Acta Informatica 21, 239-250 (1984)

Acta Informatica

© Springer-Verlag 1984

Using Circular Programs to Eliminate Multiple Traversals of Data

R.S. Bird

Programming Research Group, Oxford University, 8-11 Keble Road, Oxford OX1 3QD, UK

Summary. This paper describes a technique for transforming functional programs that repeatedly traverse a data structure into more efficient alternatives that do not. The transformation makes essential use of lazy evaluation and local recursion (such as provided by **letrec**, or its equivalent) to build a circular program that, on one pass over the structure, determines the effects of the individual traversals and then combines them.

1. Introduction

One of the advantages of programming in a functional language is that many ideas for improving efficiency can be formulated as simple transformations on the functions that constitute the program. The purpose of this note is to describe one such optimisation. The technique, which involves building a circular program in order to avoid the repeated traversal of a data structure, is familiar to a number of functional programmers, including Hughes [6] and Wadler (who first formulated it as a transformation in [9]), but it deserves to be more widely known. We shall illustrate it in a simple setting in order to make the nature of the underlying transformation apparent. It turns out that, in order to describe the transformation, one requires certain definition mechanisms to be available in the given language. We regard this aspect of the work as evidence of the importance of such mechanisms in functional programming.

The technique will be introduced through an example. Consider the problem of changing a given tree, such as the one depicted in Fig. 1 into a second tree identical in shape to the first, but with all the tip values replaced by the minimum tip value. For example, the tree of Fig. 1 is changed into the tree of Fig. 2.

In a straightforward solution to this problem the tree is traversed twice: once to find the minimum value and once to carry out the tip replacements. In

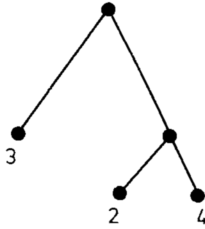

Fig. 1

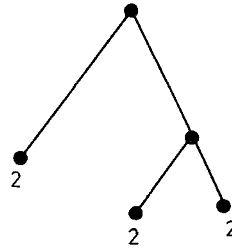

Fig. 2

Pascal, for instance, the algorithm might be implemented by the following definitions:

```
type tree = ^node;
   node = record
            case tag: (tip, fork) of
            tip: (val: integer);
            fork: (left, right: tree)
            end;

function tmin (t: tree): integer;
begin
   with t^ do
   case tag of
   tip:   tmin := val;
   fork:  tmin := min (t min (left), t min (right))
   end
end;

procedure replace (t: tree; m: integer);
begin
   with t^ do
   case tag of
   tip:   val := m;
   fork: begin replace (left, m); replace (right, m) end
   end
end
```

The required tree replacement is now achieved by the call replace $(t, \text{tmin}(t))$.

The programmer with a keen eye for possible optimisation will realise that it is not essential for the tree to be traversed twice in order to solve the problem. An alternative solution can, on a single pass through the tree, compute the minimum value and, at the same time, discover and store the addresses of the tip nodes in a linear list. Having calculated the minimum, the algorithm then inserts this value as the new tip value in each of the nodes stored in the list. If the tree contains N nodes, of which n are tip nodes, then the revised

algorithm does an extra n units of work during the traversal (to store the addresses of the tips) and a further n units to process the list. On the other hand, it traverses the tree only once. Assuming it takes N units to do this, the former algorithm takes $2N$ units, while the revised version takes $N+2n$ units. For the kind of trees we have been considering. $N=2n+1$ so the sensible programmer will reject the optimisation as not being worthwhile. However, there is nothing essential about binary trees as far as this problem is concerned (they were chosen purely for notational simplicity), and if one supposes more general trees in which the number of internal nodes can be arbitrarily larger than the number of tips (essentially, trees with unary as well as binary forks), then the revised version of the algorithm can generate a substantial saving.

