.so /staff/lloyd/specsyms

Circular Programs and Self-Referential Structures.

Lloyd Allison
Department of Computer Science
Monash University
Clayton
Victoria
Australia 3168.

uucp: lloyd@bruce.cs.monash.edu.au

written: 17 November 1987
revised: 17 May and 18 August 1988
Appeared in Software Practice and Experience **19**(2) 99-109 Feb 1989.

Abstract: A *circular program* creates a data structure whose computation depends upon itself or refers to itself. The technique is used to implement the classic data structures circular and doubly-linked lists, threaded trees and queues, in a functional programming language. These structures are normally thought to require updatable variables found in imperative languages. For example, a functional program to perform the breadth-first traversal of a tree is given. Some of the examples result in circular data structures when evaluated. Some examples are particularly space-efficient by avoiding the creation of intermediate temporary structures which would otherwise later become garbage. Lastly, the technique can be applied in an imperative language to give an elegant program.

Keywords: circular program, functional programming, lazy evaluation, call by need, recursion.

1. Introduction.

Bird[1] describes the use of *circular programs* and applies the technique to transform a program making multiple passes over a data structure into another program making only 1 pass. He attributes knowledge of the technique to Hughes and to Wadler. Bird's examples are specialized and given in a program-transformation setting. The purpose of this paper is firstly to show that circular programs are more widely applicable as a programming technique. Under suitable circumstances, circular programs can be used to program circular and doubly linked lists, threaded trees and queues[2]. These classic data structures are normally thought to require updatable variables. Secondly, a circular program can be more space-efficient than a conventional program by avoiding the creation of temporary data structure which need to be garbage collected later. Examples include removing duplicates from a list and variations on the sieve of Eratosthenese. Lastly, a circular program can often be translated into an imperative language, if that is necessary, giving an elegant and efficient program.

A circular program involves a recursive or self-referential expression for a data structure.

```
let rec ds = f(ds)
in      ds
```

Note that **ds** is a data structure and not a function. To write a circular program in a functional language requires a lazy language[3,4]. The evaluation of the data-structure refers to the data-structure itself; this plainly rules out a strict language. The evaluation must not use any part or attribute of the structure before it has been, or can be, computed as this would call for prescience. Many applications involve specifying a structure before its contents are known and this is a *forte* of lazy languages.

There appears to be no universally accepted and precise definition of lazy evaluation. The bare minimum for the programs in this paper to run correctly is a call by name mechanism for binding the right-hand sides of local definitions and for passing parameters. This would however be enormously inefficient and result in duplicate evaluation of many expressions and void the point of the exercise. The minimum requirement to gain the full advantages of circular programs is a graph reduction mechanism using call by need for the right hand side of local definitions, for parameter passing and in particular for the parameters of type constructors such as *cons* or ‘.’ for lists. Under call by need, an object is bound to a recipe[5] which will produce the desired value if and when it is needed. If the value is needed then the recipe is forced or evaluated and the recipe is also overwritten so that the value is immediately available thereafter. A value may contain sub-recipes and these are not forced until necessary. If a value contains a recursive reference to itself, this is implemented as a pointer to the top of the value - hence graph reduction. All of this is invisible to the programmer. Lazy evaluation is taken to mean such a system throughout this paper.

In the scheme above, **ds** is bound initially to a recipe for f(**ds**). As parts of **ds** are used in other computations the recipe is progressively evaluated. When f finally makes reference to **ds** all is well if the required parts and attributes have been computed. If f(**ds**) simply incorporates **ds** itself in its value, **ds** must already have been forced to avoid non-termination. In this case a pointer to the "top" of the data structure is incorporated and a circular data structure results in the underlying implementation of the language. There is then a path from the top of the data structure only through type constructors, or unevaluated functors in logic-programming terms, back to the top of the structure. This can be very efficient as some infinite values can be represented in finite space! In other examples f uses **ds** in some other way and no circular data structure is created.

The programs given here do not require the even more parsimonious evaluation rule called *full laziness* (although it does no harm) which guarantees that no expression at all is evaluated twice. As an example[6] consider:

```
let    f x y = √x+y
in let g = f 4
in     g 1 + g 2
```

the expression √x = √4 = 2 is evaluated twice. In a fully lazy system an optimization equivalent to the following is made:

```
let    f x = let sqrtx = √x in λ y. sqrtx+y
in let g = f 4                         --  = λ y.2+y
in     g 1 + g 2
```

and √x is only evaluated once.

