),(t,v1,h1)) <- lexEsc (s,v,h+1)]
>       where
>       lexEsc ((c:s),v,h)     | c `elem` "abfnrtv\\\"'" = [((([c],v,h),(s,v,h+1))]
>       lexEsc (('^':c:s),v,h) | c >= '@' && c <= '_'  = [((['^',c],v,h),(s,v,h+2))]
```

```
>       lexEsc (s@(d:_),v,h)   | isDigit d
>                            =  [((ds,v,h),(t,v,h+(length ds)))
>                                          | (ds,t) <- lexDigits s]
>       lexEsc (('o':s),v,h)  =  [(('o':os,v,h), (t,v,h+(length os)+1))
>                                          | (os,t) <- nonnull isOctDigit s]
>       lexEsc (('x':s),v,h)  =  [(('x':xs,v,h), (t,v,h+(length xs)+1))
>                                          | (xs,t) <- nonnull isHexDigit s]
>       lexEsc (s@(c:_),v,h)   | isUpper c
>                        =  case [((mne,v,h),(s',v,h+(length mne)))
>                                          | mne <- "DEL" : asciiTab,
>                                              ([],s') <- [match mne s]      ]
>                           of (pr:_) -> [pr]
>                              []     -> []
>       lexEsc _         =  []
> lexLitChar (('\n':s),v,h)   =  [((['\n'],v,h),(s,v+1,1))]
> lexLitChar ((c:s),v,h)      =  [(([c],v,h),(s,v,h+1))]
> lexLitChar ("",_,_)         =  []

> isOctDigit c  =  c >= '0' && c <= '7'

> isHexDigit c  =  isDigit c || c >= 'A' && c <= 'F'
>                           || c >= 'a' && c <= 'f'

> match                :: (Eq a) => [a] -> [a] -> ([a],[a])
> match (x:xs) (y:ys) | x == y  =  match xs ys
> match xs     ys                    =  (xs,ys)

> asciiTab = ["NUL", "SOH", "STX", "ETX", "EOT", "ENQ", "ACK", "BEL",
>            "BS",  "HT",  "LF",  "VT",  "FF",  "CR",  "SO",  "SI",
>            "DLE", "DC1", "DC2", "DC3", "DC4", "NAK", "SYN", "ETB",
>            "CAN", "EM",  "SUB", "ESC", "FS",  "GS",  "RS",  "US",
>            "SP"]
```

`lexall` returns a list of *all* the tokens in the source string, rather than just the first, and gives an error if parsing is ambiguous or impossible. Recall, the lexer returns `[]` for end-of-file.

```
> lexall :: PosdString -> [PosdString]
> lexall ps@((_:_),_,_)  = case lex ps of
>                      [(a,ys)]              -> a:(lexall ys)
>                      []                    -> error "No parse in lexall"
>                      _                     -> error "Ambiguous parse in lexall"
> lexall ([],_,_)        = []   -- probably not called
```

`tangle` takes a literate script and returns an 'illiterate' script, or throws an error if the literate script is illegal. It assumes the input has been converted to lines already (this can be done with `lines`, and undone with `unlines`). The conversion is done by replacing comment lines with blank lines, and the `>` at the beginning of source lines with ␣.

```
> data LineType = BlankLine | SourceLine | CommentLine
> instance Eq LineType
>    where BlankLine == BlankLine      =  True
>          SourceLine == SourceLine    =  True
>          CommentLine == CommentLine  =  True
>          _ == _                      =  False
```

```
> tangle :: [String] -> [String]
> tangle = check . tag
>    where tag :: [String] -> [(LineType,String)]
>          tag (x:xs) = case x of
>                           ""    -> (BlankLine,x)
>                           '>':t -> (SourceLine,' ':t)
>                           _     -> (CommentLine,x)
>                         : (tag xs)
>          tag [] = []
>
>          check :: [(LineType,String)] -> [String]
>          check ((t1,s1):(t2,s2):xs) =
>                        if (ok t1 t2) then
>                             (if t1==SourceLine then s1 else "")
>                                               : (check ((t2,s2):xs))
>                        else error "Source line adjacent to comment"
>                   where ok SourceLine CommentLine = False
>                         ok CommentLine SourceLine = False
>                         ok _ _ = True
>          check [(t,s)] = [if t==SourceLine then s else ""]
>          check [] = []
```

We also need a tokeniser, which converts the strings returned by `lexall` into tokens. These tokens are based on Appendix A of the Gofer documentation v2.20.

```
> data Token = VarId String | ConId String | VarOp String
>            | ConOp String | IntCon Int | FloCon Float
>            | ChrCon Char | StrCon String | EndOfFile
```

```
> instance Eq Token
>    where     (VarId s)  == (VarId t)  =    s == t
>              (ConId s)  == (ConId t)  =    s == t
>              (VarOp s)  == (VarOp t)  =    s == t
>              (ConOp s)  == (ConOp t)  =    s == t
>              (IntCon s) == (IntCon t) =    s == t
>              (FloCon s) == (FloCon t) =    s == t
>              (ChrCon s) == (ChrCon t) =    s == t
>              (StrCon s) == (StrCon t) =    s == t
>              EndOfFile  == EndOfFile  =    True
>              _          == _          =    False
```

`PosdToken` is parallel to `PosdString`, and is a `Token` located at a row and column.

