J. Functional Programming **7** (5): 541–547, September 1997. Printed in the United Kingdom

More haste, less speed: lazy versus eager evaluation

RICHARD BIRD, GERAINT JONES and OEGE DE MOOR
Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Abstract

Nicholas Pippenger has recently given a problem that, under two simple restrictions, can be solved in linear time by an impure Lisp program, but requires $\Omega(n \log n)$ steps to be solved by any eager pure Lisp program. By showing how to solve the problem in linear time with a lazy functional program, we demonstrate that – for some problems at least – lazy evaluators are strictly more powerful than eager ones.

Capsule Review

This paper extends a result of Pippenger that shows that programs using mutation can (under certain conditions) perform some computations more efficiently that programs not using mutation. Pippenger's programs all employ eager evaluation, and this present paper show that the the blame may be laid at that door as well, by showing that the computations Pippenger describes can be performed efficiently by programs using lazy evaluation but no mutation. Of course lazy evaluation is usually implemented by using mutation behind the scenes in the form of 'memoization' to avoid redundant re-evaluations. Thus, this paper raises the interesting question as to the extent to which this restricted form of mutation can achieve the efficiencies obtainable with general mutation.

1 Introduction

In a recent paper, Pippenger (1997) proves that impure Lisp – with mutable variables and assignment statements – is strictly more powerful than pure Lisp. He exhibits a task that can be done in constant amortised time with an impure Lisp program, but which requires $\Omega(n \log n)$ steps in one written without using explicit assignments. His proof of the lower bound makes use of the eagerness of pure Lisp evaluators; in this note we show that a lazy functional program can solve the problem with the same efficiency as an impure Lisp program.

In outline, Pippenger's example program is required to apply a given permutation repeatedly to groups of symbols drawn from a potentially infinite sequence of inputs. The heart of the computation is the application of a function *doperms*, which might be defined by:

$$\begin{aligned} &\mathit{doperms} &&:: \mathit{Int} \rightarrow [\mathit{Int}] \rightarrow [a] \rightarrow [a] \\ &\mathit{doperms}\ n\ ps &&= \mathit{concat} \cdot \mathit{map}\ (\mathit{perm}\ ps) \cdot \mathit{group}\ n. \end{aligned}$$

The list *ps* is an encoding of a permutation on n-tuples, and *perm ps* applies that permutation to a single n-tuple represented as a list of length n, so for example

$$perm\ [2,3,1,0]\ \text{“}\texttt{abcd}\text{”} = \text{“}\texttt{cdba}\text{”}.$$

The function *group n* divides a potentially infinite list into a list of n-tuples, each represented by a list of length n, so for example

$$\begin{aligned} &group\ 4\ \text{“}\texttt{abcdefghijklmnop}\text{”} = [\text{“}\texttt{abcd}\text{”}, \text{“}\texttt{efgh}\text{”}, \text{“}\texttt{ijkl}\text{”}, \text{“}\texttt{mnop}\text{”}] \\ &doperms\ 4\ [2,3,1,0]\ \text{“}\texttt{abcdefghijklmnop}\text{”} \\ &\qquad\qquad = \text{“}\texttt{cdbaghfekljiopnm}\text{”}. \end{aligned}$$

These and other standard functions used in this paper are defined in Figure 1.

The functions *concat* and *group n* each take time linear in the length of the input consumed. It is easy to see that *perm ps* takes $O(n^2)$ reduction steps, where n is the length of *ps*. A more sophisticated implementation of *perm* could bring the time down to $O(n \log n)$ steps, however the precise running time of *perm* will be unimportant so long as it is no greater than quadratic.

Throughout this paper, purely for clarity of expression, we use the type *Int* for integers and write arithmetic operations as though they were the familiar constant-cost operations on fixed-sized integers usually provided by real implementations. However, they should be read as though they are the corresponding operations on (unbounded) unary numbers; so for example calculating $2 \times n$ from the value of n takes $O(n)$ steps in the calculation.

This is different from restricting ourselves to data constructed with constructors that have bounded arity, and constant-time access to the components of those constructors. Were we allowed arrays of unbounded size with constant time access to the elements Pippenger's result would not apply.

2 The problem to be solved

To distinguish the contribution of assignment to the time complexity of programs, Pippenger imposes two constraints: that the computation be both *on line* and *symbolic*.

That a computation be on line means that for each m, the m-th output should be capable of being produced by the computation before the $(m+1)$-th input is available. In the context of lazy functional programming this means that the program, say *machine*, should be a function $[a] \to [a]$, for which each initial segment of the result depends only on the initial segment of the argument of the same length,

$$take\ m \cdot machine = take\ m \cdot machine \cdot (+\!\!+\ undefined) \cdot take\ m$$

for all m.

