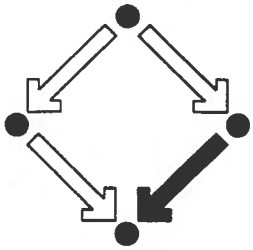
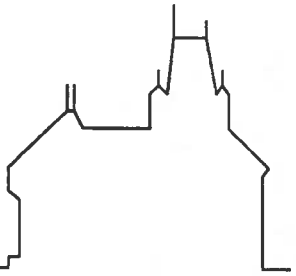


RISC-Linz
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



Circular Programs on Compound Data Structures

Hans Wolfgang LOIDL

(January 20, 1992)

RISC-Linz Report Series No. 92-06

Supported by: Austrian Forschungsförderungsfonds, project no. P-06931

Circular Programs on Compound Data Structures

Hans Wolfgang Loidl
e-mail: hwloidl@risc.uni-linz.ac.at
Parallel Computation Laboratory
RISC-Linz
Johannes Kepler University
4040 Linz, Austria, Europe

January 17, 1992

Abstract

A circular program creates a data structure whose computation depends upon (parts of) the structure itself. This paper provides a survey of what has been published about circular programs during the last years. So, in this paper it is shown that a non-strict semantics and local recursion are necessary properties of the underlying functional language. Furthermore, it will be shown that circular programs can have a better efficiency (e.g. by avoiding the production of intermediate temporary structures or by being able to traverse a compound data structure only once) than conventional programs. As the termination of a circular progam is by no means obvious, a strategy will be outlined how to prove if every element of a list that is defined by a circular program can be computed in finite time.

1 Introduction

Due to the increasing importance of functional programming languages, many programming techniques have been developed, which are special for this kind of languages. In this paper the technique of circular programs will be discussed. It is restricted to languages which have a non-strict semantics and which support local recursion. Informally speaking, a function is non-strict in an argument iff it may return a result even if the argument has not been evaluated yet. Correspondingly, a data structure is non-strict iff its constructor function is non-strict. This means that a data structure may be used although some parts of it have not been evaluated. If a function needs such a part the access to this part has to be delayed[1].

Among the papers dealing with circular programs especially the following have to be mentioned:

- The first one is [Bird, 1984], which concentrates on deriving a circular program, which traverses a tree only once, out of a conventional functional program, which traverses a tree several times, by using program transformation. It also discusses the problem of termination of circular programs.

[1] Based on this property, such non-strict data structures can be used in a parallel implementation of a non-strict functional programming language for the *synchronization of parallel processes.*

- The second paper is [Allison, 1989], which stresses the space efficiency of circular programs and shows how to use circular programs to create special compound data structures (like doubly-linked lists).
- The third paper is actually the part of a masters thesis namely of [Schreiner, 1990]. Chapter 2.4 'Dataflow Programming' of this thesis is devoted to the subject of non-strict functional programming languages. Within this chapter a very efficient prime number algorithm is presented. As the whole masters thesis deals with the parallel implementation of a non-strict functional programming language based on the dataflow model of computation, also the computation times for a circular program on various numbers of processors are presented there.
- The fourth paper is [Sijtsma, 1989], which deals with the question if each element of a list that is defined by a circular program can be accessed in finite time. This paper shows a way how to handle this question in a formal way.

Several examples and ideas are taken from these papers, as well as from [Wray and Fairbairn, 1989][Chapter 3], where two programming techniques for non-strict functional languages are discussed.

Additionally to the above papers, there are also some monographs that deal with the topic of circular programs. The most important of these are:

- One of the oldest references dealing with circular programs is [Henderson, 1980]. In Chapter 8.4 'Networks of Communicating Processes' the advantages of lazy evaluation are explained. The networks that are introduced there correspond to the dataflow diagrams in this paper. Furthermore, the examples of the list of Natural Numbers, the list of Fibonacci Numbers, the list of Prime Numbers (by using the 'Sieve of Eratosthenes') and the list of Hamming Numbers are presented in [Henderson, 1980].
- In [Field and Harrison, 1988][Chapter 4.3] circular programs are discussed under the title of 'process networks'. Chaper 4.4 in this book shows how to eliminate multiple traversals of a list by using a circular program (in this part the idea that is also presented in more details in [Bird, 1984] is discussed).
- In [Kelly, 1989][Chapter 4.3] cyclic process networks, which are a graphical representation of circular programs, are introduced in the frame of the so called pipeline parallelism. Therefore, this book stresses the parallelism that can be exploited in the execution of a circular program.
- Although [Wentworth, 1989] is actually the description of the lazy functional programming language RUFL, it also contains a dicussion of 'circular structures'. There in Chapter 9 the idea of a circular program is explained by using many examples.

Another main point of the paper is to show that non-strict functional languages, most of all circular programs, contain lots of inherent parallelism. This could therefore be exploited by a parallel implementation of such a language. Obviously, such an implementation would be an important contribution to the current attempts to improve the efficiency of the implementations of non-strict languages. The implementation of the G-Machine, which

is described in [Johnsson, 1987] and [Augustsson, 1987] and improvements made e.g. by [Peyton Jones, 1991] have shown that an efficient implementation of non-strict languages is possible. Since an implementation of a non-strict functional language is less efficient than that of a strict functional language, there could be doubts if it is worthwhile gaining the additional expressive power of such languages since much efficiency is lost. Now, in this paper it will be shown that this additional expressive power can be used to improve the efficiency of programs by making them circular. Furthermore, non-strict functional languages contain more inherent parallelism than strict functional languages. This might lead to an efficient implementation of such languages. Taking both points together circular programs can reach an extremely high efficiency.

2 Examples

Before we begin studying circular programs some remarks on the functional programming language that is used in this paper:

- The language that is used here is RUFL due to the language definition in [Wentworth, 1989]. It is a lazy functional programming language that has a strong resemblance to Miranda[2] and therefore also to the new HASKELL language. A short description of RUFL that mainly concentrates on the differences to languages like Miranda can be found in Appendix A.1.
- All keywords are written in **boldface**.
- All datatype constructors are written in SMALL CAPS style.
- All builtin functions are written in roman style.
- All user defined functions and objects are written in *italics* style.

2.1 The Infinite List of Natural Numbers

To explain the crucial points of circular programs let us start with a very simple example[3]:

$$\begin{array}{lll} \mathit{naturals} & \textbf{where} & \mathit{incList} = \mathrm{map}\ ((+)\ 1) \\ & & \mathit{naturals} = 1\ :\ (\mathit{incList}\ \mathit{naturals}) \end{array}$$

This expression can compute the infinite list of all natural numbers. Such infinte lists can only be produced in a non-strict functional language (in particular with a non-strict list type) since in such a language a function may return a result although some of its arguments are not evaluated yet. In a strict language this would yield an infinite recursion. However, here the list of all integers, namely *naturals* is being constructed. There are no intermediate lists produced. From Figure 1 we see that all lists that are produced by (*map* ((+) 1)) ... are themselves sublists of *naturals*. If a new integer has been detected,

[2] Miranda is a trademark of Research Software Ltd.

[3] The function *map* is among others defined in Appendix A.2.

$$
\begin{aligned}
\mathit{naturals} &= 1 : (\mathit{incList}\ \mathit{naturals}) \\
&= 1 : (\mathrm{map}\ ((+)\ 1)\ \mathit{naturals}) \\
&= 1 : 2 : (\mathrm{map}\ ((+)\ 1)\ (\mathit{incList}\ \mathit{naturals})) \\
&= 1 : 2 : (\mathrm{map}\ ((+)\ 1)\ (\mathrm{map}\ ((+)\ 1)\ \mathit{naturals})) \\
&= \ldots
\end{aligned}
$$

Figure 1: The Evaluation Sequence for Creating a List of Natural Numbers

this integer is added to this 'global' list. This newly added integer is immediately accessible to all other processes, since the result of the expression, namely *naturals*, is passed as an argument to all these processes. This becomes clear when we look at Figure 1 showing some steps of the evaluation sequence of the function *naturals*.

As we can see from this evaluation sequence, the occurences of the object that is just being defined can be seen as a *backward reference* or as a pointer to the data stucture that is just being defined. Due to the philosophy of a non-strict language, the whole compound data structure may be used even before all of its components have been evaluated. This provides the possibility of producing a circular data structure by using the data structure that is just being defined in the body of its definition. A language that provides this possibility is necessary for writing circular programs.

Intuitively it should be clear that the n-th element of the list contains the integer n. Due to the laziness of the list, an access to the n-th element will cause as many recursive unfoldings as are necessary to create the n-th element. However, in general it is not obvious that such a recursive definition of a list guarantees that for each n the computation of the n-th element will terminate (which is necessary for this element to be accessible).