```
> type PosdToken = (Token,Int,Int)
```

`tokenise` takes a list of positioned strings and returns a list of positioned tokens. The conversion is fairly straightforward.

```
> tokenise :: [PosdString] -> [PosdToken]
> tokenise [] = []
> tokenise ((s,v,h):xs) = (tok1 s,v,h) : tokenise xs
>    where tok1 s@(c:cs)
>              | isLower c    = VarId s
>              | isUpper c    = ConId s
>              | isDigit c    = cvtDigit s
>              | c=='\''      = cvtChar cs
```

```
>             | c=='"'        = cvtString cs
>             | c==':'        = ConOp s
>             | otherwise     = VarOp s
>         tok1 []             = EndOfFile
>         cvtDigit cs = let (n,t) = cvtIntPre 0 cs 10 in
>             case t of
>                  []         -> IntCon n
>                  ('.':ts)   -> let (x,t1) = cvtFracPre (fromInteger n) ts 0.1 10.0
>                                 in case t1 of
>                                   []        -> FloCon x
>                                   (_:ts1)   -> FloCon (cvtExp x ts1 10 10.0)
>                  (_:ts)     -> FloCon (cvtExp (fromInteger n) ts 10 10.0)
>           where cvtIntPre n (c:cs) b
>                       | isDigit c =  cvtIntPre (n*b+(ord c - ord '0')) cs b
>                       | otherwise =  (n,c:cs)
>                 cvtIntPre n [] _  = (n,[])
>                 cvtFracPre x (c:cs) f b
>                       | isDigit c
>                           = cvtFracPre (x+f*fromInteger (ord c - ord '0'))
>                                        cs (f/b) b
>                       | otherwise = (x,c:cs)
>                 cvtFracPre x [] _ _ = (x,[])
>                 cvtExp x cs bn bf
>                       = let (n,[]) = cvtIntPre 0 cs bn
>                            in x * (bf ^ n)
>         cvtChar cs = ChrCon (fst (cvtChar1 cs))
>         cvtString cs = StrCon (cvtString1 cs)
>         cvtChar1 :: String -> (Char,String)
>         cvtChar1 ('\\':c:cs)
>             | c `elem` (map fst charEscTab)
>                       = (assoc c charEscTab,cs)
>             | c == '^'
>                       = (chr (ord (head cs) - ord '@'),tail cs)
>             | isDigit c
>                       = cvtIntEsc 0 (c:cs) "0123456789" 10
>             | c == 'o'
>                       = cvtIntEsc 0 cs "01234567" 8
>             | c == 'x'
>                       = cvtIntEsc 0 cs "0123456789ABCDEF" 16
>             | isUpper c
>                       =  case [(mne,s') | mne <- map fst asciiEscTab,
>                                           ([],s') <- [match mne s]]
>                          of ((mne,s):_) -> (assoc mne asciiEscTab,s)
>                            []           -> (c,cs)
>           where  cvtIntEsc n (c:cs) ds b
>                       = cvtIntEsc (n*b + (assoc (toUpper c) (zip ds [0..])))
>                                   cs ds b
>                  cvtIntEsc n [] cs _ = (chr n,cs)
>                  charEscTab = [ ('a','\a'),('b','\b'),('f','\f'),('n','\n'),
>                                 ('r','\r'),('t','\t'),('v','\v'),('\\','\\'),
>                                 ('\'','\''),('"','"') ]
>                  asciiEscTab = ("DEL",'\DEL') : zip asciiTab (map chr [0..])
>         cvtChar1 (c:cs) = (c,cs)
>         cvtString1 ('"':s) = ""
>         cvtString1 ('\\':'&':s) = cvtString1 s
>         cvtString1 s = let (ch,t) = cvtChar1 s in ch:(cvtString1 t)
```

`assoc` takes a key value a and a list of pairs $\langle a_i, a_i \rangle$ (an *association list*), and returns b_i where $a_i = a$.