A final requirement for writing any functional program is that the data structure, **ds**, be of the single assignment type. That is, values are not changed once they are known. Many uses of data structures do have this property. Sometimes the fact is disguised in an imperative coding in that a value may be tagged, set to nil or otherwise marked until the proper value is known.

In the following sections various examples are given of circular programs in a functional language. The use of the technique in imperative languages is then discussed.

2. Functional Examples.

Various applications of circular programs follow. A simple functional language is used in which local definitions are included by ‘let in’ or by ‘where’. Recursive definitions are qualified by ‘rec’. Lists are frequently used and the empty list is denoted by nil or by ‘[ ]’ and the list constructor (cons) by ‘.’. The convention of writing a self-referential structure in **`bold`** characters is adopted simply to catch the eye.

2.1. Circular Lists.

The simplest circular program of all creates the apparently infinite list [1, 1, ...].

```
let rec ones = 1.ones
in      ones
```

This is easily evaluated in a lazy language and the implementation creates a circular list containing one cell that points to itself. Initially **ones** is bound to a recipe for 1.**ones**. Assuming that this is evaluated, a list cell is created which contains recipes for 1 and for **ones**. If the latter is evaluated, it is also overwritten - with the value of **ones** which is a pointer to the list cell. This creates the following graph:

```
ones:------------> . <-----
                  / |     ^
                 /  |     |
                /   --->--|
               1
```

A program scheme for creating many, although by no means all, circular lists generalizes **ones**:

```
let circ x = c
   where rec
       c = build x      -- c self-referential
   and build y =
          f(y) . (if p y then c else build (g y) )
```

p is some predicate and f and g are arbitrary functions. Note that **c** is a data structure and build is a function. The result is a list, **c** which is equal to the result of appending [f(x), f(g(x)), f(g^2(x)), ..., f(g^n(x))] and **c** where $n \geq 0$ and p(g^n(x)) is true. The final **c** is implemented as a pointer back to the start of the list. It is possible to eliminate **c** by substituting it in build to get the following program:

```
let uncirc x = build x
   where rec
       build y =
          f(y) . (if p y then build x else build (g y) )
```

This produces the correct value but it is no longer implemented as a circular structure; the data structure is unfolded: [f(x), ..., f(g^n(x)), f(x), ..., f(g^n(x)), f(x), ...]. This is an equivalent, although much more wasteful, way of representing the value. Note that a system using string reduction rather than graph reduction is liable to produce this data structure for program circ. Hughes[7] describes a mechanism called *lazy memo functions* which would build the circular data structure for the program uncirc by remembering and reusing function results for past inputs - such as build x.

2.2. Doubly Linked Lists.

A doubly linked list can be defined in a manner similar to a circular list. A doubly linked list is either nil or contains three things - a pointer to a predecessor, an element and a pointer to a successor node.

```
datatype dbl_list = nil |
                    dbl of dbl_list × elt_type × dbl_list

let double x = build [] x
   where rec
      build prev y =
         if p y then []
         else d
            where rec
               d = dbl(prev, f y, build d (g y))
```

If p y is immediately true build returns nil otherwise it creates a node **d**. The node points to its predecessor prev. It also points to the successor created by a recursive call of build. The predecessor of this next node is **d**. **D** is local to build because the predecessor of a node was created by the preceding call to build. On the other hand, **c** is global to the routine for circular lists because the start point remains the same through the recursive calls.

In an imperative language a doubly-linked list would be created one node at a time. The successor pointer of a node would be set to nil or left undefined until the successor was created. The pointer would then be overwritten. In the circular program above, a node is created although part of it (the successor pointer) is unevaluated. The node can still be passed as a parameter so that a pointer to it can be included as the predecessor of the succeeding node.

Given a lazy language, any amount of scanning the doubly linked list backwards and forwards causes no extra copies of the list to be created. **D** is directly recursive and can only be removed first by using a fixed-point operator and then by substituting in build but then, as in the previous example, no circular data structure is created.