As defined above, *doperms n ps* could not be on line for all arguments *ps*, in particular if *ps* represents the reverse permutation, the first element of a reversed n-tuple cannot be output until all n of the components have been read. To guarantee that an on-line solution is possible *machine* is defined to interleave inputting with outputting, adding a copy of each significant symbol of its input to the output, and

```
perm                     :: [Int] → [a] → [a]
perm ps                  = zipwith index ps · repeat
                             where repeat x = xs where xs = x : xs

group                    :: Int → [a] → [[a]]
group n                  = unfold (not · null) (take n) (drop n)

concat                   :: [[a]] → [a]
concat                   = foldr (++) [ ]

index                    :: Int → [a] → a
index n                  = head · drop n

take, drop               :: Int → [a] → [a]
take 0       xs          = [ ]
take (n + 1) [ ]         = [ ]
take (n + 1) (x:xs)      = x: take n xs
drop 0       xs          = xs
drop (n + 1) [ ]         = [ ]
drop (n + 1) (x:xs)      = drop n xs

takewhile, dropwhile     :: (a → Bool) → [a] → [a]
takewhile p     [ ]      = [ ]
takewhile p (x:xs)       = x: takewhile p xs, if p x
                         = [ ], otherwise
dropwhile p     [ ]      = [ ]
dropwhile p (x:xs)       = dropwhile p xs, if p x
                         = x:xs, otherwise

foldr                    :: (a → b → b) → b → [a] → b
foldr f z     [ ]        = z
foldr f z (x:xs)         = f x (foldr f z xs)

unfold                   :: (a → Bool) → (a → b) → (a → a) → (a → [b])
unfold p h t             = map h · takewhile p · iterate t
                             where iterate f x = x: iterate f (f x)

map                      :: (a → b) → ([a] → [b])
map f     [ ]            = [ ]
map f (x:xs)             = f x: map f xs

zipwith                  :: (a → b → c) → ([a] → [b] → [c])
zipwith f    [ ]     ys  = [ ]
zipwith f (x:xs)     [ ] = [ ]
zipwith f (x:xs) (y:ys)  = f x y: zipwith f xs ys

null                     :: [a] → Bool
null    [ ]              = True
null (x:xs)              = False

head                     :: [a] → a
head (x:xs)              = x

tail                     :: [a] → [a]
tail (x:xs)              = xs

const                    :: a → b → a
const x y                = x
```

Fig. 1. Functions used in the programs in the text.

dually reading and ignoring a dummy symbol of input for each significant symbol of the output. These extra transactions are immaterial to the real computation being performed, and are present only to 'clock' the computation.

To accommodate these extra transactions *doperms* is modified to be:

$$\begin{aligned} \mathit{doperms}'\,n\,ps &= \mathit{concat}\cdot\mathit{map}\,(\mathit{perm}'\,n\,ps)\cdot\mathit{group}\,(2\times n)\\ &\quad\textbf{where}\ \mathit{perm}'\,n\,ps = \mathit{echo}\,(\mathit{perm}\,ps)\cdot\mathit{take}\,n\\ &\quad\phantom{\textbf{where}}\ \mathit{echo}\,f\,xs\ \ = xs +\!\!\!+\, f\,xs \end{aligned}$$

so that, for example,

$$\mathit{doperms}\ 4\,[2,3,1,0]\ \text{“}\texttt{abcdefghijklmnop}\text{”} = \text{“}\texttt{abcdcdbaijlkklji}\text{”}.$$

Note that, although this function *doperms′* is on line, it is does not run in *constant amortised time*, which is to say that it cannot produce m items of output within a constant number of times m steps. To see this, observe that an output of length m requires $O(m/n)$ applications of *perm*, and so takes $O(m\,n)$ steps if *perm* takes $O(n^2)$ steps. Even if an $O(n \log n)$ implementation of *perm* were substituted, the computation would still require $O(m \log n)$ steps.

That a computation be symbolic means essentially that the function being computed should be fully polymorphic in the type of the list being processed, excepting only that list elements may be compared for equality. In a language with type classes this means that the function *machine* should have type $Eq\ a \Rightarrow [a] \rightarrow [a]$.

The function used as a touchstone in Pippenger's paper is made symbolic by its reading of a *prologue* which encodes the permutation to be applied. In Pippenger's presentation, the prologue consists of *True* and *False* symbols – we choose to pass the constant symbols used for these as parameters to our implementation. The prologue uses these symbols to represent natural numbers, each encoded in unary notation as a run of *True* symbols followed by a *False*. The prologue consists of a representation of the length n of the permutation, followed by a sequence of n unary numbers which represent the permutation, and so is $O(n^2)$ symbols long.

