Linearity and Laziness

David Wakeling and Colin Runciman
University of York*

Abstract

A criticism often levelled at functional languages is that they do not cope elegantly or efficiently with problems involving changes of state. In a recent paper [26], Wadler has proposed a new approach to these problems. His proposal involves the use of a type system based on the linear logic of Girard [7]. This allows the programmer to specify the "natural" imperative operations without at the same time sacrificing the crucial property of referential transparency.

In this paper we investigate the practicality of Wadler's approach, describing the design and implementation of a variant of Lazy ML [2]. A small example program shows how imperative operations can be used in a referentially transparent way, and at the same time it highlights some of the problems with the approach. Our implementation is based on a variant of the G-machine [15, 1]. We give some benchmark figures to compare the performance of our machine with the original one. The results are disappointing: the cost of maintaining linearity in terms of lost optimisations at compile-time, and the extra data structures that must be created at run-time more than cancels out the gains made by using linear types to reduce the amount of garbage collection. We also consider how the language and the implementation can be extended to accommodate aggregates such as arrays. Here the results are more promising: linear arrays are usually more efficient than trailered ones, but they are less efficient than destructively-updated ones. We conclude that larger aggregates are the most promising area of application for Wadler's type system.

1 Introduction

For many years researchers have advocated the use of functional programming languages because of their mathematical tractability, their economy of expression and their suitability for programming parallel computers. But functional languages do not cope "naturally" with real world situations involving changes of state, such as

*Authors' address: Department of Computer Science, University of York, Heslington, York Y01 5DD, United Kingdom. Electronic mail: dw@uk.ac.york.minster, colin@uk.ac.york.minster

altering a pixel on a bit-mapped display or updating a record in a database. Where the imperative solution to these problems is concise and efficient, the functional one is verbose and inefficient. Any function altering a bit-mapped display, for example, must take the bit-map as one of its arguments and return a new bit-map as part of its result. The verbosity of this solution is annoying, but even worse is its manifest inefficiency: the implementation cannot update the display directly without sacrificing referential transparency, and so it must copy the bit-map after each pixel has been altered.

In situations like these an optimising compiler is a double-edged sword. It may improve performance dramatically, for example by detecting that the bit-map *can* be updated directly without the loss of referential transparency, but in doing so it also turns a program which is inefficient into one which is inefficient in *unpredictable* ways : a small change to the program might (by fooling the compiler's analysis) lead to a large decrease in its efficiency which is hard to trace. It is most unfortunate that the behaviour of a functional program should depend so heavily on the cleverness of the compiler. The functional programmer should not be expected to know that it is better to write a program in one way rather than another just so that it can be compiled more efficiently.

In a recent paper [26], Wadler has proposed a new approach to problems involving changes of state. His approach does not try to reduce the verbosity of the functional solution to these problems, but it does try to increase its efficiency and predictability.

All implementations of lazy functional languages employ some notion of *sharing*, whether it is achieved indirectly by using an environment or directly by using pointers. One view is that sharing is essential to an efficient implementation, saving time by avoiding the recomputation of values and saving space by having only one copy of each value. But another view is that sharing is a source of inefficiency because the possibility of sharing prevents the implementation from re-using storage space immediately. Instead, storage space which is no longer in use must eventually be recovered for re-use by an expensive process known as *garbage collection*. Wadler has developed a type system, based on the linear logic of Girard [7], that attempts to reconcile these two viewpoints by giving the programmer greater control over storage management. In Wadler's type system there are two distinct families of types, *conventional types* and *linear types*. A value of a conventional type may be shared, as in `share x = (x,x)` or it may be thrown away, as in `throw x = ()`. A value of a linear type, on the other hand, must obey the *linearity constraint*: it cannot be shared and it cannot be thrown away.

At the implementation level, there may be many pointers to a conventional value (it may be *duplicated*), or there may be none at all (it may be *discarded*); there is always exactly one pointer to a linear value. Conventional storage can only be safely recovered for re-use by garbage collection, but linear storage can be recovered directly as a result explicit instructions in the compiled code for the program.

Wadler's idea is that the programmer should specify whether a new type is conventional or linear when it is declared — a trade-off between flexibility and efficiency. There are no restrictions on the use of values of a conventional type, but they can-

not be updated directly and they require garbage collection. Conversely, values of a linear type must be used exactly once, but they can be updated directly and they avoid the overhead of garbage collection.

The rest of this paper is organised into six sections. Section 2 briefly reviews Wadler's type system, and Section 3 describes a functional programming language that makes use of it. Section 4 gives a small example program. Section 5 is concerned with various aspects of the implementation of this language, and Section 6 describes how the language and the implementation can be extended to incorporate aggregate structures such as arrays. Finally, Section 7 reviews some closely related work, and Section 8 concludes.

2 Wadler's Type System

In Wadler's type system two distinct families of types coexist. A *conventional type* can be either a base type, a function type or a pair type:

$$T,\ U,\ V ::= K \mid (U \to V) \mid (U \times V)$$

where K ranges over conventional base types and T, U and V range over conventional types. A *linear type* can also be either a base type, a function type or a pair type:

$$P,\ Q,\ R ::= J \mid (Q\ \ R) \mid (Q \otimes R)$$

and in this case, J ranges over linear base types and P, Q and R range over linear types. Wadler's *nonlinear type system* combines these two families of types:

$$T,\ U,\ V ::=\ !K \mid (U \to V) \mid (U \times V) \mid K \mid (U\ \ V) \mid (U \otimes V)$$

Here $!K$ ranges over conventional base types, K ranges over linear base types and T, U and V range over types.

The *nonlinear λ-calculus* is a variant of the λ-calculus that combines the terms of the *conventional λ-calculus* and the *linear λ-calculus* (in which all bound variables must be used exactly once) in an analogous way*:

$$\begin{array}{rl} t,\ u,\ v ::= & x \\ \mid & (!\lambda x : U\ .v) \\ \mid & (\lambda x\ \ : U\ .v) \\ \mid & (!t\ u) \\ \mid & (t\ u) \\ \mid & (!C\ t_1 \ldots t_n) \\ \mid & (C\ t_1 \ldots t_n) \\ \mid & (\text{case } u\ of\ !C_1\ x_{11} \ldots x_{1n} \to v_1 \mid \ldots\ !C_m\ x_{m1} \ldots x_{mn} \to v_m) \\ \mid & (\text{case } u\ of\ C_1\ x_{11} \ldots x_{1n} \to v_1 \mid \ldots\ C_m\ x_{m1} \ldots x_{mn} \to v_m) \\ \mid & (\text{fix } t) \end{array}$$

*In his paper Wadler adopts an *inverse* "!" *convention*: terms from the *linear* λ-calculus are annotated with the symbol "¡". Our notation follows the tradition of linear logic.