In Section 7 a calculus will be sketched very roughly that allows to prove that all elements of a list are accessible. With this calculus, that isn't discussed in detail in this paper, it is possible to prove that in the list of natural numbers and in the list of Fibonacci Numbers (that will be presented in the next section) all elements are indeed accessible.

A very intuitive way of describing such a function is the *dataflow model*. In this model a function is symbolized by a box with in-arrows (the parameters of the function) and with an out-arrow (the result of the function). The data is thought to be flowing through this graph. When it enters a function (which is a box in the graph) and all other entries of the function are also present, a computation takes place and the result of the computaion flows out of the function. As the result of one function may be the input of another function, these arrows can be connected. So these arrows show the data dependencies between the functions. The resulting graph shows the dynamic behaviour of the program very well and the paths where data is flowing. For a detailed description of the dataflow model see [Schreiner, 1990].

Graphs that show the data dependence in a program are known under different names in the literature. In [Field and Harrison, 1988] for example they are called 'process networks'. The connection of such process networks by connecting the out-arrows of one network with the in-arrows of another network is called 'knot tying' there. In [Wentworth, 1989] such graphs are called 'Henderson network diagrams' due to P. Henderson who was one of the first who dealt with the topic of circularity (see [Henderson, 1980]). In [Kelly, 1989] the

'Kahn Principle' is mentioned, which describes the relationship between process networks (as he calls the graphs) and the program it represents. However, we chose the name 'dataflow graph' for this kind of graphical representation to stress the fact that they can be used as a basis of a computational model namely the dataflow model.

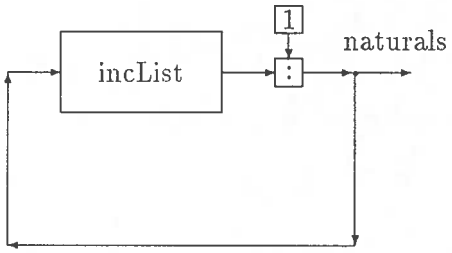


Figure 2: Dataflow Graph of *naturals*

Translating this example of creating a list of all natural numbers into the dataflow model yields the graph that is shown in Figure 2. From this diagram we see that a list is circulating in the graph. A newly constructed partial list is pushed to the left, from where it is fed back to the whole graph. So, in a dataflow graph a circular program is characterized by a circularity in the dataflow graph itself.

2.2 A Circular Program for Generating the List of all Fibonacci Numbers

In the previous example it was shown how to create a circular data structure and how such a data structure can be used in a program. This example will now show how to define a list that is not circular but whose computation depends upon itself.

In the literature of circular programming the generation of a list of Fibonacci Numbers is one of the most famous examples where a circular program is a very natural solution but also a very efficient solution (see [Field and Harrison, 1988, Page 73], [Wentworth, 1989, Page 88f], [Kelly, 1989, Page85ff]). The mathematical definition of the Fibonacci Numbers is given in Figure 3.

$$
\begin{aligned}
f_1 &= 1 \\
f_2 &= 1 \\
f_n &= f_{n-1} + f_{n-2} \qquad n > 2
\end{aligned}
$$

Figure 3: Definition of the Fibonacci Numbers

From the definition of the Fibonacci numbers in Figure 3 it can be seen that the n-th element depends on the $(n-1)$-st and on the $(n-2)$-nd element. In a straightforward

program these elements would be gained by recursive calls to the function f. But since we want to create a list of *all* Fibonacci Numbers we don't need a function that computes the n-th Fibonacci Number. Instead we can give a list comprehension that directly reflects the definition of a Fibonacci Number. Now, the Fibonacci Number f_n is nothing but the n-th element of the list. So, we have to access the $(n-1)$-st and the $(n-2)$-nd elements of the list to compute the n-th element of the list. The resulting circular program can be found in Figure 4. Due to the usage of a list comprehension with its very succinct syntax, the definition of a Fibonacci Number can be seen rather easily in the definiton of the list[4].

$$\begin{aligned} \mathit{fibs} \textbf{ where } \mathit{fibs} \;=\; & 1 : 1 : \\ & [\mathit{fibs} \,!\, (i-1) + \mathit{fibs} \,!\, (i-2) \mid i \leftarrow [2..]] \end{aligned}$$

Figure 4: List of all Fibonacci Numbers

In [Abelson and Sussman, 1985][Chapter 3.4.4, p.270] it is shown how to formulate such a program like in Figure 4 by using explicit *delay* and *force* commands. The meaning of these two commands is as follows: A construct (*delay exp*) doesn't evaluate the expression *exp* but it returns a delayed object, which we can think of as a promise to evaluate *exp* at some future time. On the other hand the *force* command takes a delayed object and performs the evaluation. With these constructs so-called streams can be realized, which are lists that have a delayed object as its tail. To realize such streams, constructor and destructor functions for streams have to written. They can be found in Figure 5. So all one has to do to get a program with explicit *delay* and *force* that is equivalent to the program in Figure 4 is to replace the ordinary list constructor and destructor functions by the appropriate stream functions in the program of Figure 4 and in all functions that are used there.

$$\begin{aligned} x \;:'\; y &= x \;:\; (\mathit{delay}\; y) \\ \mathit{head}'\; x &= \text{head}\; x \\ \mathit{tail}'\; x &= \mathit{force}\; (\text{tail}\; x) \end{aligned}$$

Figure 5: Stream constructor and destructor functions

Again we can draw the dataflow graph for this example making the evaluation process of the whole list clearer. Unfortunately, the translation of a list comprehension into a dataflow graph is not obvious. Informally speaking, the list comprehension in Figure 4 takes the list [2..] and applies the function $\lambda i.\mathit{fibs}!(i-1) + \mathit{fibs}!(i-2)$ on each element of the list. Therefore, the list comprehension is equivalent to the following construct:

$$\text{map}\; f\; [2..] \textbf{ where } f\; i \;=\; \mathit{fibs} \,!\, (i-1) \;+\; \mathit{fibs} \,!\, (i-2)$$

Now, this construct can be rather easily represented as a dataflow graph. The dataflow graph in Figure 6 shows on the one hand that the first two elements of the list must be

[4] For a definition of '!' see Appendix A.2.

present to start the computation. It also shows the circularity that is gained by feeding *fibs* back such that it can be used for computing the next element of the list.

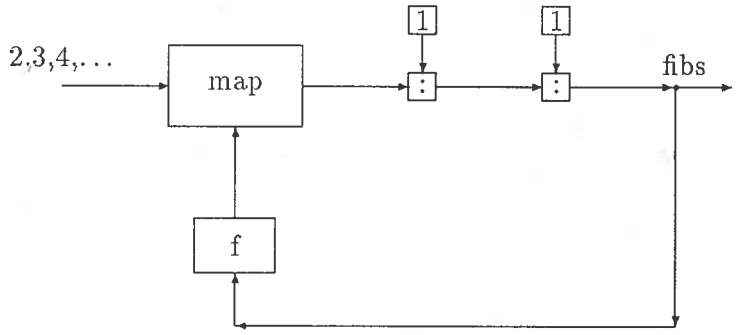


Figure 6: Dataflow Graph of *fibs*

This example of the Fibonacci Numbers is very appropriate for showing the features and the advantages of circular programs. In this section the following items are stressed:

- The list of Fibonacci Numbers is not a circular list but the computation of the list depends upon itself. Therefore, we can say that a program that uses this list is a circular program.
- Using such a circular definition yields an easily readable program.
- The overall structure is similar to the example in the previous section. This fact will be used in the next section to derive a general structure for circular programs.

In Section 6 this example will be used to show, how a general recursive program can be transformed into a circular program, thereby gaining space and time efficiency.

3 The General Structure of a Circular Program

From the example in the previous section we can already see the characteristics of a circular program. The example leads us to the following general scheme for circular programs:

$$ds \quad \textbf{where} \quad ds = f\ (ds)$$

The most important part is the recursive definition of the object *ds* (in the example of Section 2.1 this was the list *naturals*). The object *ds* is defined by calling a function *f* and submitting *ds* as a parameter to this function.

Again we can use the dataflow model to visualize the circularity of this scheme. The resulting graph is shown in Figure 7. If the function *f* had additional parameters they would be fed into the function from the left, arriving at the function together with the circular

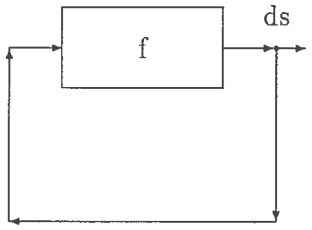


Figure 7: Dataflow Graph of the General Structure for Circular Programs

object. For interesting circular programs the function f might be rather complicated but the characteristic point of the dataflow graph of a circular program is the connection from the output of this function f to its input.