2. A Functional Algorithm

Consider now how a solution to the same problem might be expressed in a functional language such as KRC [8] or HOPE [2]. (In what follows we shall employ a KRC-like notation with HOPE-like data types: given the Pascal definitions above, we trust these will be self explanatory.) The straightforward solution is easily programmed with the aid of the following data type:

data tree = tip int | fork tree tree

The required functions are:

transform t = replace t (tmin t)
replace (tip n) m = tip m
replace (fork L R) m = fork (replace Lm) (replace Rm)

tmin (tip n) = n
tmin (fork L R) = min (tmin L) (tmin R)

The above statement of the solution seems cleaner and simpler than its Pascal counterpart, but as there is no concept of tree addresses as individual entities in a functional framework, it is not obvious how, if at all, the corresponding single-pass algorithm might be formulated. However, given certain assumptions about the semantics and definition mechanisms available in the language, a single-pass algorithm can be described and it is the object of this note to show how.

As a first step it seems natural, given the common recursive form of replace and tmin, to attempt to combine the two functions into one by defining

repmin tm = [replace tm, tmin t].

From this definition we can synthesize a new one that constructs the two components in one traversal of t. The derivation is a standard application of the unfold-fold method [3] and the two cases are:

(i) repmin (tip n) m = [replace (tip n) m, tmin (tip n)]
= [tip m, n]

by instantiating (tip n) for t and subsequently unfolding the definitions of replace and tmin;

(ii) repmin (fork L R) m
$= [\text{replace}(\text{fork } L\, R)\, m,\ \text{tmin}(\text{fork } L\, R)]$
$= [\text{fork}(\text{replace } L\, m)(\text{replace } R m),\ \text{min}(\text{tmin } L)(\text{tmin } R)]$
$= [\text{fork } t1\, t2,\ \text{min } m1\, m2]$
where $[t1, m1] = \text{repmin } L\, m$
and $[t2, m2] = \text{repmin } R\, m$.

Here, instantiation and unfolding is followed by a **where** abstraction and a fold step in which the original definition of repmin is used a second time to obtain a new recursive definition. Alternatively, we can eliminate the **where** clause in favour of an explicit function h, say, and write

$$\text{repmin}(\text{fork } L\, R)\, m = h(\text{repmin } L\, m)(\text{repmin } R\, m)$$
$$h[t1, m1][t2, m2] = [\text{fork } t1\, t2, \text{min } m1\, m2]$$

The new definition of repmin certainly achieves the objective of evaluating replace and tmin with a single traversal, but the idea does not appear to help since the two components of the result seem to have been fatally decoupled from each other. What we really want is the second argument m of repmin to be the second component of its value. Suppose we were allowed to define

$$p = \text{repmin } t\, (\text{snd } p)$$

where snd $[a, b] = b$. Then the solution to our problem can be expressed as

$$\text{transform } t = \text{fst } p,$$

where fst $[a, b] = a$. Is such a move possible? Under what circumstances can we expect the definition

$$\text{transform } t = \text{fst } p$$
$$\textbf{where } p = \text{repmin } t\, (\text{snd } p)$$

to be well-formed and correct?

The answer is that, firstly, the functional language has to permit the recursive definition of local objects through the medium of **where** clauses (or their equivalent) and, secondly, that its semantics are based on a call by need (or lazy evaluation [5]) mechanism. The combination of lazy evaluation and local recursion enables circular program structures to be constructed at run-time, and this is just what is required for single-pass algorithm. Under a call by value regime, the value of the second argument of repmin is demanded before the value of repmin can be calculated: this leads to infinite regress and non-termination. With lazy evaluation, on the other hand, the structure

$$\text{fst}(\text{repmin } t\, (\text{snd}\ \cdot))$$

is built and repmin can be elaborated without evaluation of its second argument until it is actually required. To see this, we can work through a small example. Suppose $t = \text{fork}\,(\text{tip}\,2)(\text{tip}\,1)$. The evaluation of

$$\text{fst}\,(\downarrow \text{repmin}\; t\,(\text{snd}\uparrow))$$

(where, for convenience, the backward line has been replaced by appropriately pointing arrows) is as follows:

$$
\begin{aligned}
&\text{fst}\,(\downarrow \text{repmin}\,(\text{fork}\,(\text{tip}\,2)(\text{tip}\,1))\,(\text{snd}\uparrow))\\
&\quad = \text{fst}\,(\downarrow h\,(\text{repmin}\,(\text{tip}\,2)\,(\text{snd}\uparrow))\,(\text{repmin}\,(\text{tip}\,1)\,(\text{snd}\uparrow)))\\
&\quad = \text{fst}\,(\downarrow h\,[\text{tip}\,(\text{snd}\uparrow), 2]\,[\text{tip}\,(\text{snd}\uparrow), 1])\\
&\quad = \text{fst}\,(\downarrow [\text{fork}\,(\text{tip}\,(\text{snd}\uparrow))\,(\text{tip}\,(\text{snd}\uparrow)), \text{min}\;1\;2])\\
&\quad = \text{fork}\,(\text{tip}\,(\text{snd}\uparrow))\,(\text{tip}\,(\text{snd}\uparrow)),
\end{aligned}
$$

(where $\uparrow$ still points to $[\text{fork}\,(\text{tip}\,(\text{snd}\uparrow))(\text{tip}\,(\text{snd}\uparrow)), \text{min}\;1\;2]$)

$$
\begin{aligned}
&\quad = \text{fork}\,(\text{tip}\,(\text{min}\;1\;2))\,(\text{tip}\,(\text{min}\;1\;2))\\
&\quad = \text{fork}\,(\text{tip}\,1)\,(\text{tip}\,1).
\end{aligned}
$$

At each step just that portion of the expression required to continue the derivation is expanded. For instance, the definition of h requires that its arguments be pairs of values: consequently, the derivation is continued to establish this fact (though the actual values are not required at this stage) before the rule for h can itself be invoked. Note that a call by need mechanism ensures that min 1 2 is evaluated once only - when the first tip value is demanded - and thereafter its value is passed to the remaining tips.

It should now be clear why lazy evaluation is an essential ingredient of the transformation, but the second requirement, the ability to define local recursions, may not seem so important. One can, after all, define a version of transform in which the **where** statement is replaced by an explicit function:

$$
\begin{aligned}
&\text{transform}\; t = \text{fst}\,(p\,t)\\
&\qquad\qquad p\,t = \text{repmin}\; t\,(\text{snd}\,(p\,t)).
\end{aligned}
$$

This is a perfectly correct alternative, but since no circular structure is created, merely an extra - albeit curious - recursive function p, the main advantage of the earlier version is lost: the tree is again traversed twice. To see this, consider again the example $t = \text{fork}\,(\text{tip}\,2)(\text{tip}\,1)$; we have

$$
\begin{aligned}
&\text{transform}\; t\\
&\quad = \text{fst}\,(\text{repmin}\; t\,(\text{snd}\,(p\,t)))\\
&\quad = \text{fst}\,(h\,(\text{repmin}\,(\text{tip}\,2)\,(\text{snd}\,(p\,t)))\,(\text{repmin}\,(\text{tip}\,1)\,(\text{snd}\,(p\,t))))\\
&\quad = \text{fst}\,(h\,[\text{tip}\,(\text{snd}\,(p\,t)), 2]\,[\text{tip}\,(\text{snd}\,(p\,t)), 1])\\
&\quad = \text{fst}\,[\text{fork}\,(\text{tip}\,(\text{snd}\,(p\,t)))\,(\text{tip}\,(\text{snd}\,(p\,t))), \text{min}\;2\;1]\\
&\quad = \text{fork}\,(\text{tip}\,(\text{snd}\,(p\,t)))\,(\text{tip}\,(\text{snd}\,(p\,t)))
\end{aligned}
$$

At this point, the argument $(\text{snd}\,(p\,t))$ is demanded and the value $p\,t$ is reduced once more: essentially, this means a second traversal of the tree.

It is worth pointing out that local recursions are not permitted in Turner's KRC [8], even though this language has call by need semantics. One cannot therefore exploit the full power of lazy evaluation in KRC, or in any other language without such a facility.