Thereafter, the computation proceeds in the phases described by *doperms′*, with each phase consisting of reading n additional symbols from the input while echoing them to the output, and then producing the corresponding permutation as specified in the prologue while discarding n further dummy inputs.

```
machine           :: Eq a ⇒ (a, a) → [a] → [a]
machine (t, f) xs  = head numbers ++ concat (take n (tail numbers)) ++
                          doperms′ n ps (index (n + 1) prologue)
                     where n          = head ns
                           ps         = take n (tail ns)
                           ns         = map unary prologue
                           numbers    = map number prologue
                           prologue   = iterate after xs
                           number xs  = takewhile (= t) xs ++ [stop xs]
                           unary      = length · takewhile (= t)
                           stop       = head · dropwhile (= t)
                           after      = tail · dropwhile (= t)
```

For example, with spacing added to emphasize the structure,

$$\begin{aligned} &\textit{machine}\ (\text{‘}\mathtt{a}\text{’}, \text{‘}\mathtt{b}\text{’})\ \text{“}\mathtt{aaaab\ aab\ aaab\ ab\ b\ 0123\ xxxx\ 4567\ xxxx}\text{”} \\ &\quad = \text{“}\mathtt{aaaab\ aab\ aaab\ ab\ b\ 0123\ 2310\ 4567\ 6754}\text{”}. \end{aligned}$$

In this example, the prologue describes the permutation $[2, 3, 1, 0]$, and there are two phases: $[0, 1, 2, 3] \to [2, 3, 1, 0]$ and $[4, 5, 6, 7] \to [6, 7, 5, 4]$.

Pippenger shows that *machine* can be implemented on line in constant amortised time in impure Lisp, that is to say he shows how to construct a program which produces the first m elements of $\textit{machine}\,(a, b)\,xs$ in $O(m)$ steps for all m, independently of n. However he also shows that there is no constant amortised on-line symbolic pure Lisp program which does this: indeed no on-line pure Lisp program can produce the first m elements of the corresponding output in less than $\Omega(m \log n)$ steps. We will now construct a lazy functional program that can implement *machine* on line in constant amortised time.

3 A constant amortised time on-line lazy implementation

Crucial to the fast lazy program is the observation that instead of repeatedly applying a permutation to groups of n symbols, the same result can be obtained by one application of the permutation to a group of n sequences of symbols. The sequence of lists which are to be permuted is transposed, the transposed list of sequences is permuted once, and the permuted list of sequences is transposed back again. More precisely, this follows from the observation that for any sufficiently well-behaved polymorphic function $f :: [a] \to [a]$,

$$\textit{map}\ f \cdot \textit{trans}\ =\ \textit{trans} \cdot f,$$

where *trans* is the function which transposes a list of lists, and so provided that all the lists of lists considered as arguments are ‘rectangular’,

$$\begin{aligned} \textit{map}\ f &= \textit{map}\ f \cdot \textit{trans} \cdot \textit{trans} \\ &= \textit{trans} \cdot f \cdot \textit{trans}. \end{aligned}$$

The term $\textit{perm}'\,n\,ps$ applied to lists of length $2n$ has just this property so the definition of $\textit{doperms}'$ given above can be replaced by

$$\textit{doperms}'\,n\,ps\ =\ \textit{concat} \cdot \textit{trans} \cdot \textit{perm}'\,n\,ps \cdot \textit{trans} \cdot \textit{group}\,(2 \times n).$$

With care, the function *trans* can be implemented to make a *machine* using this function work on line in constant amortised time on arbitrarily long arguments.

The leftmost occurrence of *trans* in the definition of $\textit{doperms}'$ has to turn a $2n$-tuple of potentially infinite lists into a infinite list of $2n$-tuples, producing the whole of each $2n$-tuple before inspecting the tail of any component. The rightmost occurrence of *trans* has to have the complementary property when turning an infinite list of $2n$-tuples into a $2n$-tuple of infinite lists. Both of these requirements are met by defining

$$\begin{aligned} &\textit{trans}\ ::\ [[a]] \to [[a]] \\ &\textit{trans}\ = \textit{foldr}\ (\textit{zipwith}'\,(:))\,(\textit{repeat}\ [\,]) \end{aligned}$$

where the function *zipwith′* is defined by

$$\begin{array}{lll} zipwith' & :: & (a \to b \to c) \to ([a] \to [b] \to [c]) \\ zipwith'\, f \;\; [\,]\;\; ys & = & [\,] \\ zipwith'\, f\, (x\!:\!xs)\, ys & = & f\, x\, (head\ ys) : zipwith'\, f\, xs\, (tail\ ys) \end{array}$$

to agree with *zipwith* where they are both defined, but also to be non-strict in the second list argument.