From this general structure of a circular program one can also see that the underlying language must have two properties to allow such definitions:

- The language must have a *non-strict semantics* and
- it must allow *local recursion.*

A non-strict semantics of the language is necessary because with strict semantics a recursive definition of an object will result in an endless recursion as all arguments have to be evaluated before the function may be called.

Remark: At this point it is important to make a distinction between non-strictness and lazy-evaluation:

- *Non-strictness* means that a function may return a result even if some arguments have not been evaluated yet. For data structures this means that the data structure may be used even if parts of the data structure have not been evaluated. Non-strictness is a notion concerning the **semantics** of a programming language.
- *Lazy-evaluation* means that the argument of a function is only evaluated when it is really needed. For data structures this means that a part of it is only evaluated when that part is needed. Lazy-evaluation is a notion concerning the **implementation** of a programming language.

Obviously, lazy evaluation is a correct implementation of a non-strict programming language since with this evaluation strategy a function may return a value before the arguments have been evaluated. But another possibillity for implementing a non-strict language would be to perform the evaluation of the function and of all of its arguments in parallel. At the time when the function is called the arguments have not been evaluated.

```
primes 1 = [ ]
primes 2 = [2]
primes n = 2 : 3 : p  where  p = sieve (3 : p) 5 n

sieve p t n = [ ]                                          , if n < t
              cons p r t  where  r = sieve p (t + 2) n     , otherwise

cons (p : l) r t = t : r          , if t < p * p
                   r              , if t mod p == 0
                   cons l r t     , otherwise
```

Figure 8: Prime Number Generator

On the other hand, local recursion is necessary to produce a circular data structure as it is necessary to define recursive functions. In both cases the item that is defined (either a function or an object) has to be used in the body of the definition. This is possible with local recursion since there the scope of an item includes the body of its definition.

This general structure of a circular program with a more detailed description can also be found in [Allison, 1989]. In [Bird, 1984] an example is given to explain the necessity of the above properties of the language to allow circular programs.

4 The Efficiency of Circular Programs

4.1 Another Example: A Prime Number Generator

An interesting problem, where a circular program can be used, is a prime number generator. The principle of the 'Sieve of Eratosthenes' can be used for such a program. The main idea of this principle is that when a list of prime numbers up to a bound n is given it can be used to generate all prime numbers independently up to n^2. The resulting list can then again be the input of another call of this function. Obviously, this list is the circular argument since it is input and output of the same function.

The following program and the dataflow graph are taken from [Schreiner, 1990], where also a non-circular and rather straightforward program are described. In [Allison, 1989][Page 106f] a similar program is presented. However, the program in the latter paper uses higher-order functions and is therefore more succinct than the one mentioned here.

Figure 8 shows the circular program from [Schreiner, 1990][Page 32] realizing the 'Sieve of Eratosthenes'. The meaning of the various functions and of their arguments is as follows:

- *primes:*

  Input: n ...A natural number (the upper bound).
  Output: A list of all prime numbers up to the upper bound n.

Behaviour: In the non-trivial case a circular argument p is produced by the call in the 'where' part of the definition. Due to the specification of *sieve*, p represents the list of all prime number between 5 and n. So, together with 2 and 3 this is the result of *primes*.

- *sieve:*

  Input: p ...The list of all prime numbers up to n.
  t ...The natural number that is tested for primality.
  n ...A natural number (the upper bound).
  Output: A list of all prime numbers greater than or equal t and less than or equal n.

  Behaviour: If t is smaller than the upper bound n, the list of all prime numbers greater than or equal $t+2$ is computed in r. After that the *cons* function adds t to this list r and returns the new list if t is prime and otherwise returns r itself.

- *cons:*

  Input: $p : l$ ...The list of all prime numbers up to n.
  r ...The list of all prime numbers greater or equal than $t+2$.
  t ...The natural number that is tested if it is prime.
  Output: r with t consed to the head if it is prime.

  Behaviour: If t is larger than p^2 then t is prime since it can not be a multiple of any prime number of $p : l$. However, if it is divisible by the prime number p it is no prime and therefore r is returned. Otherwise, the test of divisibilty has to be continued and therefore *cons* is recursively called with the prime list without p.

One important property of this program is that the circularity of the whole program does only affect the top-level function. All the other functions are rather straightforward and can be understood without taking this circularity into account. Furthermore, the top-level function shows the typical structure of a circular program.

Especially for a more complex program like this one it is important for understanding the dynamic behaviour of the program to derive the dataflow graph out of the given program, which can be seen in Figure 9. From this graph we see that there are two directions of communication:

- The list p is an input argument for *sieve* and therefore it is pushed to the right.
- The list r contains the result of one function call and is therfore pushed to the left.

As p is the circular argument in this program it is not only the output of the first call to *sieve* but also an input argument to the same call. This can be seen in Figure 9 where p is pushed back from the lower output pipeline to the upper input pipeline.

With this dataflow graph it is much easier to understand the dynamic behaviour of the program. At the beginning the upper bound n arrives at *primes* and therefore the computation can start. Usually the otherwise branch will be reached and there p is bound to the result of *sieve* $3 : p\ 5\ n$. As p is not yet defined, the first argument of this first call to *sieve* is the list consisting of 3 and some undefined rest. In *sieve* usually *cons* will be called with the same input list. Now, if all elements of the input list up to $\sqrt{t}$ have already

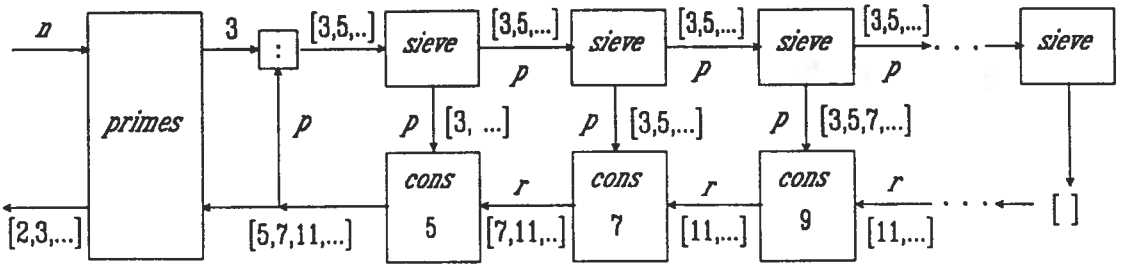


Figure 9: The Primes Circuit

been evaluated, it can be decided whether t is prime or not (otherwise this decision would have to be delayed until enough elements will be evaluated). In the first case ($t = 5$) the decision can be made and therefore *cons* adds 5 to the list it gets from the right (which is undefined until now). The resulting list, which is now $[5, ...]$, is pushed back to the input of the first call of *sieve* from where it flows to the right again. So, we see that the list p, which is the circular argument, really 'circles' in the dataflow graph.

Although to each activation of *cons* the list $p : l$ of all primes less than or equal n is passed, only the list elements up to at most t are needed by *cons*. Hence, a new prime t may be computed by only referring to the previously computed prime elements and the program does not run into a deadlock. Now, if t is prime it is consed to r. Since the result of this call to *cons* is also the result of *sieve* (*cons* is called only from one place) which is bound to p in the where-clause of the definition of *primes*, this newly added prime is entered into p and is therefore accessible to all other processes, too.

However, it has to be emphasized that this is only a model of thinking. There are no independent objects flowing! Instead we have one 'global' list p into which entries are made. Due to the circularity of the program all the calls of *sieve* have immediately access to all the entries in p. To make this point clearer let us look at the evaluation sequence of this program that is shown in Figure 10.

From this evaluation sequence we see that the p which is an argument of *sieve* and of *cons* is the same p which is just being constructed. Therefore it is in the last line possible to access the first element of p which has been computed in the third line. From this it should be clear what we mean by saying that p is a 'global' list and that there are actually no elements flowing in the memory or anything like that.

