More haste, less speed: lazy versus eager evaluation

Richard Bird, Geraint Jones, Oege de Moor
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom

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.

1 Introduction

In a recent paper Pippenger (1996) 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 real time with an impure Lisp program, but which requires $\Omega(n \log n)$ steps in one written without using 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:

$$doperms\ n\ ps \quad = \quad concat \cdot map\ (perm\ ps) \cdot group\ n.$$

The list ps is an encoding of a permutation on n-tuples, and $perm\ ps$ applies that permutation to a single n-tuple. The function $group\ n$ divides a potentially infinite list into a list of n-tuples, each represented by a list of length n. These subsidiary functions could be defined by:

$$\begin{array}{lcl} perm & :: & [Int] \to [a] \to [a] \\ perm\ ps & = & zipwith\ index\ ps \cdot repeat \\ & & \textbf{where}\ \ index\ n \quad = \quad head \cdot drop\ n \\ group & :: & Int \to [a] \to [[a]] \\ group\ n & = & unfold\ (not \cdot null)\ (take\ n)\ (drop\ n) \end{array}$$

The standard functions *zipwith* and *unfold* are defined by:

$$
\begin{array}{lcl}
\mathit{zipwith} & :: & (a \to b \to c) \to ([a] \to [b] \to [c]) \\
\mathit{zipwith}\ f\ [\,]\ ys & = & [\,] \\
\mathit{zipwith}\ f\ (x : xs)\ [\,] & = & [\,] \\
\mathit{zipwith}\ f\ (x : xs)\ (y : ys) & = & f\ x\ y : \mathit{zipwith}\ f\ xs\ ys \\
\\
\mathit{unfold} & :: & (a \to \mathit{Bool}) \to (a \to b) \to (a \to a) \to (a \to [b]) \\
\mathit{unfold}\ p\ h\ t & = & \mathit{map}\ h \cdot \mathit{takewhile}\ p \cdot \mathit{iterate}\ t
\end{array}
$$

Definitions of other standard functions and operators are omitted here.

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 $O(n^2)$ 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.

2 The problem to be solved

In order 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 produced by the computation before the $(m+1)$-th input is received. In the context of lazy functional programming this means that the program, *machine* say, should be a *non-destructive* function, which is to say that

$$\mathit{take}\ m \cdot \mathit{machine} \quad = \quad \mathit{machine} \cdot (+\!\!+\ \mathit{undefined}) \cdot \mathit{take}\ m$$

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. In order 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 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:

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

Note that although this function *doperms* is on line, it is not *real time*, which is to say that it does not produce each output within a bounded number of steps after consuming the corresponding input. 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

$$machine \quad :: \quad Eq\, a \Rightarrow [a] \to [a].$$

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. Two distinct symbols, say a and b, are first read from the input: these are to be used to represent natural numbers in what follows. The numbers are each encoded in unary notation as runs of a symbols terminated by a single b. After a and b, a representation of the length n of the permutation is read, followed by a sequence of n unary numbers which represent the permutation. The prologue ends with sufficient additional inputs to bring the total number of symbols read up to $2 + n + n^2$. (These additional symbol are ignored, and seem not to be essential to the complexity result, but we will duplicate the behaviour anyway.)

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 printing the corresponding permutation as specified in the prologue while discarding n further dummy inputs.

$$\begin{array}{lcl}
machine\ xs & = & [a, b] \mathbin{+\!\!+} \\
 & & takewhile\ (= a)\ ys \mathbin{+\!\!+} [head\ (dropwhile\ (= a)\ ys)] \mathbin{+\!\!+} \\
 & & take\ (n \times n)\ (after\ ys) \mathbin{+\!\!+} \\
 & & doperms\ n\ ps\ (drop\ (2 + n + n \times n)\ xs) \\
 & & \textbf{where}\ a = head\ xs \\
 & & \qquad b = head\ (tail\ xs) \\
 & & \qquad ys = tail\ (tail\ xs) \\
 & & \qquad n = head\ ns + 1 \\
 & & \qquad ps = take\ n\ (tail\ ns) \\
 & & \qquad ns = unfold\ (const\ True)\ unary\ after\ ys \\
 & & \qquad unary = length \cdot takewhile\ (= a) \\
 & & \qquad after = tail \cdot dropwhile\ (= a)
\end{array}$$

For example, with spacing added to emphasize the structure,

$$\begin{array}{rcl}
machine & \text{“a b aaab aab aaab ab b xxxxxx 0123 xxxx 4567 xxxx”} \\
 & = & \text{“a b aaab aab aaab ab b xxxxxx 0123 2310 4567 6754”}.
\end{array}$$

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 real time in impure Lisp, that is to say he shows how to construct a program which outputs the first m elements of *machine xs* in $O(m)$ steps for all m, independently of n. However he also shows that there is no real-time 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 $\Omega(m \log m)$ steps. We will now construct a lazy functional program that can implement *machine* on line in real time.

3 A real-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 sequence is transposed back again. Formally, this is follows from the observation that if f is a polymorphic shape-injective function then

$$trans \cdot f \quad = \quad map\ f \cdot trans.$$

The term $echo\ (perm\ ps) \cdot take\ n$ applied to lists of length $2n$ has just this property so the definition of *doperms* above can be replaced by

$$doperms\ n\ ps \quad = \quad concat \cdot trans \cdot echo\ (perm\ ps) \cdot take\ n \cdot trans \cdot group\ (2 \times n).$$

With care, the function *trans* can be written to make this implementation work on line in real time on infinite arguments.

The first occurrence of *trans* in the definition of *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 second 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

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

where the function $zipwith'$ is defined by

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

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

$$\begin{array}{lcl} zipwith'\ f\ xs & = & zipwith\ f\ xs \cdot unstrictlist \\ & & \textbf{where}\ unstrictlist \quad = \quad unfold\ (const\ True)\ head\ tail \end{array}$$

Were this not the case, the transposition of an infinite list would 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 real-time behaviour, since

there may 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. After currying each function in the given program, each application can be rewritten in the form

$$M\,N \quad \longrightarrow \quad (\mathit{applyto}\,N)\,M.$$

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

$$\mathit{zipwith}'\,f\,(x : xs)\,ys \quad = \quad f\,x\,(\mathit{head}\,ys) : \mathit{zipwith}'\,f\,xs\,(\mathit{tail}\,ys)$$

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 the standard definition of *iterate*

$$\mathit{iterate}\,f\,x \quad = \quad x : \mathit{iterate}\,f\,(f\,x)$$

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

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

but 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

in *letrec*-like constructs; 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, and has the advantage over general memoization of being implementable at a negligible additional cost by a sequential normal-order evaluator.

5 Epilogue

We have shown that in applying the function *machine*, the assignments necessary to implement a lazy evaluator are sufficient to reduce the lower-bound complexity of the program. 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 do, however, appear to have shown that – without exploiting assignment or the definitions of recursive data-objects – there is no complexity-preserving translation into an eager language of arbitrary programs written in a lazy one.

Reference

Pippenger, Nicholas. (1996). Pure versus Impure Lisp. In *23rd ACM Sigplan-Sigact conference on the Principles of Programming Languages (POPL'96)*, pp. 104–109. ACM Press.