3. Termination

The above program terminates because the second argument of repmin is demanded only when the first argument - the final tree - is required to be output. It is easy to define circular structures that do not terminate when executed, simply because such arguments are demanded too early. To illustrate this point, consider the problem of determining whether a given list of integers is palindromic, i.e. equal to its reverse. We can define

$$\text{palindrome } x = \text{eqlist } x\ (\text{reverse } x)$$

$$\text{eqlist } [\ ]\ [\ ] = \text{true}$$
$$\text{eqlist } (a:x)\ (b:y) = (a=b) \textbf{ and } \text{eqlist } x\, y$$

$$\text{reverse } x = \text{reverse}'\ x\ [\ ]$$
$$\text{reverse}'\ [\ ]\ z = z$$
$$\text{reverse}'\ (a:x)\, z = \text{reverse}'\ x\, (a:z)$$

using an efficient definition of reverse (see e.g. [5]) and a predicate eqlist for testing the equality of equal-length lists. Although it is unlikely to gain much in this case, we can at least try the effect of using the transformation on the definition of palindrome. First, we introduce

$$\text{eqrev}\, x\, y\, z = [\text{eqlist}\, x\, y,\ \text{reverse}'\, x\, z]$$

and then redefine

$$\text{palindrome } x = \text{fst}\, p \quad \textbf{where } p = \text{eqrev}\, x\, (\text{snd}\, p)\ [\ ]$$

The last step is to synthesize a new definition of eqrev, eqrev′ say. The result is

(i) $\text{eqrev}'\ [\ ]\ [\ ]\, z = [\text{eqlist}\ [\ ]\ [\ ],\ \text{reverse}'\ [\ ]\ z]$
$\qquad = [\text{true}, z]$

(ii) $\text{eqrev}\, (a:x)\, (b:y)\, z$
$\qquad = [\text{eqlist}\, (a:x)\, (b:y),\ \text{reverse}'\, (a:x)\, z]$
$\qquad = [(a=b) \textbf{ and } \text{eqlist}\ x\, y,\ \text{reverse}'\ x\, (a:z)]$
$\qquad = [(a=b) \textbf{ and } t, r]$
$\qquad\quad \textbf{where } [t, r] = \text{eqrev}\ x\, y\, (a:z).$

Unfortunately this definition has an infinite loop: eqrev′ demands information about its second argument too early. In order for rule (ii) to be used, the second argument has to be evaluated to the point where it is established that it is not the empty list. Thus in the computation of, say, palindrome [1], reduction of

$$\text{fst}\, (\downarrow \text{eqrev}'\ [1]\ (\text{snd}\uparrow)\ [\ ])$$

requires reduction of eqrev′ [1] (snd↑) [ ] and hence partial evaluation of (snd↑). In turn, this requires evaluation of eqrev [1] (snd↑) [ ]. The computation therefore gets stuck in an infinite loop.

What has gone wrong? The answer is subtle: the culprit is not the transformation of palindrome but the fold-unfold synthesis of the new definition of eqrev. The original eqrev and the new eqrev′ are *not* equivalent: one only has eqrev′ ⊆ eqrev where ⊆ is the approximation ordering of fixed point theory [1]. If the undefined value is denoted by ω, then we have

$$\text{eqrev } x\,\omega\,z \neq \omega$$
$$\text{eqrev}'\, x\,\omega\,z = \omega$$

for any defined values x and z. The reason is that with a call by need semantics a pair of values, the first of which is ω, is not the same as the undefined value ω. Moreover, the problem with the synthesis is not with folding, a transformation known to preserve only partial correctness [7], but with the choice of a list of instantiated clauses as replacement for the program. Under a call by need semantics even the functional program

$$\text{three } x = 3$$

cannot always be replaced by the clauses

$$\text{three } 0 = 3$$
$$\text{three } (x+1) = 0,$$

since the latter, which is the same as

$$\text{three } x = \textbf{if } x = 0 \textbf{ then } 3 \textbf{ else } 3,$$

is undefined for an undefined argument. When building circular programs, one has to be careful to avoid demanding information about an argument, either through pattern matching on the left hand side or an explicit conditional on the right, when such information can be delayed or avoided altogether.