Both of these models together form a very useful tool for understanding the static and the dynamic behaviour of a circular program. From the example we can therefore summarize:

$$primes = 2 : 3 : p$$

$$\begin{array}{lll}
p & = & sieve\ 3 : p\ 5\ n \\
& = & cons\ 3 : p\ (sieve\ 3 : p\ 7\ n)\ 5 & 5 < 3^2 \\
& = & 5 : (sieve\ 3 : p\ 7\ n) \\
& = & 5 : (cons\ 3 : p\ (sieve\ 3 : p\ 9\ n)\ 7) & 7 < 3^2 \\
& = & 5 : 7 : (sieve\ 3 : p\ 9\ n) \\
& = & 5 : 7 : (cons\ 3 : p\ (sieve\ 3 : p\ 11\ n)\ 9) & 9\ \mathbf{MOD}\ 3 = 0 \\
& = & 5 : 7 : (sieve\ 3 : p\ 11\ n) \\
& = & 5 : 7 : (cons\ 3 : p\ (sieve\ 3 : p\ 13\ n)\ 11) & 13 \geq 3^2 \\
& = & 5 : 7 : (cons\ p\ (sieve\ 3 : p\ 13\ n)\ 11) & 11 < 5^2 \\
& = & 5 : 7 : 11 : (sieve\ 3 : p\ 13\ n) \\
& = & \ldots
\end{array}$$

Figure 10: Evaluation Sequence of the Prime Number Generator

- The *dataflow model* is very useful to understand the **dynamic behaviour** of a program. It shows the data dependencies in the program. It can also be used to see the parallelism which lies in a program (see Section 4.4).
- The *evaluation model* (which yields an evaluation sequence when the execution of the program is simulated) is very useful to understand the **static behaviour** of a program. It can therefore be used to detect such 'global' data structures in a program like the one described above. It also offers a possibility to deduce the sequence of function calls under the assumption of lazy evaluation.

For a detailed discussion of a straightforward algorithm, of the above circular algorithms and of its dataflow graph see [Schreiner, 1990][Chapter 2.4 Dataflow Programs].

4.2 Another Example: Partitioning

In this section a problem is presented that is very appropriate for being solved with a circular program and that also shows the improved efficiency of the circular program very well. In the partitioning problem one wants to find all possible partitions of a given natural number. A partition of a number n is a list of natural numbers such that the sum of all elements of the list is the given number. In a partition the order of the elements does not matter. The formal defintion of the problem can be found in Figure 11.

Now suppose that we want to compute the list of all partitions for all integers. So the list should look like $[p_0, p_1, p_2, \ldots]$ where p_i is the result of partitioning i. The idea for solving this problem is essentially the following: If we want to compute all partitions of n we create all lists that start with 1 and are then followed by any partition of $n-1$. Then we create all lists that start with 2 and are then followed by any partition of $n-2$ and so on. In each such step we get a list of partitions. So finally we have to combine all these lists together to one list, which is the solution. Based on this function for partitioning an integer n we

Input: n ... A natural number.
Output: l ... A list of all partitions of n, where
x is a partition of l iff
x is a list of positive natural numbers $x_1, \ldots, x_k$ such that
$x_1 + \ldots + x_k = n \ \wedge \ x_1 \leq x_2 \ldots \ \wedge \ x_{k-1} \leq x_k$

Figure 11: Problem-Specification of Partition

can now define the list of all partitions for all integers with a list comprehension where the index runs from 0 to ∞.

A straightforward program that implements this idea by using recursive calls to the main function can be found in Figure 12. There one auxiliary function *testCons* is needed. This function guarantees that every resulting list of integers is non-decreasing. This is done by adding the integer x to a list ys only if x is not larger than the first element of ys which must be the minimum of ys since ys is itself non-decreasing. As *testCons* i is mapped to the list of partitions of $n - i$ only the non-decreasing partitions are added in the list-comprehension. Since *testCons* might produce empty lists these empty lists have to be cancelled by the $+++$ function, which is an append that deletes every empty list.

The idea for a circular program is here essentially the same as for the Fibonacci Numbers. We can replace each recursive call to the function *part* by fetching the appropriate element from the list with all partitons for all integers. The base cases of the function *part* are put at the beginning of this global list. This guarantees that the computation of the whole list can start. The rest of the list is a list-comprehension where the qualified expression computes the partitions for k where k runs through all integers starting from 2. The circular program can be seen in Figure 13. The auxiliary functions *testCons* and $+++$ are unchanged.

4.3 Circular Programs are Space Efficient

One reason for a functional program being very slow might be that it produces many intermediate structures (e.g. lists). For example think of a straightforward implementation of the above *primes* algorithm. In such an algorithm there would be essentially one call of the function *sieve* for every odd number as every odd number is tested whether it is a prime number or not. If the number of one call of *sieve* is prime this number will be added to the list of all prime numbers smaller than this one and the whole list will be submitted to the next call of *sieve* testing the next odd number. So, for each function call of *sieve* an intermediate list is produced although only the last list is used as the result of the whole computation. All these intermediate lists are therefore garbage at the end of the computation wasting very much space. Assuming that about $O(n/\log n)$ prime numbers are smaller than n, each function call creates a list of $O(n/\log n)$ elements which is, except for the last one, nothing but garbage. As every odd number has to be tested the total number of list elements that are produced is $O(\sum_{k=1}^{\lfloor n/2 \rfloor} (2*k+1)/\log(2*k+1))$.

In contrast to this behaviour of a conventional program, the circular program does not waste any space as there is only one 'global' list used in the whole computation as it was

```
allpart = [part n | n ← [0..]]

part 0 = [[ ]]
part 1 = [[1]]
part n = reduceR (+++) [ ]
         [map (testCons i)(part (n − i)) | i ← [1..n]]

testCons x [ ] = [x]
testCons x ys  = x : ys   , if x ≤ hd ys
               = [ ]      , otherwise

[ ] +++ ys       = ys
(x : xs) +++ ys  = xs +++ ys        , if x == [ ]
                 = x : (xs +++ ys)  , otherwise
```

Figure 12: Straightforward Partition Program

```
allcpart = p
           where p =  [[ ]] : [[1]] :
                      [reduceR (+++) [ ]
                      [map (testCons i)(p ! (k − i)) | i ← [1..k]]
                      | k ← [2..]]
```

Figure 13: Circular Partition Program

shown in Section 4.1. So the total number of list elements is in this case $O(n/\log n)$. In this case the increase of space efficiency is really enormous. Although it would be very optimistic to expect such increases of space efficiency in every case some increase can often be achieved if we have large data structures as arguments to recursively called functions. And as we have seen above, it is often the case that most of the needed space is consumed directly or indirectly by one argument of an function.

As in the primes example also in the partitioning example there is a large gain of space efficiency. If we look at the straightforward partitioning algorithm in Figure 12 we see that in the partitioning of the integer n recursive calls *part* i for each $i < n$ have to be performed. Furthermore, one can see that each call to *part* i creates some intermediate lists since an extended append operation is performed on the resulting list via *reduceR* $(+\!+\!+)\ldots$. So, in the call of *part* n intermediate lists for every $i < n$ are created. On the contary, the circular program directly accesses the result of partitioning i for every $i < n$ and so no intermediate lists are produced for these i. So, in the circular program we have far fewer intermediate lists than in the straightforward program.

In an implementation of this program in RUFL the difference becomes obvious: The straightforward algorithm is only able to partition all integers up to 20 (afterwards it runs out of space). On the other side, the circular program manages to partition integers up to 23. For details of the implementation of this algorithm see Appendix B.

In [Bird, 1984] it is shown how an algorithm that performs several passes on a tree can be transformed into an algorithm that needs only one pass. It is noticeable that in the multi-pass algorithm the argument tree is needed twice. The two travesals of the tree are performed on these two instances of the tree. On the other side, in the one-pass algorithm the argument tree occurs only once in the body of the top level function definition, indicating that only one pass is performed. Regarding that garbage collection is in most implementations of functional languages a very time consuming process this also helps to increase the time efficiency of a program.

4.4 Circular Programs are Time Efficient

In the previous section it has been described that the straightforward partitioning program performs recursive calls *part* i for every $i < n$ in order to compute *part* n. This means that the total number of calls to *part* for partitioning all integers up to n is $\sum_{i=1}^{n-1} i = (n-1) * n/2 = O(n^2)$. On the other hand, the circular program doesn't perform a recursive call at all. It just takes the results of previous calls to *part* that have been stored in the global list and combines them to the partition of n. Therefore, in this case the number of calls to *part* for partitioning all integers up to n is just $n-1 = O(n)$. So, we see that the use of a circular program has reduced the complexity of the program (in terms of calls to the main function) from quadratic to linear.

Details of the implementation of the partitioning problem can be found in Appendix B. Table 1 shows a comparison between the straightforward and the circular algorithm for the partitioning problem. The second column in this tables shows the maximal integer up to which all integers can be partitioned with each algorithm without getting a memory overflow. The last column shows the computation time for partitioning all integers up to 20. This last column shows the enormous speed-up that is achieved by using a circular algorithm. The reasons for this speed-up are on the one hand the reduced complexity of

Program	Highest Possible Input	Computation Time for Input 20
Straightforward Program	20	25 min.
Circular Program	23	10 sec.