2.3. Threaded Trees.

A node of a binary tree contains an element and pointers to the left and right subtrees. Most of the tree consists of leaves and most of the pointers are empty. A *threaded tree*[8] uses those right pointers that would be empty to point to successor nodes in infix order. The threads allow the elements in the tree to be processed sequentially by an iterative or linear-recursive routine.

```
datatype threaded_tree = empty |
   thrd of threaded_tree |
   fork of threaded_tree × elt_type × threaded_tree
```

Such a tree may be created in an imperative language by overwriting empty pointers when the threads were known.

Provided that all the elements to be placed in the tree are given at one time, a circular program can be written to create a threaded tree. (In this example the tree is also a binary search tree without duplicates.)

```
let thread L = build true empty L
   where rec
      build isleft succ L =
         if null L then
            if isleft then empty else thrd(succ)
         else t
            where rec t =
               fork(build true t (filter (< hd L) L),
                    hd L,
                    build false succ (filter (> hd L) L))
```

The input elements are in the list L. Filter is a common function to select elements from a list according to a predicate or test:

```
let rec filter p l =
   if null l then []
   else let h=hd l
        and rest = filter(tl l)
        in  if p h then h.rest else rest
```

If L is not null, a node **t** is built. **T** contains a left subtree and either a right subtree or a thread. The successor thread for the left subtree is **t** itself. The successor thread for the right subtree is the successor of **t**. This example uses only right threads but left or predecessor threads are easily added. The requirement that all elements be given in a list L is to ensure that the tree does not need to be updated when new elements arrive.

2.4. Breadth-First Traversal.

The next example and following ones use expressions which are recursive or circular but whose values, the result of evaluation, are not and thus no circular data structures are created.

Prefix, infix and postfix traversals of a tree are easily programmed in a functional language but breadth-first traversal is harder. The usual imperative algorithm employs a queue and seems to need destructive assignment. An element is taken from the front of the queue for traversal and its children are added to the end of the queue. This appears to imply that either the queue must be updated or that new copies of a modified queue must be created at each step. A circular program can be written however in which elements are removed from the front of the queue as the end is still being computed:

```
datatype tree = empty | fork of tree × elt_type × tree

let bfirst t = r                      -- bfirst: tree→list
  where rec
      r = case t of
             empty => [] |
             fork(left,elt,right) => t.(bf r 1)
  where rec
      bf q n =                        -- bf: list→int→list
         if n=0 then []               -- q is used up
         else let root = hd q
              and rest = bf (tl q)
              in case root of
                     fork(empty,e,empty) => rest(n-1)   |
                     fork(left, e,empty) => left.(rest n) |
                     fork(empty,e,right) => right.(rest n) |
                     fork(left, e,right) => left.(right.(rest(n+1)))
```

Bfirst returns a list or queue of the nodes of the tree t in breadth-first order. If t is not empty, the first node is the root t itself; at this point the queue has 1 known element. Bf is the central function. It absorbs a queue q whose known part is of length n while computing the result queue; the two queues are in fact one. N indicates the shape of that part of the structure **r** that can be used safely. In an imperative language n would not be necessary and **r** might be temporarily terminated by nil but that cannot be done in a functional language. Bf places non-empty children in the result queue and adjusts its length accordingly; each call to bf uses one element from q and adds 0, 1 or 2 elements. Rest is a function to build the result literally for the rest of the input queue after the current node. Note that bfirst can traverse even infinite trees. It uses one list cell for each node that is traversed.

2.5. Unique.

The previous examples illustrated the use of circular programs to implement classical data structures in functional programs. The next example uses the technique to make a space-efficient program.

Consider the problem of writing a function *unique*. It is to accept a list as parameter and to return a list with the same members but with duplicate members removed and the order of first occurrence is to be maintained. The problem is close to finding the union of two sets represented by lists.

It is easy to write such a function if the constraint on order is dropped:

```
let rec uniqueL L =
   if null L then []
   else if member (hd L) (tl L) then
      uniqueL (tl L)
   else (hd L).(uniqueL (tl L))
```

uniqueL preserves the order of *last* occurrence so reverseuniqueLreverse would solve the original problem.