The way to establish that a cyclic program is well-behaved is to look at the partial approximations of the program as defined by fixed point theory. For instance, suppose we have a definition of the form

$$\text{fun } x = \text{fst } p$$
$$\quad\quad \textbf{where } p = \text{pair } x\,(\text{snd } p)$$

where pair is some function which returns a pair of values, and fst and snd are as defined above. Fixed point theory tells us that

$$\text{fun } x = \text{fst}\,\Big(\bigcup_{n \geq 0} p_n\Big)$$

where

$$p_0 = [\omega, \omega]$$
$$p_{n+1} = \text{pair } x\,(\text{snd } p_n).$$

In particular,

$$p_1 = \text{pair } x\,(\text{snd } p_0)$$
$$= \text{pair } x\,\omega$$
$$p_2 = \text{pair } x\,(\text{snd } p_1).$$

From this it is easy to prove that $\text{fun}\, x \neq \omega$ for all $x \neq \omega$ provided that the following conditions are satisfied:

(C1) $\text{snd}\,(\text{pair}\, x\, \omega) \neq \omega$ for all $x \neq \omega$

(C2) $\text{fst}\,(\text{pair}\, x\, y) \neq \omega$ for all $x, y \neq \omega$

By (C1) we know $\text{snd}\, p_1 = \text{snd}\,(\text{pair}\, x\omega) \neq \omega$ provided $x \neq \omega$, and by (C2) that $\text{fst}\, p_2 \neq \omega$. Hence, since $\text{fst}\, p_2 \sqsubseteq \text{fun}\, x$, one can conclude $\text{fun}\, x \neq \omega$. A further example of this kind of reasoning is given in the next section.

Returning to the palindrome problem, the way out of the difficulty is to rewrite eqrev so that its second argument is "passive" in the associated reduction rules. This means that the definition must be recast in the form

$$\begin{aligned}
&\text{eqrev}''\ [\ ]\, y\, z = [\text{true}, z]\\
&\text{eqrev}''\ (a:x)\, y\, z = [(a = \text{hd}\, y)\ \textbf{and}\ t, r]\\
&\qquad\qquad\qquad \textbf{where}\ [t, r] = \text{eqrev}''\ x\, (\text{tl}\, y)\, (a:z).
\end{aligned}$$

Here the functions hd and tl are defined by the rules $\text{hd}(b:y) = b$ and $\text{tl}(b:y) = y$. Under the assumption that defined values of y are lists of the same length as x, one can show that

$$\text{eqrev}''\ x\, y\, z = [\text{eqlist}\ x\, y,\ \text{reverse}'\ x\, z]$$

for all values of x, y and z. Hence with the revised definition of eqrev the difficulty is removed and termination guaranteed.

4. A Further Example

Since programming with circular structures requires a little practice, we consider a third example closely related to the first. In this problem we are again required to transform a binary tree into one of the same shape, but this time the tip values have to be the tip values of the original tree arranged in increasing order. The direct solution can be formulated as follows:

$$\begin{aligned}
&\text{transform}\ t = \text{replace}\ t\ (\text{sort}\,(\text{tips}\ t))\\
&\text{replace}\,(\text{tip}\, n)\ [m] = \text{tip}\, m\\
&\text{replace}\,(\text{fork}\ L\ R)\, x = \text{fork}\,(\text{replace}\ L\,(\text{take}\,(\text{size}\ L)\, x))\\
&\qquad\qquad\qquad\qquad\qquad (\text{replace}\ R\,(\text{drop}\,(\text{size}\ L)\, x))\\
&\text{tips}\,(\text{tip}\, n) = [n]\\
&\text{tips}\,(\text{fork}\ L\ R) = \text{tips}\ L +\!\!+\ \text{tips}\ R\\
&\text{size}\,(\text{tip}\, n) = 1\\
&\text{size}\,(\text{fork}\ L R) = \text{size}\ L + \text{size}\ R
\end{aligned}$$

In this program the tree is traversed a first time in order to discover and sort the list of tip values (infix operator ++ concatenates two lists, and sort is some suitably chosen sorting function whose definition we omit). The tree is then traversed a second time with function replace. This function selects

appropriate chunks of the sorted sequence x – take kx takes the first k values and drop kx takes all but the first k values – in order to pass them on to the left and right subtrees. At each step, the number of values selected depends on the size of the left subtree, so implicit in the algorithm is a *third* traversal which determine sizes. Since the size of the left subtree is recalculated for every internal node, the algorithm takes cn^2 steps in the worst case, n being the number of tips. Even if multiple recomputations of size were avoided – say through use of a memo function that stored sizes at internal nodes – the algorithm still possesses a quadratic worst case complexity. There are two separate reasons for this: firstly, the computation of tips is inefficient since repeated use of concatenation ++ gives quadratic behaviour; secondly, take and drop are inefficient. Thus, even if one uses a guaranteed $O(n \log n)$ sorting algorithm, the running time is dominated by purely housekeeping operations. Nevertheless, the foregoing version is the natural way to specify the problem since it describes the recursive case in terms of renumbering the left and right subtrees independently.