Table 1: Computation Results for a Straightforward and a Circular Algorithm of the Partitioning Example

the algorithm as it was described above. On the other hand, the improved space efficiency of the program probably yields less garbage collection for the circular program than for the straightforward program. Only both arguments together can justify a speed-up as it was achieved in this example.

Up to now our underlying model of computation has been a sequential one. But it has already been mentioned that non-strict functional languages in general and circular programs in especial contain lots of inherent parallelism. Exploiting this inherent parallelism by a parallel implementation of a functional language would drastically increase the time efficiency of a circular program.

Let us once more look at the circular program for solving the partitioning example in Figure 13. Assume that the partitions of all integers up to some n have already been computed. This means that some part of the partitioning of all larger numbers can also be computed. Especially, all those parts can be computed in parallel that only depend on already computed partitions. As the new partitions are added to the same 'global' list, we have essentially the same structure of the program as in the primes example.

To show this inherent parallelism in more detail again look at the example of Section 4.1. The first function call of *sieve* performs a recursive call. However, as we have seen above the result of this function call is not needed by *cons* which therefore can return its prime number without waiting for the return value r of the recursive call. Assuming lazy evaluation this would mean that the evaluation of the recursive call is delayed. But it is also possible to perform the evaluation of this recursive call in parallel to the evaluation of *cons*. Now, again the first thing that this recursive call will do after having checked that $n \geq t$ is to start a process to evaluate the recursive *sieve* call. Afterwards it will start to check if its number t is prime or not. That means that all the *sieve* processes are created very soon and therefore the checks of primality in the *cons* processes can be performed almost in parallel. Only when a process needs an entry of the list of prime numbers that has not been evaluated yet it is delayed until some other process creates this entry. So we have a large number of parallel processes that are synchronized via a 'global', non-strict data structure.

This circular program and another straightforward program realizing the 'Sieve of Eratosthenes' have been implemented in the assembler code of an abstract dataflow machine that was designed and implemented on a transputer network (see [Schreiner, 1990]). The results of the computation of both algorithms are described in Table 2, which is taken

Program	Input	1 processor 2 processes	16 processors 16 processes
Straightforward Program	2000	302	41
Circular Program	10000	237	51

Table 2: Computation Results for a Straightforward and a Circular Algorithm of the Primes Example

from this thesis (page 146). In this table the two algorithms are compared with different inputs n. The second and the third columns show the computation times (in seconds) of the algorithms on different numbers of processors.

From these results the increase of time efficiency is obvious.

This example shows that a parallel implementation of a non-strict functional language and the use of circular programs together yield highly efficient programs. These programs have computation times that are comparable with equivalent programs written in an imperative language such as C. On the other side non-strict functional programming languages have many advantages such as referential transparency, the possibility to perform mathematical transformations and so on. So, this programming technique may contribute to increase the acceptance of such languages in the community of computer scientists.

5 Eliminating Multiple Traversals of Data

As a last example of a circular program let us now look at a program that works not on lists but on binary trees. With this compound data structure it can be shown how circular programs can reduce the number of traversals that have to be performed. Thereby, the time efficiency (by performing only one traversal instead of several ones) as well as the space efficiency (by not having to create intermediate trees between the several traversals) can be improved.

One of the first papers that dealt with circular programs was [Bird, 1984]. There it was shown how to transform a program that traverses a tree several times into one that makes only one traversal by using circularity. To explain how this can be done, let us look at the example that is presented in this paper. The compound data structure in this example is a tree with integers as its tip values. The problem we want to solve is the following:

Given: A tree t with the integers $x_1, \ldots, x_n$ as its tip values.

Find: A tree $\bar{t}$ that has the same shape as t but that has $\min(x_1 \ \ldots \ x_n)$ in each of its tips.

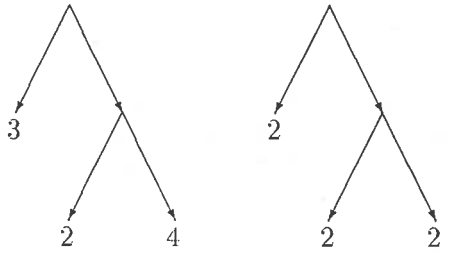


Figure 14: The Specification of a Tree Traversing Algorithm (Input Tree and the Corresponding Output Tree)

```
data tree              = TIP Int | FORK tree tree

transform t            = replace t (tmin t)

replace (TIP n) m      = TIP m
replace (FORK L R) m   = FORK (replace L m) (replace R m)

tmin (TIP n)           = n
tmin (FORK L R)        = min (tmin L) (tmin R)
```

Figure 15: The Straightforward Algorithm

$$
\begin{array}{lcl}
\mathit{transform}\ t & = & \mathrm{fst}\ p \\
 & & \textbf{where}\ p = \mathit{repmin}\ t\ (\mathrm{snd}\ p) \\
\mathit{repmin}\ (\textsc{fork}\ L\ R)\ m & = & [\textsc{fork}\ t_1\ t_2, \min\ m_1\ m_2] \\
 & & \textbf{where}\ [t_1, m_1] = \mathit{repmin}\ L\ m \\
 & & [t_2, m_2] = \mathit{repmin}\ R\ m \\
\mathit{repmin}\ (\textsc{tip}\ n)\ m & = & [\textsc{tip}\ m, n]
\end{array}
$$

Figure 16: The Circular Algorithm

Figure 14 shows the transformation that should be performed on a rather simple tree with three tips. Note that all the tip values of the resulting tree are the same namely $2 = \min(3\ 2\ 4)$.

A straightforward functional algorithm that solves this problem can be found in Figure 15. First of all the datatype of a tree with integers as tip values is introduced. The algorithm uses two sub-functions: *replace* and *tmin*. Each of the functions traverses the tree once, which is rather obvious from the definitions of the functions. Now, the whole problem can be solved by first computing the minimal tip value of the tree (this is done by *tmin*) and afterwards replacing each tip value by this minimal value. This is done by the top-level function *transform*.

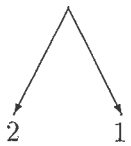
Looking at the this straightforward algorithm in Figure 15 the two traversals are quite obvious. Another item that was already mentioned in Section 4.3 is that the input tree, which might be very large, is copied when executing *transform* since the argument t appears twice in the body of this function. So, this is the point where the inefficiency lies since from this point on a second traversal becomes inevitable.

The idea for getting a one pass algorithm is to avoid a second traversal of the tree by combining both sub-functions of the straightforward algorithm to one function *repmin* that has the following property:

$$\mathit{repmin}\ t\ m\ = [\mathit{replace}\ t\ m, \mathit{tmin}\ t]$$

As we can see from Figure 16, a function *repmin* with this property can be defined that does not traverse the tree t twice. Until now no circularity has been introduced into our program. But when we examine the specification of *repmin* we see that for the computation of the first component in the tuple we need the minimal tip value of the tree, which is the second part of the same tuple. This gives rise to the definition of *transform* in Figure 16. This definition is circular since the second part of the output of *transform* is also an input to the same function.

Figure 17 shows the evaluation sequence of this circular algorithm for the following tree



transform (FORK (TIP 2) (TIP 1)) =

=	fst $\downarrow_0$ (*repmin* (FORK (TIP 2) (TIP 1)) (snd $\uparrow_0$))	
=	fst $\downarrow_0$ [FORK (fst $\uparrow_1$) (fst $\uparrow_2$), min (snd $\uparrow_1$) (snd $\uparrow_2$)]	$\downarrow_1$ *repmin* (TIP 2) (snd $\uparrow_0$) $\downarrow_2$ *repmin* (TIP 1) (snd $\uparrow_0$)
=	$\uparrow_3$	$\downarrow_0$ [$\uparrow_3$, min (snd $\uparrow_1$) (snd $\uparrow_2$)] $\downarrow_1$ *repmin* (TIP 2) (snd $\uparrow_0$) $\downarrow_2$ *repmin* (TIP 1) (snd $\uparrow_0$) $\downarrow_3$ FORK (fst $\uparrow_1$) (fst $\uparrow_2$)
=	$\uparrow_3$	$\downarrow_0$ [$\uparrow_3$, min (snd $\uparrow_1$) (snd $\uparrow_2$)] $\downarrow_1$ [(TIP (snd $\uparrow_0$)), 2] $\downarrow_2$ [(TIP (snd $\uparrow_0$)), 1] $\downarrow_3$ FORK (fst $\uparrow_1$) (fst $\uparrow_2$)
=	$\uparrow_3$	$\downarrow_0$ [$\uparrow_3$, min 2 1] $\downarrow_3$ FORK (TIP (snd $\uparrow_0$)) (TIP (snd $\uparrow_0$))
=	FORK (TIP 1) (TIP 1)	

Figure 17: An Evaluation Sequence of the Circular Algorithm

that is represented by FORK (TIP 2) (TIP 1).