```
> assoc :: Eq a => a -> [(a,b)] -> b
> assoc a ((a1,b1):ps)   | a==a1        = b1
>                        | otherwise    = assoc a ps
```

We also need code to implement the layout rule. Here is the layout rule, as expressed in chapter 13 of the Gofer documentation v2.20, Mark P. Jones, 1991:

> The layout rule used in Gofer is the same as that of Haskell, and can be described as follows:
>
> 1. An opening brace { is inserted in front of the first token at the beginning of a file or following one of the keywords `where`, `let` or `of`, unless that token is itself an opening brace.
> 2. A ; token is inserted in front of the first token in any subsequent line with exactly the same indentation as the token in front of which the opening brace was inserted.
> 3. The layout rule ends and a } token is inserted in front of the first token in a subsequent line whose indentation is strictly less than that of the token in front of which the opening brace was inserted.
> 4. A closing brace } will also be inserted at any point where an otherwise unexpected token is encountered. This part of the rule makes it possible to use expressions such as:
>
>    ```
>    let a = fact 12 in a+a
>    ```
>
>   without needing to use the layout characters explicitly as in:
>
>    ```
>    let {a = fact 12} in a+a
>    ```
>
> 5. Lines containing only whitespace (blanks and tabs) and comments do not affect the use of the layout rule.
> 6. For the purposes of determining the indentation of each line in a file, tab stops are assumed to be placed every 8 characters, with the leftmost tab stop in column 9. Each tab character inserts one or more spaces as necessary to move to the next tab stop.
> 7. The indentation of the end of file token is zero.

The implementation below does all of this except for rule 4, which is supposed to insert a } whenever an otherwise unexpected token is encountered. This is not possible without integrating the layout rule with the parser.

```
> layoutrule :: [PosdToken] -> [PosdToken]
> layoutrule [] = []
> layoutrule (x@(t,v,h):xs)
>             = if t /= VarOp "{"
>               then (VarOp "{",v,h) : x : layoutrule1 xs t [h] v
>               else x : layoutrule1 xs t [] v
>    where layoutrule1 (x@(t,v,h):xs) lastTok [] _
>             = if lastTok `elem` [VarId "where", VarId "let", VarId "of"]
>                       && t /= VarOp "{" then
>                  (VarOp "{",v,h) : x : layoutrule1 xs t [h] v
>               else
>                  x : layoutrule1 xs t [] v
>         layoutrule1 (x@(t,v,h):xs) lastTok (lasth:lasths) lastv
>             = if lastTok `elem` [VarId "where", VarId "let", VarId "of"]
>                       && t /= VarOp "{" then
>                  (VarOp "{",v,h) : x : layoutrule1 xs t (h:lasth:lasths) v
>               else if v /= lastv then
>                  if h == lasth then
>                       (VarOp ";",v,h) : x : layoutrule1 xs t (lasth:lasths) v
>                  else if h < lasth then
>                       (VarOp "}",v,h) : layoutrule1 (x:xs) lastTok lasths lastv
>                            -- since may need multiple '}'s
>                  else
>                       x : layoutrule1 xs t (lasth:lasths) v
>               else
>                    x : layoutrule1 xs t (lasth:lasths) v
>         layoutrule1 [] _ [] _
>             = []
```

So now we can write `llexall`, a literate lexer and tokeniser of the entire source file.

```
> llexall :: String -> [PosdToken]
> llexall = layoutrule . tokenise . lexall .
>              locateatstart . unlines . tangle . lines
>     where locateatstart :: String -> PosdString
>           locateatstart s = (s,1,1)
```

End

Appendix C

`parser.lgs`: The parser

This file is `parser.lgs`. It is a parser for LetTrace source files, which are intended to be a (very restricted) subset of Gofer.

It relies on `lexer.lgs` being loaded, which defines `PosdString`, `Token` and `PosdToken` data types. Obviously the lexer in `lexer.lgs` should be used on the source file to generate the `[PosdToken]` required here. It also relies on `lambda.lgs` to define the datatype `Term`.

Written by Keith Wansbrough 1995-06-29...1995-08-08.

History

1995-06-29:	Done initial version of parser.
1995-08-08:	Modify `Binding` to include range of term. Change parser to return `TrmRnge` as well as `Term`.

To Do

None.

Known Bugs

None.

Program

The syntax of the source language is as follows:

(intention)

```
<Program>      ::=  `{' <Definitions>

<Definitions>  ::=  <Definition> `;' <Definitions>
               |    <Definition> `}'

<Definition>   ::=  identifier `=' <Term>

<Term>         ::=  identifier                    /* variable */
               |    Identifier                    /* constant */
               |    intConstant                   /* integer constant */
               |    `\' identifier `->' <Term>    /* abstraction */
               |    <Term> <Term>                 /* application */
               |    `(' <Term> `)'                /* parentheses */
```

(implementation)

```
<Program>      ::=  `{' <Definitions>

<Definitions>  ::=  <Definition> <Definitions1>

<Definitions1> ::=  `;' <Definitions>
               |    `}'

<Definition>   ::=  identifier `=' <Term>

<Term>         ::=  identifier <Term1>
               |    Identifier <Term1>
               |    intConstant <Term1>
               |    `\' identifier `->' <Term> <Term1>
               |    `(' <Term> `)' <Term1>