Here x ranges over variables and t, u and v range over terms. The novel feature of this calculus is that it allows algebraic type declarations of the form

$$K = C_1\ T_{11}\ \ldots\ T_{1p} \mid \cdots \mid C_n\ T_{n1}\ \ldots\ T_{nq}$$

where K is a new base type name, the C_i are new constructor names, and the T_{ij} are types.

Figure 1 gives the typing rules for the nonlinear λ-calculus in the usual style. Wadler discusses these rules in detail, but for the purposes of this paper the important points to note are:

- Each assumption in A about a linear variable must be used exactly once in the typing (rule VAR). An assumption list is nonlinear if each assumption $x : T$ in it has nonlinear T. In other words, the type-checker enforces the linearity constraint.
- The closure of a conventional function may not incorporate a linear value (rule $\rightarrow \mathcal{I}$). This is because there are no restrictions on the use of conventional functions. If the closure could bind a linear value there would be no restriction on the use of this value either, and so this binding must be disallowed.
- The two rules for applications (rule $\rightarrow \mathcal{E}$ and rule $\mathcal{E}$) make it clear that no linear variable may appear in both the function and argument portion of an application. Clearly, if each linear variable in t occurs exactly once in A and each linear variable in u occurs exactly once in B then each linear variable in the conjunction of the two lists, $A.B$ occurs exactly once in $(t\ u)$.
- A conventional data structure may not have any linear components (rule $!K\mathcal{I}$). This is because updating any component of a data structure updates the structure itself, and the updating of conventional data structures must be disallowed.

3 A Functional Language

Although the nonlinear λ-calculus is both simple and elegant, the syntax is so Spartan that only a fanatic would advocate using it to program a computer. We have developed a functional programming language with a more agreeable syntax which is based on the nonlinear λ-calculus and uses the nonlinear type system. This language is called Nonlinear Lazy ML (NLML), and it is a variant the language Lazy ML (LML) developed at Chalmers University by Augustsson and Johnsson [2]. In this section we shall be concerned mainly with the type system and type inference, an area where there are significant and interesting differences between the two languages.

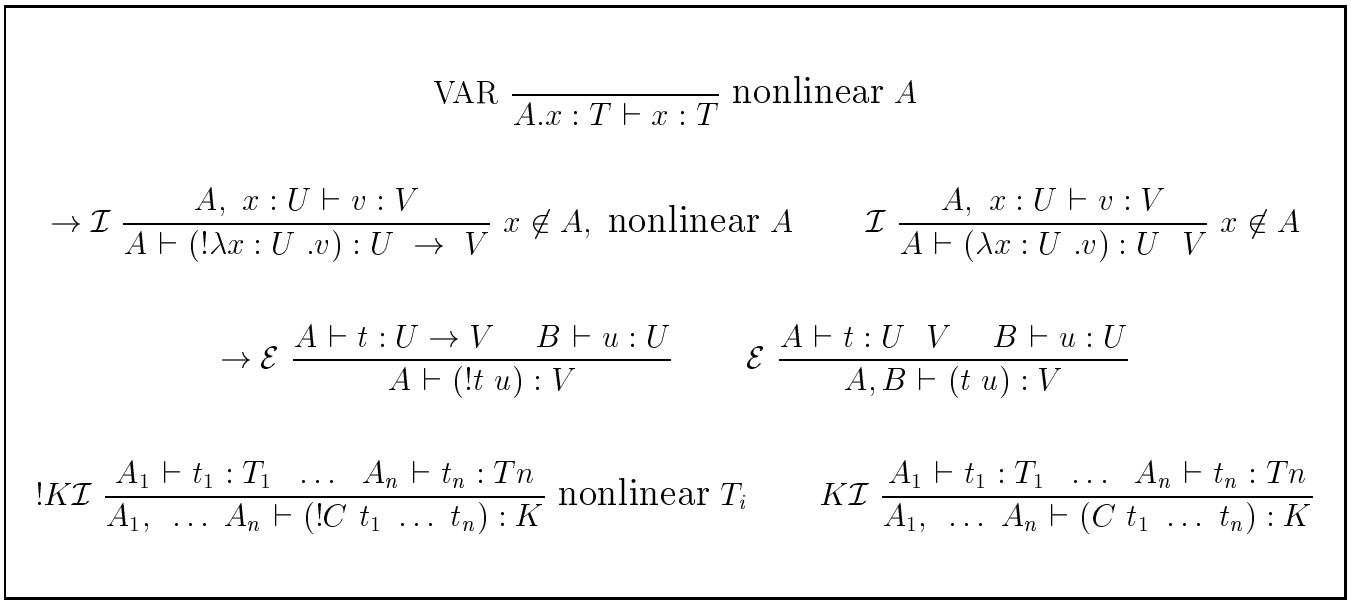

Figure 1: Typing rules for the nonlinear λ-calculus

3.1 Types

In NLML the conventional base types are `Int`, `Bool` and `Char`, and there are no linear base types. One of the first decisions taken in the design of NLML was that providing linear versions of the basic types was not worthwhile. This decision was partly the result of reading Lafont's work [16, 17], and partly the result of our unsuccessful attempts to write useful functions using linear integers.

Conventional type variables are written as `!*a` and linear type variables are written as `*a`. Conventional functions are constructed with the `->` arrow and linear functions are constructed with the `-o` arrow. The requirement that the closure of a conventional function must not incorporate a linear value means that the function

```
signature f: *a->(!*b->*a);
f x y = x
```

is type-incorrect because it allows a linear value to be used many times (every time the conventional function `(f v)` is applied to a conventional argument, the implementation duplicates the linear value `v`). There would be no problem, and the type-checker could be more permissive, if it could guarantee that the function `(f v)` would be applied only once. This, of course, is what the `-o` arrow is used for, and the following definition is type-correct.

```
signature g: *a->(!*b-o*a);
g x y = x
```

On first acquaintance linear functions appear to be exotic beasts. Their role in

NLML, however, is a very minor one: they serve only to placate the type-checker by restricting the use of partial applications, ensuring the integrity of linear values. In our experience, the programming style that results from a more ambitious use of linear functions is not to be recommended. Lafont [17], for example, makes extensive use of them and his programs are rather difficult to understand.