As circular data structures are built in the evaluation of *transform*, pointers to data structures have to be introduced. So, $\uparrow_0, \uparrow_1, \uparrow_2$ are such pointers to tuples consisting of a tree and an integer. Corresponding to these pointers $\downarrow_0, \downarrow_1, \downarrow_2$ are used to denote the tuples, where these pointers are referring to. The tuples that are referenced in the evaluation sequence are written at the right margin of Figure 17. A consequence of using such pointers is that in line 2, after having performed the fst operation, we can not simply drop the tuple because it is used by the tuples that are referenced by $\downarrow_1$ and $\downarrow_2$. We therefore use a new pointer $\uparrow_3$ to denote the first part of the tuple. This first part is now referenced by the tuple itself and by the expression that is just being evaluated (it is 'shared'). The only thing we have to do from this step on is to 'chase the pointers' and to perform the correct operations on them. Note that only in the last step (but not sooner) the tuple that is referenced by $\uparrow_0$ can be dropped since the $\uparrow_0$ pointers have already been resolved at that moment.

From this evaluation sequence of the circular algorithm it becomes clear that this algorithm performs only one traversal of the whole tree. The essential point in this evaluation sequence is that both parts of the tuple always refer to the same tree. The termination of the circular program is guaranteed since the minimal tip value is only needed when in the traversal of the tree the tip values have been reached but not earlier. However, at that point this minimal value can easily be computed by simply applying the min function on these tip values.

From this example we again see the increase of efficiency that is achieved with a circular program by introducing a 'global' data structure. In this case care has to be taken about the meaning of 'global'. Of course, it does not mean that all the operations are performed on the input tree, thereby altering the contents of the tree. Such destructive operations would violate the principle of referential transparency and therefore are forbidden in functional programming languages. What 'global' means in this context is that only one instance of the input tree is passed to the next function and all operations refer to this single instance of the tree. The reason for introducing all these pointers in the evaluation sequence in Figure 17 is that all the computation is based on only one such 'global' object, which is rather complex.

Remark: In the original paper [Bird, 1984] the emphasizes lies on showing how to automatically perform the transformations that yield such a circular program. The goal of this process is that a programmer need not know anything about circular programs. He writes a straightforward but inefficient program, which is afterwards transformed into a very efficient circular program. Therefore, this topic should also be very interesting for those people, who work in the areas of program transformation and compiler design.

6 The Expressive Power of Circular Programs

In this section we try to draw some generalizations from the examples in the previous sections. However, this should be regarded more as an outlook on some future work in the area of circular programs rather than a description of some concluded research.

If one compares the circular programs for the Fibonacci Number example in Figure 4 and for the Partitioning example in Figure 13 one can easily detect a common structure in these to programs. In both programs a list of all function values of the main function is created. Therefore, a recursive call to this main function can be replaced by accessing the appropriate element in the list. After having computed a new function value this value is added to the list that is identical to the list that is used for looking up the already computed function values.

This principle can be used for any recursive function definition. So if we have some arbitrary function $f : \alpha \to \beta$ and and arbitrary function $g : (\beta \to \ldots \to \beta)_{k-times} \to \beta$ the general recursive definition of f that is shown in Figure 18 can be replaced by the circular program that is shown in Figure 19. In these figures $n, i_1, \ldots, i_k$ are integers.

$$\begin{aligned} f\ 1 &= a \\ f\ n &= g\ (f\ (n - i_1))\ \ldots (f\ (n - i_k)) \end{aligned}$$

Figure 18: General Recursive Program

$$\begin{aligned} f\ n\ =\ & p \\ & \textbf{where}\ p = a : [g\ f_1\ \ldots f_k\ \textbf{where} \\ & \qquad f_1 = p\,!\,(n - i_1),\ \ldots,\ f_k = p\,!\,(n - i_k) \\ & \qquad \mid n \leftarrow [2..]] \end{aligned}$$

Figure 19: General Circular Program

Now we see that the programs for computing the Fibonacci Numbers and for Partitioning are only special instances of the general scheme[5]. This scheme should clarify that such a transformation can be performed for a quite large class of programs.

Actually, the above scheme could be formulated even more general. We could replace the integer parameter n by several parameters $x_1, \ldots, x_l$ from arbitrary domains $D_1, \ldots, D_l$. In this case we would need a bijective function $h : D_1 \to \ldots \to D_l \overset{bij.}{\to} \mathbf{N}$ that assigns to each input of the function f the index where the result of $f\ x_1\ \ldots x_l$ is stored in the global list. This shows that the whole principle of program transformation is neither restricted to the number nor to the types of the parameters of f.

Another fact that might be not obvious yet is that this transformation principle also allows the handling of the non-functional construct of a loop. It is well known that a loop can be expressed by a simple tail recursion. But as we now can transform a recursive program into a circular program we can also transform a loop into a circular program. The resulting list that we get by this transformation is essentially the 'graph' of the loop. That means that

[5] One slight deviation is the usage of two initial arguments in both programs instead of only one in the general scheme.

the i-th element of the list is the result of i unfoldings of the body of the loop provided that the loop condition remains true. Otherwise, it is the result of k unfoldings of the loop body where k is the minimal number of unfoldings after that the loop condition turns to false. The result of the transformed loop itself is the fixpoint of the list.

Summerizing we can say the following about the expressive power of circular programs:

- A big class of recursive programs can quite automatically be transformed into a circular program. The idea of the transformation is to replace all recursive calls to the function by accessing elements of the 'global' list of all function values.
- This transformation principle also includes the possibility to transform loops into circular programs, thereby embedding the non-functional construct of a loop into a purely functional language.
- The usage of a circular program offers the possibility of improving the space and time efficiency of the program because
  - in a circular program there are usually fewer intermediate data structures created and
  - in a circular program that has been derived by the above transformation it is guaranteed that no function call to the main function is evaluated twice for the same arguments. This is obvious since in the circular program all the results of already computed function calls are stored in the 'global' list and can therefore be directly accessed by all computations. This characteristic reminds of the concept of 'Lazy Memo-functions' that has been introduced by J. Hughes in [Hughes, 1985]. The main difference to these 'Lazy Memo-functions' is that we don't need an extension of the language to get that 'memoization'. All we need is a circular program of the structure that was described above.
- Since we use a non-strict functional language for implementing a circular program we can use infinite lists. Especially the generator that creates the indices for the circularly defined list can be taken from the infinite list of all integers. If lazy-evaluation is used for implementing non-strictness it is guaranteed that only those parts of the list will be computed that are actually necessary for the computation of the solution of the whole program.

7 Termination

The most important item concerning the termination of a circular program is to guarantee that the circular argument is not demanded too early, as this argument is at each step of the computation only partially evaluated. For example in the prime number generator the prime list always has some unevaluated rest. Only when we are sure that we do not need the unevaluated part we can drop it and use the part we have evaluated so far. Now, when we access a part of this list we have to keep this fact in mind. When we look at the functions *sieve* and *cons* we see that the latter function is the only one that accesses the list p of all prime numbers up to n. As it only splits the list into a head and a tail it must be guaranteed that the list has been evaluated far enough to allow such a splitting.

Here this is the case, since the incoming list p contains at least the prime number 3 (as its head) and some unevaluated part (as its tail). And as the previous calls to *sieve* and *cons* add all prime numbers up to the square root of the tested number, each *cons* has enough information from the list to decide whether t is prime or not.

Now, to demonstrate the problem of termination, let us suppose that we want to get access to all elements of the list p in the function *cons*. We can see from the dataflow graph (Figure 9) that at the beginning of the program each function gets a p that has an unevaluated part. Therefore, the list can not be split into its arguments immediately and *cons* has to be delayed. But *cons* itself is the only function that adds elements to this list p since its output is pushed back to the input p. So, *cons* will wait forever and we have a deadlock. So summarizing we can say that this program does not terminate because the elements of the circular argument (namely all the elements of the list) are demanded too early.

One interesting point in this context is that in a process called *strictness analysis* information is derived that may be exploited for deciding whether such a circular program terminates or not. Roughly speaking, the strictness analysis yields the information how far an argument of a function may be evaluated to maintain the non-strict semantics of the function. Since the circular argument is always the argument of one function (see Section 3) we get information how much of the circular argument is needed by the function. Taking into account that the circular argument is not evaluated at all at the beginning, we can now decide if it contains enough information for the function that uses the argument.