It would also create garbage in the shape of two intermediate lists.

Another solution to the problem that is not quite good enough is

```
let rec uniqueF L =
  if null L then []
  else (hd L).(uniqueF (filter (≠ hd L) L))
```

UniqueF certainly preserves the order of first occurrence but each call of filter creates a temporary list. UniqueF creates O(|L|) such lists and uses O(|L|^2) space.

An imperative programmer might arrive at the following informal description of a solution. Unique should create a list r. It should examine the input L, element by element. If the current element of L is in r it should not be added again. If it is not in r it should be added to r. There is no need to use an imperative language to implement this algorithm. A circular program can use the list r while creating it at the same time:

```
let unique L = r
   where rec
       r = u L 0
   and u L n =
          if null L then []
          else if member (hd L) r n then
            u (tl L) n
          else (hd L).(u (tl L) (n+1))
   and member e L n =
          if n=0 then false
          else if e=hd L then true
          else member e (tl L) (n-1)
```

R is the self-referential data structure that function u both creates and uses at the same time. Member is a variation on the conventional list membership function. While the result **r** is being built its end is unknown; it terminates in a recipe. Member cannot therefore use null(L) to detect the current end point of the search list. As in breadth-first traversal, an integer parameter n is added to keep track of the length of the known part of **r**; it stops member from forcing the recipe and causing an infinite loop. Note that the shape of a list is represented by a single integer but that the shape of a tree is more expensive to represent; a circular program that computes a tree where the computation depends on the shape of the part already evaluated is unlikely to be so efficient.

Note that uniqueF and unique will operate on infinite lists but that only unique creates no intermediate lists and runs in space linear in the amount of output.

2.6. Primes.

The final functional examples of circular programs are variations on the sieve of Eratosthenese. A typical non-circular coding, similar to one in Henderson's book[5] is

```
let knot p x = not(p x)
and mult m n = n mod m = 0
in let rec
    from n = n.(from (n+1))
and sieve L = (hd L).(sieve (filter (knot mult (hd L)) L))
in sieve (from 2)
```

This program creates many intermediate lists - from 2 and various sublists containing fewer and fewer composites. This is due to the many successive calls on filter each of which returns a list.

There are two main families of primes programs in imperative programming. A sieve program finds successive primes and for each prime removes all multiples of it from the set of numbers. A program of the other family maintains a set of primes. There is a loop over new candidates and each candidate is tested for primality against members of the set. It may then be added to the set. The filtering of each candidate becomes the inner operation and it can be coded as follows:

```
let rec
   multiple L n =
      if sqr(hd L)>n then false
      else if mult (hd L) n then true
      else multiple (tl L) n
in let rec
   primes = 2.(filter (knot (multiple primes)) (from 3))
in primes
```

In this circular program, the expression for **primes** is self-referential. It starts with 2 and a sublist of from 3 follows. All composite numbers are removed by one call to filter and so no intermediate lists are created. The predicate ‘multiple **primes**’ tests if a number is composite by examining only primes already calculated. When a new number is tested for primality, primes exceeding its square root are known and so it is not necessary to pass the number of known primes to multiple (compare this with breadth-first and unique).

The central expression ‘knot (multiple **primes**)’ is precisely the predicate isprime. This observation yields the alternative circular program:

```
let rec
    primes = 2.(filter isprime (from 3))
and isprime = knot (multiple primes)
in primes
```

primes and isprime form a mutually recursive data structure and function pair. Alternatively, substituting primes in isprime gives:

```
let rec
   isprime = knot(multiple (2.(filter isprime (from 3))))
```

Substituting and moving between these three programs brings no disadvantage except perhaps for the loss of the ability to refer to both **primes** and isprime, because no circular data structure is created in any of the programs. The behaviour of isprime is interesting because it runs faster the second time that it is called on a number of a given order of magnitude because the primes that it needs have already been calculated.

3. Imperative Languages.

On circular programs, Bird[1] states ‘the Pascal programmer confronted with the same idea for optimization has to undertake a major revision of his or her program to achieve the same end’. It is certainly true that Bird’s transformations, and many others, are very hard to apply systematically in an imperative language because of side-effects but a circular program can often be written quite easily in such a language. If using an imperative language is a condition of the job, a circular program might still be coded to give an elegant and efficient result.