NLML allows the programmer to declare algebraic data types. There are two different kinds of type declaration, conventional ones and linear ones. For example,

```
type Clist !*a = Cnil + Ccons !*a (Clist !*a)
```

declares the type of a conventional list of conventional values. There are no restrictions on lists of this type; they may be used any number of times. The declaration

```
linear type Llist !*a = Lnil + Lcons !*a (Llist !*a)
```

declares the type of a linear list of conventional values. A list of this type must be used exactly once; it cannot be shared or thrown away. The use of the list *items*, though, is unrestricted. The linear type announces that the programmer is prepared to trade flexibility in exchange for a more efficient implementation. Flexibility is lost because the type system insists that linear lists should be used exactly once, but efficiency is gained because the implementation can re-use the space occupied by linear list cells explicitly, so avoiding the overhead of garbage collection.

In NLML, the explicit recovery of storage is accomplished by *destructive case-analysis*. Consider the following definition of a concatenation function for linear lists:

```
signature nconc: (Llist !*a)->(Llist !*a)-o(Llist !*a);
   nconc Lnil ys = ys
|| nconc (Lcons x xs) ys = Lcons x (nconc xs ys)
```

Here pattern-matching is being used to perform *case-analysis* of the first argument. When one of the two clauses has been chosen, the space occupied by the linear list cell that was examined can safely be recovered — the type system guarantees that it is not referred to elsewhere. Thus, the `nconc` function *destroys* its first argument in computing its result.

3.2 Type-checking

In NLML type-checking takes place after the program has been translated into the nonlinear λ-calculus using techniques similar to those described in Peyton Jones' book [21]. During this translation, information from any type signatures in the program, along with information about the types of the primitive operations, is used to annotate the resulting terms. So, for example, the function

```
signature id: !*a->!*a;
id x = x
```

is translated into

$$id = ((\lambda x.x :!X) :!X \rightarrow !X)$$

Type-checking is then performed in two stages. The first stage uses a simple variant of Milner's archetypal type-checking algorithm $\mathcal{W}$ [20] using information supplied in the type signatures. At this stage, a function is considered to be type-incorrect if it is *ambiguous*. For example, if its type signature is omitted, then `id` is ambiguous because it can have four possible types ($X \rightarrow X$, $!X \rightarrow !X$, $X \quad X$ and $!X \quad !X$). The programmer must often write type signatures to resolve ambiguities for the type-checker.

In the second stage, the type-checker ensures that all variables declared to have a linear type obey the linearity constraint. This is a simple syntactic check, performed by following each possible path through a function and counting the occurrences of the variables with linear types. Some additional checks ensure that linear values are never incorporated in cyclic structures created with `let rec`, that they cannot appear in the closures of conventional functions, and that they cannot appear as the components of conventional data structures.

This two stage implementation is largely a matter of convenience. A direct implementation of the typing rules given in Section 2 involves reference counting the assumption lists maintained by the type-checker, and this turns out to be rather awkward.

From the above description it might seem that type-checking is quite straightforward. However, there are many pitfalls for the unwary. Consider, for example, the following function which increments the nth element of a linear list.

```
signature inc: Int->(Llist Int)->(Llist Int);
inc n Lnil = Lnil
inc 1 (Lcons x xs) = Lcons (x+1) xs
inc n (Lcons x xs) = Lcons x (inc (n-1) xs)
```

Using the pattern-matching transformation described by Wadler in Peyton Jones' book [21], this function is translated into the nonlinear λ-calculus as follows (type annotations have been omitted for the sake of clarity):

```
inc = λA1.λA2.
    case A2 in
        Lnil : Lnil
    ||  Lcons x xs :
            case A1 in
                1 : case A2 in
                        Lnil : ERROR
                    ||  Lcons x xs : Lcons (x + 1) xs
            ||  n : case A2 in
                        Lnil : ERROR
                    ||  Lcons x xs : Lcons (inc (n - 1)) xs
```

The problem here is the repeated case-analysis of *A2*. It must be a linear list, yet in the translation it may be used twice, and this is a type-incorrect. As Wadler points out, it is straightforward to improve the translation of pattern-matching to avoid the repeated case-analysis of a single variable. However, there are other more subtle problems that cannot be solved in this way, as we shall now show.

In NLML, as in LML, argument patterns are always matched strictly. (This differs from languages like Miranda[†] where tuple patterns are matched *lazily*: the argument is not evaluated unless one of its components is required during evaluation of the right-hand side.) But some form of lazy pattern-matching is essential, and in both NLML and LML this is achieved using the binding mechanism of the `let`-expression. Consider an expression of the form

```
let p = q in r
```

When this expression is evaluated, no check is made that q matches the pattern p until the value of one of the variables in p is required in r. Now suppose we declare two linear types as follows.

```
linear type Lpair *a *b = Lpr *a *b

linear type Signal = S Bool Signal
```

The first is the type of pairs of linear values and the second is the type of infinite sequences of booleans. There is nothing sinister about these two declarations, but now we use them to define the function `divide` which copies a `Signal`:

```
signature divide: Signal->(Lpair Signal Signal);
divide (S x xs) =
    let (Lpr xs1 xs2) = divide xs in Lpr (S x xs1) (S x xs2)
```

[†]Miranda is a trademark of Research Software Limited

This function is translated into the nonlinear λ-calculus so that the pattern-matching in the `let`-expression is lazy (once again, type annotations have been omitted for the sake of clarity):

```
divide = λA1.
    case A1 in
        S x xs :
            let rec t = divide xs
            and xs1 = case t in
                          Lpr u v : u
                        || _ : ERROR
            and xs2 = case t in
                          Lpr u v : v
                        || _ : ERROR
            in Lpr (S x xs1) (S x xs2)
      || _ : ERROR
```

This will not type-check either. Here the local variable t introduced during the pattern-matching transformation must be a linear pair, and yet it is used by two case-expressions, each of which discards one of its linear components.

Now, it is an entertaining, if somewhat futile, exercise to attempt to translate the `divide` function into the nonlinear λ-calculus — where, remember, there is no pattern-matching of any kind — while preserving both the linearity and the laziness suggested by the original definition. It cannot be done: linearity demands that the result of *(divide xs)* must not be shared at all; laziness demands that it must be shared among the selectors for *xs1* and *xs2*. The problem evinced by `divide` is in fact very serious: at one time, this simple function seemed to throw our whole enterprise into jeopardy. We do have a solution of sorts, but since it depends on details of the implementation we shall postpone discussion of it until Section 5.