<Term1>        ::=  <Term> <Term1>
               |    /* nothing */
```

Note that function application associates to the left, i.e. `f g h` is parsed as `((f g) h)`, not `(f (g h))`.

As is hinted in the above grammar, an initial-lowercase `identifier` represents a variable, whereas an initial-uppercase `Identifier` represents a constant. The built-in constants are currently

`Succ`	successor function:	$\lambda x.x+1$
`Pred`	predecessor function:	$\lambda x.x-1$
`Iszero`	zero test function:	$\lambda x,y,z.(x=0)?y:z$
`Add`	addition function:	$\lambda x,y.x+y$

`Delta` is represented by a bare integer constant; `Plus` is not currently implemented.

—

```
> type Ident = String

> data Binding = Bind Ident (Term,TrmRnge)
> instance Eq Binding
>    where     (Bind id1 (t1,tr1)) == (Bind id2 (t2,tr2))
>                     = id1==id2 && t1==t2 && tr1==tr2
```

—

The following monadic parser is based closely on that given by Philip Wadler in *Monads for Functional Programming*, M. Broy, editor, *Program Design Calculi*, Proceedings of the Marktoberdorf Summer School, 30 July–8 August, 1992.

Parser type for `PosdToken` lists to whatever.

```
> type Parser a = PState -> [(a,PState)]
> type PState = [PosdToken]
```

`unit` and `‘bind‘` (basic monadic operators):

```
> unit :: a -> Parser a
> unit a x = [(a,x)]
>
> bind :: Parser a -> (a -> Parser b) -> Parser b
> (m ‘bind‘ k) x = [(b,z) | (a,y) <- m x, (b,z) <- k a y]
```

`item` (get first item):

```
> item :: Parser PosdToken
> item [] = []
> item (a:x) = [(a,x)]
```

Alternation `‘alt‘` and its identity `zero`:

```
> zero :: Parser a
> zero x = []
>
> infixr 2 ‘alt‘
> alt :: Parser a -> Parser a -> Parser a
> (m ‘alt‘ n) x = (m x) ++ (n x)
```

Filter by a predicate:

```
> flt :: Parser a -> (a -> Bool) -> Parser a
> m ‘flt‘ p = m ‘bind‘ \a -> if p a then unit a else zero
```

Literal:

```
> lit :: Token -> Parser PosdToken
> lit a = item ‘flt‘ (\(x,_,_) -> a==x)
```

Biased choice:

```
> bch :: Parser a -> Parser a -> Parser a
> (m ‘bch‘ n) x = let mx = m x in
>                   if null mx then n x else mx
```

Guarantee of success (to improve laziness):

```
> guarantee :: Parser a -> Parser a
> guarantee m x = let u = m x in (fst (head u), snd (head u)) : tail u
```

Iterate a parser to longest possible string (Wadler's second version of `'reiterate'`):

```
> reiterate :: Parser a -> Parser [a]
> reiterate m = guarantee ((        m              `bind` \a ->
>                               reiterate m      `bind` \x ->
>                               unit (a:x))
>                       `bch`     unit [])
```

—

Now we can do our parser for the above grammar. Note that the parser is almost identical to the grammar!

```
> parseProgram :: Parser [Binding]
> parseProgram =    lit (VarOp "{")     `bind` \_ ->
>                   parseDefs

> parseDefs :: Parser [Binding]
> parseDefs =       parseDef           `bind` \d ->
>                   parseDefs1         `bind` \ds ->
>                   unit (d:ds)

> parseDefs1 :: Parser [Binding]
> parseDefs1 =     (lit (VarOp ";")     `bind` \_ ->
>                   parseDefs          `bind` \ds ->
>                   unit ds)
>           `alt` (lit (VarOp "}")     `bind` \_ ->
>                   unit [])

> parseDef :: Parser Binding
> parseDef =        parseident         `bind` \(VarId id,_,_) ->
>                   lit (VarOp "=")     `bind` \_ ->
>                   parseTerm          `bind` \(t,tr) ->
>                   unit (Bind id (t,tr))

> parseTerm :: Parser (Term,TrmRnge)
> parseTerm =      (parseident         `bind` \(VarId id,v,h) ->
>                   parseTerm1 (Var (V id),TRVar (R (P v h) (P v h))))
>           `alt` (parseIdent         `bind` \(ConId id,v,h) ->
>                   parseTerm1 (case id of
>                          "Succ"    -> (Con Succ,TRCon (R (P v h) (P v h)))
>                          "Pred"    -> (Con Pred,TRCon (R (P v h) (P v h)))
>                          "Iszero"  -> (Con Iszero,TRCon (R (P v h) (P v h)))
>                          "Add"     -> (Con Add,TRCon (R (P v h) (P v h)))))
>           `alt` (parseInt           `bind` \(IntCon n,v,h) ->
>                   parseTerm1 (Con (Delta n),TRCon (R (P v h) (P v h))))
>           `alt` (lit (VarOp "\\")    `bind` \(_,v1,h1) ->
>                   parseident         `bind` \(VarId id,_,_) ->
>                   lit (VarOp "->")    `bind` \_ ->
>                   parseTerm          `bind` \(t,tr) ->
>                   let (R _ p2) = rootRange tr in
>                     parseTerm1 (Abs (V id) t,TRAbs tr (R (P v1 h1) p2)))
>           `alt` (lit (VarOp "(")     `bind` \_ ->
>                   parseTerm          `bind` \(t,tr) ->
>                   lit (VarOp ")")     `bind` \_ ->
>                   parseTerm1 (t,tr))
```