Each of the mentioned sources of inefficiency can be removed by transformational programming. Let us deal first with concatenation. We define

$$\text{ntips}\, t\, x = \text{tips}\, t ++ x.$$

Since $\text{tips}\, t = \text{ntips}\, t\, [\ ]$, use of ++ can be avoided by synthesing an efficient version of ntips. Now

$$\begin{aligned}
\text{(i)}\quad \text{ntips}(\text{tip}\, n)\, x &= \text{tips}(\text{tip}\, n) ++ x \\
&= [n] ++ x \\
&= n : x
\end{aligned}$$

$$\begin{aligned}
\text{(ii)}\quad \text{ntips}(\text{fork}\, L\, R)\, x &= \text{tips}(\text{fork}\, L\, R) ++ x \\
&= (\text{tips}\, L ++ \text{tips}\, R) ++ x \\
&= \text{tips}\, L ++ (\text{tips}\, R ++ x) \\
&= \text{tips}\, L ++ n\, \text{tips}\, R x \\
&= \text{ntips}\, L (\text{ntips}\, R x)
\end{aligned}$$

The new definition of ntips is linear in the number of tips.

We are now in a position to deal with take and drop. The solution is to consider an algorithm that combines the multiple traversals into one. With a little foresight, we define

$$\begin{aligned}
\text{repnd}\, t\, x\, y = [&\text{replace}\, t\ (\text{take}\,(\text{size}\, t)\, x), \\
&\text{drop}\,(\text{size}\, t)\, x,\ \text{ntips}\, t\, y].
\end{aligned}$$

Provided we can derive an efficient alternative definition for repnd, the solution to the problem can be put in the form

$$\begin{aligned}
&\text{transform}\, t = \text{fst}\, p \\
&\qquad\qquad \textbf{where}\ p = \text{repnd}\, t\ (\text{sort}\,(\text{thd}\, p))\ [\ ].
\end{aligned}$$

Here, $\text{thd}\, [a, b, c] = c$. This version of the algorithm builds a circular structure in the way we have seen before.

It remains to tackle the synthesis of repnd:

(i) repnd (tip n) x y
  $=$ [replace (tip n) (take (size (tip n)) x),
   drop (size (tip n)) x, ntips (tip n) y]
  $=$ [replace (tip n) (take 1 x), drop 1 x, $n : y$]
  $=$ [tip (hd x), tl x, n: y]

(ii) repnd (fork L R) x y
  $=$ [replace (fork L R) (take (size (fork L R)) x),
   drop (size(fork L R)) x, ntips(fork L R) y]
  $=$ [fork (replace L(take (size L) (take (size L + size R) x))
    (replace R (drop (size L) (take (size L + size R) x))),
   drop (size L + size R) x, ntips L(ntips R y)].

To continue the derivation, we need the following facts about take and drop, all of which can be proved by straightforward means:

take n (take $(n+m)$ x) = take n x
drop n (take $(n+m)$ x) = take m (drop n x)
drop $(n+m)$ x = drop m (drop n x).

Using these results we can continue the derivation:

repnd (fork L R) x y
  $=$ [fork (replace L (take (size L) x))
    (replace R (take (size R) (drop (size L) x))),
   drop (size R) (drop (size L) x),
   ntips L(ntips R y)]
  $=$ [fork $t1$ $t2$, $x2$, $y1$]
   where [$t1$, $x1$, $y1$] = repnd L x $y2$
    and [$t2$, $x2$, $y2$] = repnd R $x1$ y.