Worrying as they are, these problems with type-checking have not prevented us from writing several interesting NLML programs. One of these is described below.

4 An Example: Generating the Mandelbrot Set

The Mandelbrot set [19] is a set of complex numbers governed by the iterative formula $z \leftarrow z^2 + k$. If this formula converges for an initial z of (0,0) then the point k is within the set, otherwise it is not. Unfortunately, it is impossible to find all and only those points for which the iteration converges. However, it is possible to find an *approximation* to the set by making use of a simple and sufficient condition for *divergence*: the sequence of iterations will diverge if the size of the complex number z, written $|z|$, exceeds 2. Any point for which the iteration has not diverged after a fixed finite number of iterations is assumed to lie within the set. When enough points have been computed they can be plotted in the complex plane (Figure 2).

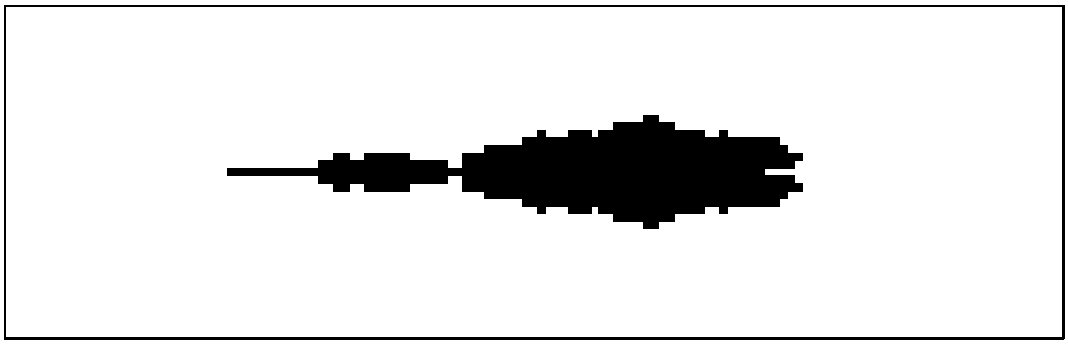

Figure 2: An Approximation of the Mandelbrot set

At the heart of our program to generate the set is an implementation of the iteration $z \leftarrow z^2 + k$. Trading flexibility for efficiency, we shall make the type of complex numbers a linear one

```
linear type Complex = C Float Float
```

allowing a true assignment to be used in the iterative loop, just as it would be in an imperative programming language. We can now define the central function `inset` which returns `true` if the point `k` is still with in the set after a certain predefined number of iterations, and `false` otherwise.

```
signature inset: Complex->Bool;
inset k = inset1 0 (C 0.0 0.0) k
```

The `inset` function uses an auxiliary `inset1` to compute its result. The three arguments that it gives to this function are an initial value for the iteration counter, an initial value for `z` and the point `k`. The iterations stop when either the iteration count reaches the predefined maximum, `NITER`, or |`z`| exceeds 2. The first clause of the `inset1` function tests the iteration counter:

```
inset1 NITER z k = yes z k
```

If the number of iterations has reached `NITER`, then the point `k` is assumed to be within the set and the result is `true`. Unfortunately, it would not be type-correct simply to return `true` immediately — that would involve discarding two linear values, `z` and `k`. Instead, an intermediate function `yes` is required:

```
signature yes: Complex->Complex-oBool;
yes (C r1 i1) (C r2 i2) = true
```

This function returns `true` only after using up both of its arguments during destructive case-analysis; the function `no` is similar. The second clause of the `inset1` function tests for $|z| > 2$ and it is rather more complicated:

```
inset1 n z k =
    let (Lpr z1 z2) = copy_complex z in
    if squared_size z1 >= 4.0 then
        no z2 k
    else
        let (Lpr k1 k2) = copy_complex k in
        inset1 (n+1) (add_complex (sqr_complex z2) k1) k2
```

The size of a complex number is found by squaring each of its parts, adding them together and taking the square root of the sum. However, since we only want the size of `z` in order to compare it with 2, we avoid the square root operation by comparing with 4 instead. The function `squared_size` returns the square of the size of a complex number:

```
signature squared_size: Complex->Float;
squared_size (C r i) = (r .* r) .+ (i .* i)
```

Complex numbers are represented by a linear type and so every operation on a complex number, including `squared_size`, consumes it by destructive case-analysis. Thus, we must take a copy of `z` before testing it so that we can use it again afterwards. Assuming that `z` fails the test, we can carry out another iteration, which involves computing the value of `z*z + k`. To do this we must explicitly copy `k` in order to satisfy the type-checker which insists that all linear values must be used exactly once. We can avoid copying `z` by specialising the function `mul_complex` of two arguments to a function `sqr_complex` of one:

```
signature sqr_complex: Complex->Complex;
sqr_complex (C r i) =
    C ((r .* r) .- (i .* i)) ((r .* i) .+ (r .* i))
```

The Mandelbrot set itself may now be obtained by mapping the `inset` function over a grid of complex numbers represented by a list.

This program illustrates many of the problems that we have encountered while writing programs in NLML. The loss of flexibility that results from using linear types is dramatic. Extra functions must often be written to copy or throw away linear values, and in order to avoid writing these artificial functions, one often resorts to programming in unnatural and devious ways.

Having to supply type signatures can be irksome, especially for quite simple functions. More seriously, type signatures can lead to a creeping loss of polymorphism. A signature which is appropriate for a function in the context of the program being developed can mask its true polymorphic nature, something which a system performing pure type inference would reveal.

Our implementation has no built-in linear types or library functions to process them. In this particular example, the programmer has to define linear pairs explicitly.

In time, of course, it is likely that some linear types would become built-in to the implementation, and some functions for processing them would find their way into libraries. But many of these new library functions would just be imitations of existing ones which process conventional types. This need to provide "two of everything" complicates life for both the programmer and the implementor.

Overall, programming in NLML is quite laborious. The programs are more cumbersome and there is a significant loss of flexibility when compared with a conventional language like LML. The restrictive nature of the type-system means that the prods that one receives from the type-checker during program development are both frequent and sharp, and many of the problems that it finds can be hard to correct.

5 Implementation

This section describes the implementation of NLML using an abstract machine called the *nonlinear G-machine*. The nonlinear G-machine is closely related to Johnsson and Augustsson's G-machine [15, 1], and in what follows we shall refer to their machine as the *conventional G-machine*. Some familiarity with the idea of programmed graph reduction and the conventional G-machine is assumed in this section; for those without such familiarity an excellent tutorial description can be found Peyton Jones' book [21].