$$\begin{array}{l} zipwith'\, f\, xs \;=\; zipwith\, f\, xs \cdot unstrictlist \\ \qquad\qquad\qquad \mathbf{where}\ unstrictlist \;=\; unfold\, (const\ True)\, head\ tail. \end{array}$$

Were this not the case, the transposition of an infinite list would in general be undefined.

This revised implementation of *doperms′* can compute outputs of length $2nk$ in $O(t(n) + 2nk)$ steps for all $k \geq 1$, where $t(n)$ is the cost of one application of *perm ps* with a permutation *ps* of length n. The effect of the double transposition is that *perm* is applied only once, and the application of the permutation is evaluated fully during the first phase, when the first $2n$ elements of the output are computed.

Although this might at first sight seem to fall short of constant amortised time behaviour, since there must be more than $O(n)$ steps involved in computing the first $2n$ post-prologue outputs of *machine*, this does not matter: provided that $t(n)$ is no greater than $O(n^2)$ the extra time needed can be attributed to the $O(n^2)$ symbols read and written during the prologue. Thus with this *doperms′* component the lazy implementation of *machine* produces outputs of length m in time $O(m)$, independent of the size n of the permutation.

4 Lazy versus normal-order

It is well known that normal-order reduction can be simulated in an eager language by systematic translation. Eager evaluation of the resulting program corresponds to normal-order reduction of the original program. The translation might be applied to the program for *machine* to yield an on-line eager program which consumes and generates streams.

This does not contradict Pippenger's result since the translation does not preserve time complexity. More specifically: if the original program is not syntactically linear, the normal-order translation executed by an eager evaluator may execute the same sub-computation more than once. For example, the second list argument appears twice in

$$zipwith'\, f\, (x : xs)\, ys \;=\; f\, x\, (head\ ys) : zipwith'\, f\, xs\, (tail\ ys)$$

which is essentially not linear. Hence *trans* is not linear and a normal-order evaluation of our solution would re-evaluate the permutation (and everything involved in building the structure to be permuted) for each of the $2n$-tuples of the output.

It might appear that the non-linearity in the definition of *trans* could be eliminated by defining *zipwith′* in terms of *zipwith* and *unstrictlist*, but *unstrictlist* uses *iterate* and the standard definition of *iterate*,

$$iterate\ f\ x = x : iterate\ f\ (f\ x)$$

is clearly not linear. This particular non-linearity can be eliminated by transforming the definition into

$$iterate\ f\ x = fix\ ((x{:}) \cdot map\ f)\ \textbf{where}\ fix\ f = y\ \textbf{where}\ y = f\ y$$

but now this is not linear unless the equation $y = f\ y$ is implemented by the construction of a circular data-object y.

The inefficiency of normal-order evaluation of non-linear programs is exactly what is eliminated in a lazy evaluator: whenever an identified – and possibly shared – expression is evaluated, the closure for that expression is replaced by its value. Pippenger's result shows that in order to eliminate this inefficiency there must be some mechanism added to an eager evaluator which he excludes from his model of a pure Lisp evaluator. Such mechanisms are the definition of circular data-objects of unbounded size; assignment; memoization of the results of function application; or of course the overwriting of a closure by its value. This last is essentially a restricted memoization.

Epilogue

We have shown that in applying the function *machine*, just those assignments necessary to implement a lazy evaluator are sufficient to reduce the lower-bound complexity of one particular on-line program. Pippenger observes that our strategy could be applied by an eager evaluator if he were to relax the on-line constraint; thus the delaying of evaluation necessary to implement a lazy evaluator is sufficient to bring a batch program on line.

Note that we do not, and cannot, claim that a lazy implementation can solve all problems with the same efficiency as an impure Lisp solution. We have, however, shown that without assignments or circular data structures there is no complexity-preserving translation into an eager language of arbitrary programs written in a lazy one.

We are grateful for the assistance of a number of referees and for particularly insightful comments from Chris Okasaki.

References

Pippenger, N. (1997) Pure versus Impure Lisp. *ACM TOPLAS*, **19**(2), March, 223–238. (This is an extended version of a paper in *23rd ACM Sigplan-Sigact conference on the Principles of Programming Languages (POPL'96)*, pp. 104–109. ACM Press.)