Let us again look at the prime number generator. To decide how much of the list p is needed by *sieve* we examine its definition. There the structure of p is not evaluated at all but the list is submitted as an argument to *cons*. So, we also have to look at this function. Now, there the input list is splitted into its head and its tail. Therefore, the list p in the definition of *sieve* has to be evaluated such far. We know that p is the first argument of sieve. Therefore the first argument in the function call of sieve in *primes* also has to be evaluated such far. And this is really the case since this argument is $3 : p$. If the where clause in *primes* was

$$p = \mathit{sieve}\ p\ 3\ n$$

p would not be evaluated far enough and the whole program would not terminate! Looking at the dataflow graph in Figure 9 we see that in this case the *cons* functions would wait forever for the first part of the list to decide whether $t < p * p$ or not.

For more details about strictness analysis see [Burn, 1991], [Clack and Peyton Jones, 1985], [Burn *et al.*, 1986] and [Loogen, 1990][Chapter 6].

7.1 A Calculus for Proving the Termination of Circular Programs

The material that is presented in this section is essentially a short version of what is discussed in detail in [Sijtsma, 1989]. There it is shown how it can be proven that each component of a list is accessible since it is computed in finite time. For that the notion of productivity of a circular program has to be defined formally. Based on this definition some theorems are presented that allow to decide if a circular program yields a list where

each component can be accessed. However, we will only try to sketch the main ideas of this approach in this paper. The interested reader is referred to the paper cited above and to [Sijtsma, 1988], which contains these ideas in a separate chapter and then concentrates on the task of verifying infinite-list programs.

7.1.1 The Notion of Productivity

As we have seen in the previous examples, it is not trivial to decide if a circular program really "works" or if it runs into a deadlock or into an infinite sequence of computations. Such a deadlock might occur when in the computation of the n-th element of a list another element has to be accessed that itself needs the n-th element for its computation. A very simple example for an infinite sequence of computations is the list where the n-th element should be equal to the $(n+1)$-st element:

$$l = [l\,!\,(n+1) \mid n \leftarrow [1..]]$$

To express the fact that the computation of a list is free of deadlocks and of infinite sequences we define the notion of *productivity*. Informally speaking, the productivity of a list implies that every element of the list can be computed in finite time. To achieve this, productivity is defined such that a list is productive iff all of its components are productive. An element in the basic domain *Int* or *Bool* is called productive if it is different from $\perp$. This idea of a definition can be applied to finite lists as well as to infinite lists.

The problem with circular programs that create lists is that at each step we have a list where only some elements are productive, namely those whose value has already been computed. Therefore also the notion of *segment productivity* is useful. This notion allows us to make propositions about how much of a list is productive. If the initial productive segment of the list is as large as the list itself we have the case of a productive list. Therefore, the idea how to show that a circular list really yields a productive list is the following: We prove that the initial part of the list that is productive becomes longer at each step of unfolding the recursive definition of the list.

8 Conclusion

In this paper it has been shown that the programming technique of circular programming can be used to increase the space as well as the time efficiency of a given program. A general scheme for such programs has been presented that reflects the idea of using the data structure that is being defined in the body of its definition. This is only possible for a special class of languages, which have a non-strict semantics and local recursion.

Furthermore, it has been shown that there is a class of circular programs that can be derived quite automatically from a non-circular program. This class shows many very favourable features like 'memoization', which improves the time efficiency of the program. As a quite large area of problems can be covered with such programs it is very important to know the general transformation technique that creates such circular programs.

Two examples were presented that showed different aspects of circular programs:

- The Prime Number Generator of Section 4.1 showed that circular programs can be used to avoid the production of many intermediate structures.
- The Tree Example of Section 5 showed that circular programs can also be used to transform a program that traverses a tree several times into a program that makes only one traversal.

Furthermore, the usefulness of the dataflow model for understanding the data dependencies in a given program has been discussed. This is very important since in a non-strict language those data dependencies determine the order of evaluation of a function. And in a parallel implementation they show the degree of parallelism that lies in a program.

Up to now one of the main objectives against non-strict functional languages has been their lack of efficiency. On sequential implementations, they are several factors slower than strict programming languages. On the other side there are promising attempts to develop parallel implementations of non-strict functional languages that can exploit the additional parallelism of such languages. Now, the approach of circular programs offers the possibility of speeding up programs even in the sequential case without destroying parallelism. So, both approaches together may give rise to really efficient programs written in non-strict functional languages.

Appendix

A The Functional Programming Language RUFL

A.1 An Introduction to RUFL

In this section the main features of the functional programming language RUFL (Rhodes University Functional Language) will be sketched. Overall, RUFL bears a strong resemblance to the better known functional language Miranda. Therefore, especially those features that differ from Miranda will be mentioned here. A detailed description of RUFL is given in [Wentworth, 1989] by the author of the original RUFL compiler himself.

The most important features of RUFL are:

- For each function on the outermost level the type must be specified explicitly. However, as an exception to this rule the functions that are defined locally needn't get an explicit type declarations. A type declaration begins with the keyword **dec**. The type constructors are the same as for Miranda (i.e. [] for list types, $->$ for function types and (,) for tuple types). Contrary to Miranda type-variables (for allowing polymorphism) have to be declared explicitly. But, the variables `alpha`, `beta` and `gamma` are already declared as type-variables in the special module `Library`. So, these variables can be used as type-variables if they are imported with the command **import** `Library`.
- The top-level-function of a RUFL program must have the special name `main`. It mustn't have an explicit type declaration.

- RUFL provides an off-side rule like Miranda. This rules allows it to replace the grouping symbols '{' and '}' and the seperator ';' by indentation.
- Patterns, guards and higher-order functions can be used like in Miranda.
- Input/Output is realized via side-effects. For example `writelnInt x` prints the integer value `x` with a line feed. `readInt` reads an integer from the input. This integer is the result of calling `readInt`.

  Especially for input/output there is a **seq** { $\texttt{expr}_1$; ...; $\texttt{expr}_k$ } construct that forces the expressions $\texttt{expr}_1$; ...; $\texttt{expr}_k$ to be evaluated one after another. The result of the last expression is the result of the whole construct.
- RUFL programs can be split into several modules. In order to get independent modules one has to split one module into two parts:
  - the declaration part, containing the type declaration of all exported functions and the declaration of new data types,
  - the implementation part, containing the definitions of all functions.

  However, the current module-system has some disadvantages:
  - Type information can't be hidden. That means that new data types, that are used by exported functions must be put in the declaration part which is visible from the outside.
  - All names are global. Therefore, name clashes between different modules might occur.
  - Modules can't be nested.
- The declaration of a new data type is preceeded by the keyword **data**. The declaration of a type synonym is preceeded by the keyword **type**. The usage of both is the same as in Miranda.
- RUFL allows the usage of list comprehensions with the usual notation, but it doesn't allow array comprehensions.
- In RUFL anonymous functions can be defined by using λ-expressions. The syntax of a λ-expression in RUFL is as follows: **fn** `var=expr` means $\lambda var\ .\ expr$.

A.2 Some Predefined Functions

In this section the declarations and definitions of some functions are given that are used in the paper. Most of them are taken from the `Library` module of RUFL.

```
                                  {- fixity declaration:          -}
infixl 200  !                     {-  ! is left-associative       -}
                                  {-  infix op. with priority 200 -}

dec map  :: (alpha -> beta) -> [alpha] => [beta]
```

```
def map f []     = []
    map f (a:u) = f a : map f u


dec fst :: (alpha,beta) -> alpha

def fst (x,y) = x


dec snd :: (alpha,beta) -> beta

def snd (x,y) = y


dec nth :: Int -> [alpha] => alpha          {- predefined in Library -}

def nth i (a:u) = a ,           if i==1
                = nth (i-1) u , otherwise

dec (!) :: [alpha] -> Int -> alpha

def xs ! n = nth (n+1) xs                {- ! starts counting with 0 -}
```

B Partitioning Programs

In this section the programs are presented that were used for comparing the straightforward partitioning algorithm with the circular partitioning algorithm. The core of both algorithms was already discussed in Section 4.2. In order to make the test of the runtimes of both programs easier only the number of partitions for each integer is printed. Therefore, in an additional list comprehension the number of partitions for each integer is stored.