5.1 The Nonlinear G-machine

After type-checking and lambda-lifting [15], the NLML compiler compiles every function into code for the nonlinear G-machine. This abstract machine has instructions to construct and manipulate graphs representing expressions. It can be thought of as a finite-state machine with the following components:

I, the instructions remaining to be executed;

S, a stack of pointers;

V, a stack of basic values;

C, the conventional partition of the graph;

L, the linear partition of the graph;

E, a global environment;

D, a dump stack.

Together, these seven components specify the entire state of the abstract machine, written as

$\langle$I, S, V, C, L, E, D$\rangle$

The effect of each abstract machine instruction is described by a state transition rule. So, for example, the effect of the PUSH instruction is described by the rule:

$$\langle \mathsf{PUSH}\ m.\mathrm{I},\ n_0 \cdots\ n_m.\mathrm{S},\ \mathrm{V},\ \mathrm{C},\ \mathrm{L},\ \mathrm{E},\ \mathrm{D}\rangle \Rightarrow \langle \mathrm{I},\ n_m.n_0 \cdots\ n_m.\mathrm{S},\ \mathrm{V},\ \mathrm{C},\ \mathrm{L},\ \mathrm{E},\ \mathrm{D}\rangle$$

An important difference between the nonlinear G-machine and the conventional one is that the nonlinear G-machine *partitions* the graph with respect to the type of the vertices. The single graph, G, that appears in the state of the conventional G-machine is replaced by two graphs, C and L, in the state of the nonlinear G-machine. This partition is essential to support linear data structures. However, it is not the only difference between the two machines.

5.2 Destructive Case Analysis

Case analysis of both conventional and linear data structures is performed using a single CASE instruction. Two further instructions are used for accessing the components of a data structure: LSPLIT pushes the components of a linear data structure onto the stack, and CSPLIT does the same for a conventional one. The only difference between these two instructions is the effect that they have on the data structure node. The LSPLIT instruction destroys it — although the components can subsequently be accessed on the stack, the node itself has vanished from the graph. The understanding here is that its storage has also been recovered for re-use. The CSPLIT instruction behaves in a similar manner, but the node that it operates on remains in the graph. A small example serves to illustrate the use of the CASE and LSPLIT instructions to implement destructive case analysis. We can define the `tail` function on linear lists as

```
signature tail: (Llist !*a)->(Llist !*a);
tail (Lcons x xs) = xs
```

This function compiles into the following instructions:

```
tail: PUSH 0
      EVAL
      CASE (Lnil,L1) (Lcons,L2)
L1:   STOP
L2:   LSPLIT 2
      PUSH 1
      UPDATE 3
      POP 2
      UNWIND
```

The CASE instruction examines the value at the top of the stack and selects the appropriate label. If control reaches `L2` then the LSPLIT instruction destroys the `Lcons` node and pushes its components onto the stack.

5.3 The Protection Mechanism

In Section 3 we showed that the function `divide` is translated into a type-incorrect form in the nonlinear λ-calculus, and we also noted that it is impossible to translate this function into a form that exhibits both the linearity and the laziness suggested by the original definition. Our solution to this problem allows the transformation of lazy pattern-matching to take place unhindered. A run-time *protection mechanism* is then used to delay the destruction of linear nodes until they are no longer shared. This works as follows:

- when the linear node becomes shared, a *protection count* is attached to it. This protection count is set to the number of pointers sharing the node;
- every time the node is accessed by one of the selectors the protection count is decremented;
- when the node is accessed by one of the selectors and the protection count is one, it is destroyed.

The compiler determines the protection count by examining the lazily matched pattern. In the case of *divide*, for example, the protection count is two. This solution can easily be generalised to more complicated patterns than pairs. However, it only works properly when all of the variables in the pattern are used at least once. Otherwise it leads to a space leak because the protection count attached to the shared linear node is never reduced to one. This weakness means that lazy pattern-matching must be used with care. Nevertheless, we have found the protection mechanism to be a workable solution to an extremely difficult problem.

To implement the protection mechanism another variant of the SPLIT instruction must be added to the nonlinear G-machine. The PSPLIT instruction is like LSPLIT except that it respects the protection count of the linear graph node that it operates on. If the protection count is greater than one the PSPLIT instruction causes it to be decremented, but the node itself is still protected and so it remains in the graph. Otherwise, the node is destroyed. The compiler detects case-expressions that are being used to select components of shared linear values and it uses PSPLIT instead of LSPLIT for them.

Other run-time solutions to the problem of lazy pattern-matching are also possible. For example, the LSPLIT instruction could be modified to update all references to the components of the node that it destroys. There would then be no need for PSPLIT. Unfortunately, it is hard to implement such schemes using only the source-to-source transformations employed by the NLML compiler.

5.4 Heap Organisation and Garbage Collection

The nonlinear G-machine has two heaps: a *conventional heap* managed using the classic scheme suggested by Fenichel and Yochelson [6], and a *linear heap* which is is divided into a number of *free-lists*, one for each possible linear graph node size. There is also a *non-volatile storage area* for graph nodes representing compile-time constants such as integers and strings. This avoids having to allocate space for them on the heap whenever they are needed (see Figure 3). The conventional heap

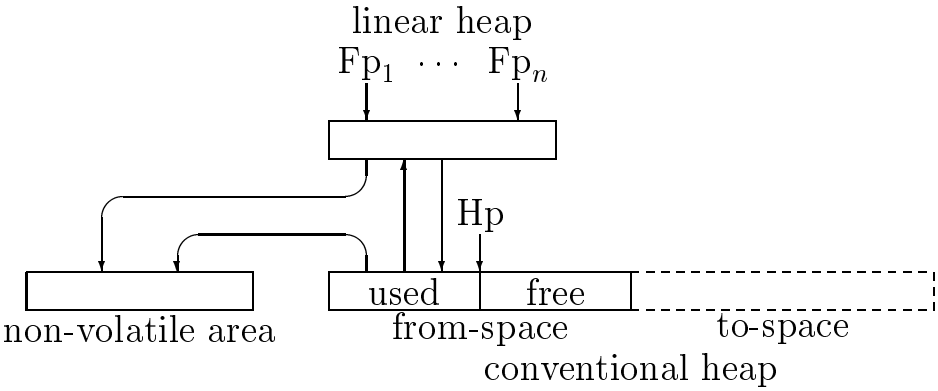

Figure 3: The conventional and linear heaps

supports the implicit destruction of disused nodes by garbage collection, while the free-list organisation of the linear heap supports the explicit destruction of disused nodes by destructive case-analysis.