```
> parseTerm1 :: (Term,TrmRnge) -> Parser (Term,TrmRnge)
> parseTerm1 (t1,tr1) =  (parseTerm          `bind` \(t2,tr2) ->
>                         let (R p1 _) = rootRange tr1
>                             (R _ p2) = rootRange tr2 in
>                         parseTerm1 (App t1 t2,TRApp tr1 tr2 (R p1 p2)))
>                  `bch`  unit (t1,tr1)

> parseident :: Parser PosdToken
> parseident (a@(VarId _,_,_):x) = [(a,x)]
> parseident _ = []

> parseIdent :: Parser PosdToken
> parseIdent (a@(ConId _,_,_):x) = [(a,x)]
> parseIdent _ = []

> parseInt :: Parser PosdToken
> parseInt (a@(IntCon _,_,_):x) = [(a,x)]
> parseInt _ = []
```

—

And the wrapper:

```
> parse :: [PosdToken] -> [Binding]
> parse ts = let x = parseProgram ts in test x
>    where test :: [([Binding],[PosdToken])] -> [Binding]
>          test [(bs,[(EndOfFile,_,_)])] = bs
>          test [(_,(_,v,h):_)] = error ("Unexpected input at "
>              ++ show v ++ "," ++ show h)
>          test [] = error "No parse found in parse"
>          test _ = error "Ambiguous parse in parse"
```

And a function that does everything: `parsefile "filename"` returns the list of bindings from the file `filename`.

```
> parsefile :: String -> [Binding]
> parsefile = parse . llexall . openfile
```

—

`convert` takes a list of bindings and a term to be evaluated inside them, and returns a single term to be evaluated in the null environment, consisting of the bindings wrapped around the term as `Let`s.

All terms are paired with `TrmRnges`.

Note that for the ranges to make sense, the term to be evaluated should be positioned *after* the bindings.

```
> convert :: [Binding] -> (Term,TrmRnge) -> (Term,TrmRnge)
> convert ((Bind id (t1,tr1)):xs) ttr =
>           let (t2,tr2) = convert xs ttr
>               (R p1 _) = rootRange tr1
>               (R _ p2) = rootRange tr2 in
>             (Let (V id) t1 t2,TRLet tr1 tr2 (R p1 p2))
> convert [] ttr = ttr
```

—end.

End

Appendix D

genfig.lgs: The XFig-file generator

This file is `genfig.lgs`. It generates `*.fig` files intended for input to XFig version 3.1.

Written by Keith Wansbrough 1995-07-19–1995-08-26.

History

1995-07-19:	Start.
1995-07-20:	Fix output bug.
1995-07-31:	Workaround Xfig bug. Modify `termtofig`. Add `evaltofig`. Add `lines`.
1995-08-08:	Experiment with `writefig` to attempt speedup. Quick hack to enable `(Term,TrmRnge)` evals to be accepted.
1995-08-09:	Use Tsils instead of lists for extra speed.
1995-08-10:	Remove Tsils because they don't help.
1995-08-11:	Add arrows and supporting logic.
1995-08-14:	Fix `matchRanges` 'feature'. Shift arrow positioning.
1995-08-22:	Write out file one Fig at a time for speed.
1995-08-26:	Add `FigBox`, `EvCtx` drawing.

Known bugs

1995-08-11:	Variables abstracted out don't get arrows (see `termtofig0`).
1995-08-26:	`EvCtx` right-hand ends are a hack.

To do

Nothing.

Program

First we define a data type for a Fig object:

```
> data Figobj = FigText String Int Int
>             | FigLine Int Bool Bool Int Int Int Int Int
>             | FigBox Int Int Int Int Int
```

We support the following objects:

`FigText s x y` Text starting at given coordinates. We assume the following: default colour, 0 depth, 0 pen style, LaTeX Roman font, 12pt, 0 angle, 0 height, 0 width. Coordinates are in 1/1200 inch. Note that 1 em is 135 units. Correct line spacing is around 240.

`FigLine th fa ba asz x1 y1 x2 y2` A line from `x1`,`y1` to `x2`,`y2`, of thickness `th`, with an optional forward arrow (`fa`) and backward arrow (`ba`), `fa`,`ba` booleans. `asz` is the length of the arrowhead.

`FigBox th x1 x2 y1 y2` A box from `x1`,`y1` to `x1`,`y2`, of thickness `th`.

Now we output a list of objects to a file. `writefigfile` takes a filename and a list of `Figobjs`, and writes the list to the given file.

For speed, we use `Gnirts`, which are snoc-lists of char, used to speed appends. Actually, we *did* use these; now we don't; hence the prefix `*` rather than `>`, indicating the following is commented out.