A direct translation of unique, for example, into Pascal will *not* work because Pascal uses strict evaluation. But

```
function f(...):...;
begin ... f:=e; ... end
```

can be replaced by

```
procedure f(... var fresult:...);
begin ...; fresult:=e ... end
```

which is defined more often than the former. If this is applied to unique, the result is:

```
function unique(L:list):list;
  var r :list;
  function member(e:element;
                  L:list;
                  n:integer):boolean; ...;

  procedure u(L:list; n:integer; var uresult:list);
  begin
    if L=nil then uresult:=nil
    else if member(L^.hd, r, n) then
      u(L^.tl, n, uresult)
    else
    begin
      uresult:=cons(L^.hd, nil);      (* nil - not a recipe *)
      u(L^.tl, n+1, uresult^.tl)
    end
  end;
begin r:=nil; u(L, 0, r); unique:=r
end
```

Note that the list r is nil terminated so a more usual version of member can be adopted and parameter n of routines u and member can be dropped. Needless to say the Pascal version will not run on infinite lists. A similar transformation can be performed to give a Pascal version of the breadth-first traversal program and of the other programs.

The Pascal version of unique is simply doing what the implementation of a lazy language would do for a circular program. It marks the current end of the result r as nil for unevaluated and this is overwritten when its value is known.

One of Bird's examples is to transform a tree, into a tree of the same shape but replacing the leaf values by the minimum leaf value of the input tree. His circular program builds a leaf with a minimum but unevaluated value while incorporating this leaf in a tree of the correct shape. The leaf's value is calculated during the same traversal that copies the tree's shape. A Pascal version is almost as simple. It creates a node, traverses the input tree incorporating the node in a new tree and also searching for the minimum value. When all is done this value is stored in the leaf. Again this mimics the lazy implementation.

4. Conclusions.

A circular program uses a recursive expression for a data structure. In cases where the evaluation of the expression incorporates the data structure directly, the result is a circular data structure. Circular programs permit classic data structures such as circular and doubly-linked lists, threaded trees, and queues to be used in a functional programming language and brings some of the efficiency of imperative programming to functional programming. Provided that the structures are subject to the single assignment rule, reference variables and assignment (:=) are not needed. Many times a circular program is more space-efficient than its conventional counterpart by avoiding the creation of intermediate structures.

A lazy functional language is needed to write a circular program. Garbage collectors based purely on reference counts cannot collect circular structures but mark-scan collectors, copying collectors and hybrid schemes can[9]. Since the programs are space efficient, the collector should in any case be called infrequently.

In addition to the use of functional languages, a circular program can often be translated into an imperative language such as Pascal with only minor revision of ideas.

References.

1. R. S. Bird, ‘Using circular programs to eliminate multiple traversals of data’,
Acta Informatica, **21**(3), 239-250 (1984).

2. L. Allison, *Two functional programming techniques: continuations and circular programs*,
Monash University, Dept. Computer Science, TR 87/91, Jan 1987.

3. D. P. Friedman and D. S. Wise, *Cons should not evaluate its arguments*,
Automata, Languages and Programming,
Edinburgh University Press, 257-284, 1976.

4. P. Henderson, *A lazy evaluator*,
3rd ACM Symposium on Principles of Programming Languages, 95-103, 1976.

5. P. Henderson, *Functional Programming: Application and Implementation*,
Prentice Hall, 1980.

6. S. L. Peyton-Jones, *An introduction to fully-lazy supercombinators*,
Combinators and Functional Programming Languages,
G. Cousineau, P-L. Curien and B. Robinet eds.,
Springer Verlag, LNCS 242, 176-208, 1985.

7. J. Hughes, *Lazy memo-functions*,
Functional Programming Languages and Computer Architecture,
J-P. Jouannaud ed.,
Springer Verlag, LNCS 201, 129-146, 1985.

8. E. S. Page and B. Wilson, *Information Representation and Manipulation in a Computer*,
Cambridge University Press, Cambridge Computer Science Texts 2, 1973.

9. J. Cohen, ‘Garbage collection of linked data structures’,
Computing Surveys **13**(3), 341-367 (Sep 1981).