5.5 The Simulated Stack and Free-list

One of the most important optimisations performed by Augustsson and Johnsson's LML compiler involves the use of a *simulated stack*. When an abstract machine instruction such as PUSHINT 3 is encountered, no machine code is emitted; instead the value 3 is pushed onto a simulated stack maintained by the code generator. During code generation, operands are taken from simulated stack if possible; otherwise some machine code is emitted to take them from the real stack at run-time.

The NLML compiler retains this optimisation and adds a *simulated free-list*. When a node is destroyed by destructive case analysis, no machine code is emitted to link it back onto the appropriate free-list; instead it is stored on a simulated free-list maintained by the code generator. During code generation, space for new linear nodes is allocated by re-using nodes on the simulated free-list if possible; otherwise code to acquire a new cell from the appropriate free-list must be emitted.

Some care is required in managing the simulated stack and the simulated free-list in the presence of destructive operations. For example, the components of a linear

node that has been the subject of destructive case analysis must be saved in registers or on the real stack, bypassing the simulated stack. Disaster ensues if references to them via the destroyed node appear on the simulated stack because this node may have been reallocated in the meantime. It is equally important to ensure that the simulated free-list is "flushed" to the real one to avoid space leaks.

5.6 Updating and Sharing

Apart from during type-checking, the nonlinear G-machine treats linear functions and their applications in exactly the same way as conventional ones. This means that the graph nodes representing all functions and their applications are stored in the C partition of the graph. But a problem arises when the result of a function application is a linear value. The node representing the application must be updated with the node representing the value in order to ensure lazy evaluation. The conventional G-machine updates by copying the root node of the result over application node. However, the nonlinear G-machine clearly cannot do this: that would make a nonsense of its attempts to keep graph nodes partitioned by type. Instead the compiler must arrange to update using an indirection node.

The instruction set of the nonlinear G-machine is largely borrowed from that of the conventional G-machine. This gives rise to an unexpected, but benign, form of sharing of linear values. As we have seen, the PUSH instruction works by *copying* a pointer to the top of the stack. If that pointer is to a linear value, then the linear value becomes shared. Fortunately, the linear value will only ever be accessed via this new pointer during execution. In fact, such sharing causes a problem only during garbage collection, where the garbage collector must be prepared to encounter pointers to linear nodes that have already been destroyed.

5.7 Benchmarks for List Structures

Four benchmark programs were written in NLML and compiled with our prototype compiler. With minor alterations (omitting the extra functions required to maintain linearity), the same four programs were written in LML and compiled with the Chalmers LML compiler, version 0.95. The four benchmark programs were as follows.

- *adder*: a gate-level simulation of a four-bit ripple-carry performing 20,000 additions.
- *mandelbrot*: generates a crude view of the Mandelbrot set on an ordinary terminal.
- *turtle*: draws 20 Hilbert curves using simple turtle graphics.
- *qsort*: a quicksort of a list of 4,000 random numbers.

The benchmark figures given below were recorded on a lightly loaded SUN 3/280 file-server with 16M bytes of memory running version 3.5 of the SUN UNIX operating system. For both implementations the total heap space was limited to 1Mbyte in order to force a significant number of garbage collections. The NLML compiler was instructed to generate code for a version of the nonlinear G-machine in which 20% of the total heap space was reserved for the linear heap, and the remainder was divided into two semispaces for the conventional heap. The LML compiler was similarly instructed to generate code for a version of the conventional G-machine in which the total heap space was divided into two semispaces for the single conventional heap. For each of the benchmarks we measured the total execution time, the amount of time spent garbage collecting, the number of garbage collections, and the amount of storage allocated from the heap(s). Execution time was measured in seconds, and the storage allocated was measured in bytes. Table 1 gives the results for the programs produced by the NLML compiler, and Table 2 gives the results for the programs produced by the LML compiler.

	execution time		GCs	heap storage allocated	
	total	GC		conventional	linear
adder	44.34	0.27	89	34,003,380	13,120,876
mandelbrot	29.73	0.08	36	14,553,948	3,098,952
turtle	44.32	0.70	68	10,806,240	1,931,280
qsort	17.98	3.17	23	7,026,112	5,037,568

Table 1: Results for NLML programs using linear lists

	execution time		GCs	heap storage allocated
	total	GC		conventional
adder	21.16	0.08	48	22,482,252
mandelbrot	20.18	0.03	22	11,116,536
turtle	40.24	0.59	54	10,754,160
qsort	15.30	5.19	34	8,915,200

Table 2: Results for LML programs using conventional lists

These benchmark figures are very disappointing. The performance of the NLML programs is generally much worse than that of the LML ones. There are several reasons for this.

The first is the need to apply extra functions to share linear values (by explicit copying) and to throw them away (by explicit case analysis). These functions are costly in terms of both space and time. The graphs representing their applications must be built in the heap and later garbage collected. This is a particular problem in both the *adder* and *mandelbrot* programs, and in both cases it is exacerbated by the fact that the extra functions are in the "inner loop" of the entire program. Notice the increase in the number of garbage collections in both cases.

One of the advantages of the conventional G-machine's semispace heap organisation is that only a single test for heap exhaustion has to be made before a sequence of allocations. The free-list organisation of the linear heap means that a test for free-list exhaustion must be made before every allocation. However, the cost of this test is actually quite low: omitting it produces only a 4% speed up at most. The reason is that the simulated free-list optimisation is quite effective: the speed up from using it can be as much as 11%.

Programming in NLML often involves passing around tuples instead of single values in order to maintain linearity. These tuples are expensive for two reasons. Firstly, there is the obvious cost of creating and destroying them. This cost should not be underestimated: in the *mandelbrot* program 12% of the storage allocated in the linear heap is for tuples whose sole purpose is to maintain linearity; in the *adder* program, the figure rises to 39%. Secondly, and more insidiously, these tuples exact a significant cost in terms of lost opportunities for compile-time optimisation. The upshot of this is that the nonlinear G-machine builds many graphs at run-time that were optimised away for the conventional G-machine at compile-time.

6 Aggregate Structures

Problems involving large aggregates, such as arrays and file systems, have always been something of a *bête noir* for function languages. It is difficult for the implementation to allow the aggregate to be updated while maintaining both referential transparency and acceptable efficiency. In the past, a number of solutions to this *aggregate update problem* have been proposed including run-time checks [10], syntactic restrictions [23], and abstract interpretation [13, 3]. In this section we shall give yet another solution by showing how the nonlinear type system can be used in the implementation of arrays.