```
* type Gnirts = Tsil Char

* writefigfile :: String -> [Figobj] -> Dialogue
* writefigfile name figs = writeFile name s
*                     (\ioerror resps -> [AppendChan stdout (show' ioerror)])
*                     done
*   where s =
*               "#FIG 3.1\nLandscape\nCenter\nInches\n1200 2\n\n"
*               ++ "#\n# Fig file (" ++ name
*               ++ ") generated by writefigfile\n#\n\n"
*               ++ writefigs figs
*               ++ "\n#\n# end of file\n#\n"

*   where s = tsil2list
*              (list2tsil "#FIG 3.1\nLandscape\nCenter\nInches\n1200 2\n\n"
*               +- "#\n# Fig file (" +- name
*               +- ") generated by writefigfile\n#\n\n"
*               +-+ writefigs figs
*               +- "\n#\n# end of file\n#\n")

* writefigs :: [Figobj] -> Gnirts
```

```
* writefigs (f:fs) = writefig f +-+ "\n" +-+ writefigs fs
* writefigs []     = Nil

* writefig :: Figobj -> Gnirts

* writefig (FigText s x y) = list2tsil "4 0 -1 0 0 1 12 0 0 0 0 "
*                            +- show x +- " " +- show y +- " " +- s +- "\\001"

* writefig (FigLine th fa ba asz x1 y1 x2 y2) =
*   list2tsil "2 1 0 " +- show th +- " 0 0 0 0 -1 0 0 0 0 "
*   +- (if fa then "1" else "0") +- " "
*   +- (if ba then "1" else "0") +- " "
*   +-+ (if fa then list2tsil "2 1 " +- show th +- " "
*                  +- show asz +- " " +- show asz +- " " else Nil)
*   +-+ (if ba then list2tsil "2 1 " +- show th +- " "
*                  +- show asz +- " " +- show asz +- " " else Nil)
*   +- "2 " +- show x1 +- show y1 +- show x2 +- show y2
```

Now the actual code that is used.

```
> writefigfile name figs =
>        writeFile name s1
>          (\ioerror resps -> [AppendChan stdout (show' ioerror)])
>          (writefigfile0 name figs
>             (\ioerror resps -> [AppendChan stdout (show' ioerror)])
>             (appendFile name s2
>                (\ioerror resps -> [AppendChan stdout (show' ioerror)])
>                done))
>  where s1 =
>               "#FIG 3.1\nLandscape\nCenter\nInches\n1200 2\n\n"
>               ++ "#\n# Fig file (" ++ name
>               ++ ") generated by writefigfile\n#\n\n"
>        s2 =
>               "\n#\n# end of file\n#\n"
>        writefigfile0 :: String -> [Figobj]
>                          -> FailCont -> SuccCont -> Dialogue
>        writefigfile0 name [] fail succ =
>               succ
>        writefigfile0 name (f:fs) fail succ =
>               appendFile name s
>                 fail
>                 (writefigfile0 name fs fail succ)
>          where s = "\n" ++ writefig f

> writefigs :: [Figobj] -> String
> writefigs (f:fs) = writefig f ++ "\n" ++ writefigs fs
> writefigs []     = ""

> writefig :: Figobj -> String

> writefig (FigText s x y) = "4 0 -1 0 0 1 12 0 0 0 0 "
>                            ++ show x ++ " " ++ show y ++ " " ++ s ++ "\\001"

> writefig (FigLine th fa ba asz x1 y1 x2 y2) =
>   "2 1 0 " ++ show th ++ " -1 7 0 0 -1 0 0 0 -1 "
>   ++ (if fa then "1" else "0") ++ " "
>   ++ (if ba then "1" else "0") ++ " "
>   ++ "2 "
>   ++ (if fa then "\n\t0 0 " ++ show th ++ " "
>                  ++ show (asz/2) ++ " " ++ show asz ++ " " else "")
```

```
>   ++ (if ba then "0 0 " ++ show th ++ " "
>                  ++ show (asz/2) ++ " " ++ show asz ++ " " else "")
>   ++ "\n\t"
>   ++ show x1 ++ " " ++ show y1 ++ " " ++ show x2 ++ " " ++ show y2

> writefig (FigBox th x1 y1 x2 y2) =
>   "2 1 0 " ++ show th ++ " -1 7 0 0 -1 0 0 0 -1 "
>   ++ "0 0 "
>   ++ "5 "
>   ++ "\n\t"
>   ++ show x1 ++ " " ++ show y1 ++ " " ++ show x2 ++ " " ++ show y1 ++ " "
>   ++ show x2 ++ " " ++ show y2 ++ " " ++ show x1 ++ " " ++ show y2 ++ " "
>   ++ show x1 ++ " " ++ show y1
```

Now for some stuff more specific to what we are doing.