Observe that with this last fold step we have built a second cyclic structure into the algorithm: $y2$ depends on $x1$ which in turn depends on $y2$. One can appreciate its similarity to previous examples of cyclic structures by recasting it in the form

repnd (fork L R) x y = combine p q
        where p = repnd L x (thd q)
         and q = repnd R (snd p) y
combine [$t1$, $x1$, $y1$] [$t2$, $x2$, $y2$] = [fork $t1$ $t2$, $x2$, $y1$].

It is easy to check that the program is well-behaved since

snd (repnd t ω ω) $\neq \omega$
thd (repnd t ω y) $\neq \omega$

for all t, x, and y ($\neq \omega$), and so both p and q are well-defined.

The final algorithm is the following:

```
transform t = fst p
                where p = repnd t (sort (thd p)) [ ]
repnd (tip n) x y = [tip (hd x), tl x, n : y]
repnd (fork L R) x y = combine p q
                         where p = repnd L x (thd q)
                           and q = repnd R (snd p) y
combine [t1, x1, y1] [t2, x2, y2] = [fork t1 t2, x2, y1]
```

Apart from the time spent sorting, the rest of the algorithm is linear in the number of tips. Though incomprehensible taken by itself, the final version has been derived systematically from the original specification using cyclic programming in conjunction with other transformations.

5. Conclusions

Although the examples deal with trees and linear lists, the technique obviously applies to any other structure which has to be multiply traversed in order to deliver a result. Moreover, once understood, the underlying transformation is simple to implement, though one does have to check conditions such as those given in Sect. 3 to ensure termination. The Pascal programmer confronted with the same idea for optimisation has to undertake a major revision of his or her program to achieve the same end. Given the necessary features - lazy evaluation and local recursion - in their language, functional programmers have a much easier task.

Finally, one should point out that there are alternative transformations for eliminating multiple traversals that do not involve the construction of circular programs. For instance, Feather's composition transformation [4] can be used to avoid the construction of intermediate lists in a program (in fact. given appropriate restrictions on the form of function definitions allowed. Wadler [10] has shown in his listless transformer that such optimisations can be done quite automatically). There is an interesting connection between Feather's transformation and ours. Briefly, Feather's composition deals with expressions of the form $f(gx)$, where both of f and g are list-to-list functions. In deference to the B combinator of combinatory logic, one may call this B-composition transformation. Now, the transformation described in the body of the paper deals essentially with expressions of the form $fx(gx)$. Such expressions are related to the S combinator of combinatory logic, so one could call the technique S-composition transformation.

Acknowledgements. The author first appreciated the power of circular program structures in the above context from John Hughes [6], who uses the idea extensively in his super-combinator compiler. Phil Wadler outlined to the author how the idea could be formulated as a transformation. Geraint Jones also knew about the technique. Without their insights this note could not have been written. Thanks are also due to Tony Hoare for detailed comments on a first draft of the paper. The research was supported by Science and Engineering Research Council Grant GR/c/78704.

References

1. Bird, R.S.: Programs and Machines – An Introduction to the Theory of Computation. London: John Wiley 1976
2. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**, 44–67 (1977)
3. Burstall, R.M., Macqueen, D.B., Sannella, D.T.: HOPE: an experimental applicative language. Int Res Report. Dept Computer Science, University of Edinburgh 1980
4. Feather, M.: A system for assisting program transformation. ACM Trans Progr. Lang. Syst. **4**, 1–20 (1982)
5. Henderson, P.: Functional Programming: Application and Implementation. Englewood Cliffs: Prentice-Hall 1980
6. Hughes, R.J.M.: The Design and Implementation of Programming Languages. D. Phil. Thesis. Oxford University 1983
7. Kott, L.: About a transformation system: a theoretical study. Proc. Third Symp. Progr. Paris, 1971
8. Turner, D.: Recursion equations as a programming language. In: Functional Programming and its Applications (Darlington, J., Henderson, P., Turner, D. (eds.). Cambridge: University Press, 1982
9. Wadler, P.: (personal communication)
10. Wadler, P.: Listlessness is better than laziness. Ph. D. Thesis, Carnegie-Mellon University, 1984

Received January 28, 1984 / May 15, 1984