6.1 Implementation

The literature describes two implementation techniques for conventional arrays, *trailered arrays* and *destructively-updated arrays*. Trailered arrays require run-time checks to maintain referential transparency when they are updated; destructively-updated arrays require compile-time checks. Bloss describes both techniques in [3]. She shows that trailered arrays are very expensive at run-time and that destructively-updated arrays are very expensive at compile-time.

Linear arrays offer a solution to this dilemma. If an array is declared to have a linear type then the single-threaded use of that array will be verified at compile-time by the type-checker; at run-time all updates to the array can then be done destructively.

A linear array is created in much the same way as conventional one. In NLML the operation

```
array: (List !*a)->(Array !*a)
```

allocates space for a linear array whose elements are drawn from a list. The update operation

```
update: (Array !*a)->Int-o!*a-o(Array !*a)
```

is similar to that for conventional arrays, but the index operation

```
index: (Array !*a)->Int-o(Xpair !*a (Array !*a))
```

is rather different. The nonlinear type system prevents a linear array from being thrown away, and so the index operation must return the array as part of its result. The type `Xpair` simply pairs a conventional value with a linear one.

```
linear type Xpair !*a *b = Xpr !*a *b
```

If an array really is to be disposed of, then this must be done explicitly. The operation

```
yarra: (Array !*a)->(List !*a)
```

destroys an array and returns a list of its elements.

From this brief description of the linear array primitives it is not obvious how the destructive update operations are sequenced so that the program remains "safe". Consider the function `swap` which swaps two elements in a linear array.

```
swap: (Array !*a)->Int-oInt-o(Array !*a);
swap a i j =
    let (Xpr x a) = index a i  in
    let (Xpr y a) = index a j  in
    update (update a i y) j x
```

Here, the data dependencies are all that is required to ensure that the indexing operations are performed prior to the updates. No extra machinery, such as the sequential `let!` of [26] or the sequential `let*` of [9] is necessary.

Although the most obvious place to store linear arrays would be in the linear heap, our implementation actually stores them in the conventional heap and garbage collects them because of the complexities of managing a free-list for objects whose size cannot be determined at compile-time.

6.2 Benchmarks for Arrays

Four benchmark programs were written in NLML and compiled with our prototype compiler. With minor alterations (no result tuples were used to maintain linearity), the same four programs were written in LML and compiled with two modified versions of the Chalmers LML compiler, the first with trailered arrays and the second with destructively-updated arrays. In all cases, a strict `let` was used to sequence the update operations correctly.

The four benchmark programs were as follows.

- *histogram*: counts occurrences in a list of 40,000 random decimal digits.
- *warshall*: finds the transitive closure of 18 identical 26-node graphs.
- *life*: charts the evolution of 20 generations of a small colony of cells on a 32×32 board.
- *qsort*: an in-place quicksort of an array of 4,000 random numbers.

The benchmarks were recorded in the same conditions as those described in Section 4, except that the total heap size was raised to 2M bytes. Table 3 gives the results for the programs produced by the NLML compiler using linear arrays, Table 4 gives the results for the programs produced by the LML compiler using trailered arrays and Table 5 gives the results for the programs produced by the LML compiler using destructively-updated arrays.

	execution time		GCs	heap storage allocated	
	total	GC		conventional	linear
histogram	11.43	6.39	17	4,960,544	640,000
life	44.23	18.13	29	18,937,568	3,605,440
warshall	14.94	0.24	13	10,421,928	2,905,344
qsort	9.40	0.48	7	4,303,620	2,280,928

Table 3: Results for NLML programs using linear arrays

	execution time		GCs	heap storage allocated
	total	GC		conventional
histogram	13.35	7.93	21	5,920,544
life	24.83	1.11	24	17,895,280
warshall	15.23	0.26	15	11,816,424
qsort	9.27	0.61	8	4,632,828

Table 4: Results for LML programs using trailered arrays

	execution time		GCs	heap storage allocated
	total	GC		conventional
histogram	11.01	6.34	17	4,960,544
life	23.81	0.98	23	17,428,336
warshall	13.52	0.20	13	10,421,928
qsort	8.22	0.52	7	4,303,620

Table 5: Results for LML programs using destructively-updated arrays

We found that linear arrays are usually faster than trailered arrays, but slower than destructively-updated ones. They fare particularly badly in programs like *qsort* and *life* — both of these programs perform an order of magnitude more indexing than updating operations. The reason for this, of course, is the need to pass tuples around in order to maintain linearity. As was the case with fine-grained data structures, these tuples exact a high cost in terms of lost optimisations at compile-time and extra work that must be performed at run-time.

7 Related Work

One of the things that prompted Wadler to develop the nonlinear λ-calculus and the nonlinear type system was the observation that languages based on the linear λ-calculus have several shortcomings. We know of two such languages, one developed by Lafont and the other by Holmström.

In his thesis [17], Lafont describes the implementation of a small functional programming language based on the linear λ-calculus. However, instead of attempting to apply what he calls the "brutish" compilation scheme of his published papers [8, 16] to an ordinary functional language, he designs his own linear functional language, called LIVE. This language exposes the programmer to the full rigours of the linear λ-calculus, and only a few "small" types (such as integers) are permitted to escape the linearity constraint. LIVE is implemented using a linear variant of the Categorical Abstract Machine [4] called the Linear Abstract Machine. The advantage of this machine is that it does not require garbage collection. However, there are also two obvious disadvantages: the grain of reduction is very small (similar to that of the SK reduction machine [24]), and the results of computations are never shared because the machine was designed to implement a linear language without any sharing at all. This can be very inefficient.

Holmström [11] has described another functional programming language based on the linear λ-calculus. In his language all functions and data types inherit the linearity constraint from the linear λ-calculus. However, Holmström considers this constraint to be unacceptable in general, and so he provides a way to lift it which works for all types, not just those with a propitious machine representation. Unfortunately,

Holmström has found that his language still exhibits the same fundamental lack of flexibility that Lafont's does [12]. Holmström sketched an implementation of his language using a linear variant of Landin's SECD machine [18]. This machine performs direct interpretation of the program source code. It is of interest because it allows a restricted form of sharing, and thus requires garbage collection. However, there is still no mechanism for sharing the result of a computation and, as we remarked before, this can be very inefficient.