```
> textlen :: String -> Int
> textlen s = 135 * (length s)
```

First, a data structure parallel to `Term` for storing the (x,y) position of each term on the page.

```
> data TrmPos = TPVar (Int,Int) | TPCon (Int,Int) | TPAbs TrmPos (Int,Int)
>                 | TPApp TrmPos TrmPos (Int,Int)
>                 | TPLet TrmPos TrmPos (Int,Int)
>                 | TPNil

> instance Eq TrmPos
>    where (TPVar r      == TPVar r')       = (r == r')
>          (TPCon r      == TPCon r')       = (r == r')
>          (TPAbs m r    == TPAbs m' r')    = (m == m' && r == r')
>          (TPApp m n r  == TPApp m' n' r') = (m == m' && n == n' && r == r')
>          (TPLet m n r  == TPLet m' n' r') = (m == m' && n == n' && r == r')
>          (TPNil        == TPNil)      = True
>          (_         == _)           = False
```

`termtofig0` takes a term, a start x coordinate and a y coordinate, and returns a list of fig objects paired with a `TrmPos` tree, paired with the end x coordinate.

`termtofig` takes a term and a start x and y coordinate, and returns a list of fig objects paired with a `TrmPos` tree.

```
> termtofig :: (Term,TrmRnge) -> Int -> Int -> ([Figobj],TrmPos)
> termtofig (t,_) x y = fst (termtofig0 t x y)

> termtofig0 :: Term -> Int -> Int -> (([Figobj],TrmPos),Int)
> termtofig0 t@(Var _) x y = (([(FigText s x y)],TPVar (x,y)), x + (textlen s))
>                               where s = show t
> termtofig0 t@(Con _) x y = (([(FigText s x y)],TPCon (x,y)), x + (textlen s))
>                               where s = show t
> termtofig0 (Abs v m) x y =    (([(FigText "\\\\ " x y)]
>                                 -- the space is a workaround for an
>                                 -- Xfig bug.
>                           ++ fs1
>                           ++ [(FigText " -> " x2 y)]
>                           ++ fs2,TPAbs tpm (x,y)), x4)
>                 where x1 = x + textlen "\\"
>                       ((fs1,_),x2) = termtofig0 (Var v) x1 y
```

```
>                        x3 = x2 + textlen " -> "
>                        ((fs2,tpm),x4) = termtofig0 m x3 y
> termtofig0 (App m n) x y = (([(FigText "(" x y)]
>                         ++ fs1
>                         ++ fs2
>                         ++ [(FigText ")" x4 y)],TPApp tpm tpn (x,y)), x5)
>                  where x1 = x + textlen "("
>                        ((fs1,tpm),x2) = termtofig0 m x1 y
>                        x3 = x2 + textlen " "
>                        ((fs2,tpn),x4) = termtofig0 n x3 y
>                        x5 = x4 + textlen ")"
> termtofig0 (Let v m n) x y =    (([(FigText "let " x y)]
>                              ++ fs1
>                              ++ [(FigText " be " x2 y)]
>                              ++ fs2
>                              ++ [(FigText " in " x4 y)]
>                              ++ fs3,TPLet tpm tpn (x,y)),x6)
>                  where x1 = x + textlen "let "
>                        ((fs1,_),x2) = termtofig0 (Var v) x1 y
>                        x3 = x2 + textlen " be "
>                        ((fs2,tpm),x4) = termtofig0 m x3 y
>                        x5 = x4 + textlen " in "
>                        ((fs3,tpn),x6) = termtofig0 n x5 y
```

A function to take a `TrmRnge` and a `TrmPos` and return a list of (`Range`,`Position`) pairs:

```
> matchRangePos :: TrmRnge -> TrmPos -> [(Range,(Int,Int))]
> matchRangePos (TRVar     r) (TPVar       p) = [(r,p)]
> matchRangePos (TRCon     r) (TPCon       p) = [(r,p)]
> matchRangePos (TRAbs m   r) (TPAbs m'    p) = (r,p) : matchRangePos m m'
> matchRangePos (TRApp m n r) (TPApp m' n' p) = ((r,p) : matchRangePos m m')
>                                               ++ matchRangePos n n'
> matchRangePos (TRLet m n r) (TPLet m' n' p) = ((r,p) : matchRangePos m m')
>                                               ++ matchRangePos n n'
```

Support functions and datatype:

```
> data RnPs = RP (Range,(Int,Int))
> instance Eq RnPs
>    where ((RP (r1,p1)) == (RP (r2,p2))) = (r1==r2)
> instance Ord RnPs
>    where ((RP (r1,p1)) <= (RP (r2,p2))) = (r1<=r2)

> toRnPs :: [(Range,(Int,Int))] -> [RnPs]
> toRnPs [] = []
> toRnPs (x:xs) = (RP x) : (toRnPs xs)
> fromRnPs :: [RnPs] -> [(Range,(Int,Int))]
> fromRnPs [] = []
> fromRnPs ((RP x):xs) = x : (fromRnPs xs)
```

A function to show the transformation between two terms by drawing arrows between corresponding elements, given in addition to two term position lists, two deltas: the vertical distance to shift the tail and head of the arrow, respectively, to avoid writing over the text.