Guzmán and Hudak [9] have developed a variant of the λ-calculus capable of expressing destructive operations, together with a type system which ensures that these operations do not compromise referential transparency. In their paper, they reject an approach based purely on linear logic, such as the one described here, as being too constraining. Instead, their type system captures the notion of state by annotating the type of each function in one of seven possible ways to indicate how it uses its argument. The resulting type system is more complex than the non-linear one, but it controls destructive operations with considerably more precision, allowing non-destructive operations in contexts where destructive ones are permitted (but not vice versa). Guzmán and Hudak [9] plan to implement an extension of Haskell [14] based on their ideas.

Wadler has proposed a `let!`-expression based on the observation that it is perfectly safe to have more than one reference to a linear value temporarily, so long as only one reference exists when it is updated. The expression

```
let! (x) y = u in v
```

is used to grant "read-only" access to a linear value `x` within `u`. Unfortunately, this construction comes with a number of extremely ad-hoc restrictions: for example, the evaluation of `u` should be hyperstrict, and it must not be possible for `u` (or any component of `u`) to evaluate to `x` (or any component of `x`). Wadler formalises these requirements [26], but he is still unsure of how they relate to any existing theory [27]. We have avoided this construction on the grounds of its complexity, its poor interaction with lazy evaluation, and its dubious theoretical foundation. Wadler is currently trying to bridge the gap between theory and practice; his latest paper [25] attempts to establish the connection between the theoretically-based work of Lafont and Holmström and the more practically-based work of Guzmán and Hudak.

A more detailed account of our own work can be found in the first author's DPhil thesis [28].

8 Conclusions and Future Work

In this paper we have described the design and implementation of a functional language based on Wadler's approach to problems involving changes of state. Our

work has revealed a number of drawbacks of the approach. Broadly, these are as follows.

The right to use destructive operations is accompanied by the onerous responsibility to maintain linearity. This leads to a significant loss of flexibility, and programming in NLML is rather difficult. What is needed is some extension of the underlying non-linear λ-calculus such as the read-only access granted by Wadler's `let!`-expression. Unfortunately, the logical foundation of this expression is unclear and so there is a need for further research to find either a logical justification of the `let!`-expression, or some similar construction with such a justification. Another possible approach would be to follow Guzmán and Hudak [9] by adopting a more sophisticated type system whose connection with linear logic is of a looser kind.

The nonlinear G-machine is an attempt to implement NLML using the graph reduction technique that is used so successfully in the implementation of LML. Its performance is disappointing. More research is needed into abstract machine architectures suitable for implementing languages like NLML. We particularly favour an approach based on a machine that can make better use of sharing information, such as the TIM [5] or the Spineless Tagless G-machine [22].

It has been suggested to us that it might be better to dispense with the run-time machinery for dealing with linear values. Instead, all values would be stored in a single garbage-collected heap. Destructive operations on linear values would then be restricted to those which could be detected at compile-time using the simulated free-list. With this restriction, it would be pointless to pass tuples around to maintain linearity, and so there would be no lost optimisations at compile-time or extra data structures created at run-time. This suggestion amounts to a weakening of the linearity constraint: linear values still cannot be shared, but they can now be thrown away. In other words, they are *single-threaded* [23]. Single-threaded type systems are a promising area for future research, but they represent a departure from Wadler's original proposal which we have not investigated yet.

Our linear array implementation shows some promise. Linear arrays are usually more efficient than trailered ones, but they are less efficient than destructively-updated ones. We are firmly convinced that larger aggregates are the most promising area of application for the nonlinear type system, and we intend to continue our work on arrays and file systems.

Acknowledgements

We have benefitted greatly from correspondence with Lennart Augustsson and Thomas Johnsson, whose work on the LML compiler and the G-machine served as the basis for much of our own, and from correspondence with Yves Lafont, Sören Holmström and Phil Wadler, whose papers first introduced us to linear logic. We are also grateful for the comments of Neil Jones, Paul Hudak and Simon Peyton Jones.

Wakeling was funded by a research studentship from the Science and Engineering Research Council of Great Britain.

References

[1] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Chalmers University of Technology, S-412 96 Göteborg, November 1987.

[2] L. Augustsson and T. Johnsson. *Lazy ML Users Manual*, July 1989. (Distributed with the LML compiler, version 0.95).

[3] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, pages 26–38. ACM Press, September 1989.

[4] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[5] J. Fairbairn and S. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, pages 34–45. Springer-Verlag, September 1987. LNCS 274.

[6] R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *CACM*, 12(11):611–612, November 1969.

[7] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[8] J.-Y. Girard and Y. Lafont. Linear logic and lazy computation. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'87)*, pages 52–66. Springer-Verlag, March 1987. LNCS 250.

[9] J. C. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the Fifth Annual IEEE Symposium on Logic In Computer Science*, pages 333–343, June 1990.

[10] S. Holmström. A simple and efficient way to handle large data structures in applicative languages. In *Proceedings of the SERC/Chalmers Workshop on Declarative Programming*, pages 185–187. University College London, April 1983.

[11] S. Holmström. A linear functional language. In *Proceedings of the Workshop on the Implementation of Lazy Functional Languages, Aspenäes*, pages 13–32, September 1988. Report 53, Programming Methodology Group, Chalmers University of Technology, S-412 96 Göteborg.

[12] S. Holmström. Quicksort in a linear functional language. PMG Memo. 65, Chalmers University of Technology, S-412 96 Göteborg, January 1989.

[13] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987.

[14] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical report, University of Glasgow, Department of Computer Science, April 1990.

[15] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, S-412 96 Göteborg, February 1987.

[16] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[17] Y. Lafont. *Logiques, Catégories et machines.* PhD thesis, Université de Paris 7, 1988.

[18] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[19] B. B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman, 1983.

[20] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[21] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[22] S. L. Peyton Jones and J. Salkild. The Spineless Tagless G-machine. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, pages 184–201. ACM Press, September 1989.

[23] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[24] D. A. Turner. A new implementation technique for applicative languages. *SOFTWARE — Practice and Experience*, 9(1):31–50, January 1979.

[25] P. Wadler. Is there a use for linear logic? Technical report, Department of Computing Science, University of Glasgow, December 1990.

[26] P. Wadler. Linear types can change the world! In *IFIP Working Conference on Programming Concepts and Methods*, Sea of Gallilee, Israel, April 1990.

[27] P. Wadler. Private communication, February 1990.

[28] D. Wakeling. Linearity and laziness. DPhil thesis, Department of Computer Science, University of York, November 1990. Technical Report YCST 90/07.