```
> showTransform :: [(Range,(Int,Int))] -> [(Range,(Int,Int))]
>                  -> Int -> Int -> [Figobj]
> showTransform tps0 tps1 d0 d1 = drawArrows (matchRanges (sortRP tps0)
>                                               (sortRP tps1)) d0 d1
>    where drawArrows :: [((Int,Int),(Int,Int))] -> Int -> Int -> [Figobj]
>          drawArrows [] _ _ = []
>          drawArrows (((x0,y0),(x1,y1)):xyxys) d0 d1
>                       = (FigLine 1 True False 100 x0 (y0+d0) x1 (y1+d1))
>                         : (drawArrows xyxys d0 d1)
>          matchRanges :: [(Range,(Int,Int))] -> [(Range,(Int,Int))]
>                        -> [((Int,Int),(Int,Int))]
>          matchRanges [] _ = []
>          matchRanges _ [] = []
>          matchRanges rps0@((r0,p0):rpt0) rps1@((r1,p1):rpt1)
>                          | r0 < r1 = matchRanges rpt0 rps1
>                          | r0 > r1 = matchRanges rps0 rpt1
>                          | otherwise = (p0,p1) :
>                                 if drop0 r0 rpt0 then
>                                      matchRanges rpt0 rps1
>                                 else
>                                      matchRanges rps0 rpt1
>                  where drop0 a []        = False
>                        drop0 a ((b,_):_) = a==b
>          sortRP :: [(Range,(Int,Int))] -> [(Range,(Int,Int))]
>          sortRP = fromRnPs . sort . toRnPs
```

A function to find the position of the hole in a term.

```
> findHolePos :: EvCtx -> TrmPos -> (Int, Int, Int, Int)
> findHolePos c tp = head (findHolePos0 c tp)

> findHolePos0 :: EvCtx -> TrmPos -> [(Int, Int, Int, Int)]
> findHolePos0 (ECHole) (TPVar     (x,y)) = [(x,y,x+100,y)]
> findHolePos0 (ECHole) (TPCon     (x,y)) = [(x,y,x+100,y)]
> findHolePos0 (ECHole) (TPAbs _   (x,y)) = [(x,y,x+100,y)]
> findHolePos0 (ECHole) (TPApp _ _ (x,y)) = [(x,y,x+100,y)]
> findHolePos0 (ECHole) (TPLet _ _ (x,y)) = [(x,y,x+100,y)]
> findHolePos0 _          (TPVar       _) = []
> findHolePos0 _          (TPCon       _) = []
> findHolePos0 (ECAbs m  ) (TPAbs m'    _) = findHolePos0 m m'
> findHolePos0 (ECApp m n) (TPApp m' n' _) = findHolePos0 m m'
>                                          ++ findHolePos0 n n'
> findHolePos0 (ECLet m n) (TPLet m' n' _) = findHolePos0 m m'
>                                          ++ findHolePos0 n n'
```

A function to show the mapping between `EvCtx`s in two successive terms, with `d0`,`t0`,`d1`,`t1` the distances from text and thickness of box for the first and second terms.

```
> showEvCtx :: (Int,Int,Int,Int) -> (Int,Int,Int,Int)
>              -> Int -> Int -> Int -> Int -> [Figobj]
> showEvCtx (x1a,y1a,x2a,y2a) (x1b,y1b,x2b,y2b) d0 t0 d1 t1
>    = [(FigBox 1 x1a (y1a+d0) x2a (y2a+d0+t0)),
>       (FigBox 1 x1b (y1b+d1) x2b (y2b+d1+t1))]
```

Now a function to take an evaluation sequence and return a list of fig objects:

```
> evaltofig :: [(Rule,(Term,TrmRnge),EvCtx,(Term,TrmRnge))] -> [Figobj]
> evaltofig qs = evaltofig0 qs TPNil [] 600 600
```

```
> evaltofig0 :: [(Rule,(Term,TrmRnge),EvCtx,(Term,TrmRnge))]
>            -> TrmPos -> [(Range,(Int,Int))] -> Int -> Int -> [Figobj]
> evaltofig0 [] _ _ _ _ = []
> evaltofig0 ((r,m,c,n@(_,nr)):qs) tps0 rps0 x y = [(FigText (show r) x y)]
>                  ++ tfs
>                  ++ (if tps0==TPNil then []
>                       else (showEvCtx holeps0 holeps 70 60 (-230) 60))
>                  ++ showTransform rps0 rps (100) (-200)
>                  ++ evaltofig0 qs tps rps x (y + 960)
>            where (tfs,tps) = termtofig n (x + 300) (y + 480)
>                  rps = matchRangePos nr tps
>                  holeps0 = findHolePos c tps0
>                  holeps  = findHolePos c tps
```

End