B.1 Problem Specification of Partitioning

Input: n ...A natural number.
Output: l ...A list of all partitions of n, where
x is a partition of l iff
x is a list of positive natural numbers $x_1, \ldots, x_k$ such that
$x_1 + \ldots + x_k = n \ \wedge \ x_1 \leq x_2 \ldots \ \wedge \ x_{k-1} \leq x_k$

B.2 Straightforward Program

```
--                                  Hans Wolfgang Loidl, 28.3.1991
--
```

```
--  Straightforward program for computing a list of all partitions
--  for all integers i.e. [p_0, p_1, p_2, ...] where p_i is a list of
--  partitions of i due to the problem specification below.
--
-- Problem Specification of partition:
--
--  Given: n, n ... Integer.
--  Find:  All x_1,...,x_k s.t.
--             x_1,...,x_k are Integers > 0  and
--             x_1+...+x_k=n and
--             x_1 <= x_2 <= ... <= x_k.


import Library

type Rep = [Int]

infixr 300   +++

dec allpart  :: [[Rep]]
dec testCons :: alpha -> [alpha] -> [alpha]
dec (+++)    :: [[alpha]] -> [[alpha]] -> [[alpha]]


def allpart = [part' k | k <- [0..] ]
              where
               part' 0 = [[]]
               part' 1 = [[1]]
               part' k =  reduceR (+++) []
                             [map (testCons i) (part' (k-i))
                             | i <- [1..k] ]

def testCons x [] = [x]
    testCons x ys = x : ys ,  if x <= (hd ys)
                  =     [] ,  otherwise

def []     +++ ys = ys
    (x:xs) +++ ys =     (xs +++ ys)  , if x==[]
                  = x : (xs +++ ys)  , otherwise

def main = seq
              writelnString "List of numbers of solutions of n<-[0..]"
              writelnInts [(length sol) | sol <- allpart]
```

B.3 Circular Program

In the following program *allcpart* computes the list of partitions of all natural numbers. As in the straightforward program only the numbers of partitions for each natural number are printed. The function *cpart* computes a list of all partitions of a given number n. This function is not used in the main program at all. It is only presented to make the idea of the algorithm, which is also used in *allcpart*, clearer.

```
--                                        Hans Wolfgang Loidl, 28.3.1991
--
--  CIRCULAR program for computing a list of all partitions
--  for all integers i.e. [p_0, p_1, p_2, ...] where p_i is a list of
--  partitions of i due to the problem specification below.
--
-- Problem Specification of partition:
--
--  Given: n, n ... Integer.
--  Find:  All x_1,...,x_k s.t.
--             x_1,...,x_k are Integers > 0  and
--             x_1+...+x_k=n and
--             x_1 <= x_2 <= ... <= x_k.


import Library

type Rep = [Int]

infixr 300   +++

dec allcpart :: [[Rep]]
dec cpart    :: Int -> [Rep]
dec testCons :: alpha -> [alpha] -> [alpha]
dec (+++)    :: [[alpha]] -> [[alpha]] -> [[alpha]]


def allcpart = p
             where p =  [[]] : [[1]] :
                        [reduceR (+++) []
                          [map (testCons i) (nth (k-i+1) p)
                          | i <- [1..k] ]
                        | k <- [2..] ]

def testCons x [] = [x]
    testCons x ys = x : ys ,  if x <= (hd ys)
                  =     [] ,  otherwise

def []     +++ ys = ys
```

```
    (x:xs) +++ ys =      (xs +++ ys)   ; if x==[]
                  = x : (xs +++ ys)   ; otherwise

def cpart n = nth (n+1) p                              -- not necessary
            where p =  [[]] : [[1]] :                  -- just for idea
                       [reduceR (+++) []
                         [map (testCons i) (nth (k-i+1) p)
                         | i <- [1..k] ]
                       | k <- [2..n] ]

def main = seq
             --   Partitions for all natural numbers
             writelnString "List of numbers of solutions of n<-[0..]"
             writelnInts [(length sol) | sol <- allcpart]
```

C More Circular Programs

In this section some circular programs are presented that have not been discussed in the paper. These programs don't reveal new details of the circular programming technique but they

- show that the circular programming technique can be applied on a large number of problems,
- may help the reader to apply the circular programming technique for his own problems.

C.1 The General Hamming Problem

In [Wentworth, 1989][Page87] a circular program for the so called Hamming Problem is given. This problem can be stated as follows:

> Generate the strictly ascending sequence of all numbers such that:
>
> 1. 1 is in the sequence.
> 2. If x is in the sequence, so are $2x, 3x$ and $5x$.
> 3. No other values are in the sequence.

Obviously, one can generalize this problem by replacing 1 by a list of initial values and by replacing the functions $\lambda x.2x, \lambda x.3x$ and $\lambda x.5x$ by a list of non-decreasing functions from integer to integer. Now, this generalized Hamming Problem is solved by the following circular program:

```
--                                          Hans Wolfgang Loidl, 9.5.1991
--
-- CIRCULAR program for the General Hamming Problem (Closure Problem):
--
-- Given: xs, fs    xs ... a sorted list of initial values
--                  fs ... a list of non-decreasing fcts
--                         (i.e. for all f in fs :
--                                 for all x : (f x) >= x )
-- Find:  ys        s.t.
--                   for all x in xs : x in ys   AND
--                   for all x : x in ys =>
--                      (for all f in fs : (f x) in ys) AND
--                   ys is sorted

import HwlLib

dec hamming :: [alpha] -> [alpha -> alpha] -> [alpha]
dec merge_n :: [[alpha]] -> [alpha]

 {- merge_n performs a sorted merge on n lists by folding performing a
    dual merge right associative to each of the lists                -}

def hamming xs fs = ham
                    where ham = (hd xs) :
                                (tl xs) +++
                                (merge_n [[f x | x <- ham] | f <- fs])

  {- Since each fct is non-decreasing the first element of
     the sorted list of initial values must be the first
     element of the resulting list. This value has to be
     put into the resulting list immediately so that the same
     list can be used in the list comprehension (otherwise
     the program would deadlock) -}

def merge_n xss  = reduceR (+++) [] xss

def main =
        let
          l = [[fix (pow (float x) (float i))
               | i <- [1..5]] | x <- [2,3,5]]

          {- Conversion functions:
              float :: Int  -> Real
              fix   :: Real -> Int
             pow :: Real -> Real -> Real
             pow x y yields x^y          -}
```

```
    xs1 = [1,3..9]
    fs1 = [f where { f = ((+) 2) }]

    xs2 = [1]
    fs2 = [((*) i) | i <- [2,3,5]]

    {- % ... modulo function
       all p xs yields TRUE iff (p x) is TRUE for all x in xs
    -}

    isPrime x    = all (fn y = x % y != 0)
                       (2:[3,5..(fix (floor (sqrt (float x))))])
    nextPrime p = until isPrime ((+) 2) (p+2)
    xs3 = [3]
    fs3 = [nextPrime]
  in
    seq
      writelnString "1) Result should be all odd numbers <=29:"
      writelnInts (takewhile ((>=) 29) (hamming xs1 fs1))

      writelnString "2) Original Hamming Problem up to 50 "
      writelnString "   (Init.:[1] "
      writelnString "   Fcts.:[((*) 2), ((*) 3), ((*) 5)]): "
      writelnInts (takewhile ((>=) 50) (hamming xs2 fs2))

      writelnString "3) Result should be odd primes up to 50 "
      writelnInts (takewhile ((>=) 50) (hamming xs3 fs3))
```

C.2 The Transitive Closure

Also the well known problem of determining the transitive closure of a given relation can be solved with a circular program. However, in this case the circularity is not quite so obvious as in the previous examples.

The first idea for a circular program might be the following: If we want to compute the transitive closure of one single element x we get all the elements $y_1, \ldots, y_n$ that can be reached from x in one step and then we add to each y_i the transitive closure of y_i itself. But it is quite clear that such a program would run into an infinite recursion if the input relation has cycles in it. Therefore, we can't just take the described recursive algorithm and transform it into an circular program.

The main idea for a correct circular program is to associate with each element x of the relation a function that takes a list of all the elements that are already in the transitive closure as an argument and adds all those successors of x to this list that aren't members of the list at that time. From these new elements we must recursively compute the transitive closure. So here we have again a recursive part that can be transformed into a circular program in the usual way.

The whole program is split into two steps that can be seen in the where part of the definition of *trans*:

1. In the first step for each element of the given relation a function that will compute the transitive closure is produced. The parameter of this function contains all elements that are already contained in the transitive closure. The result of this step is a list of tuples consisting of an element and the respective function that computes the transitive closure.

2. In the second step all the functions have to be called at the beginning with the empty list as parameter, since at the beginning no element is contained in the transitive closure. The result of each of the function calls is the transitive closure for each of the elements.

When in the second step the functions are called each function adds the new elements (that can be reached in one step) to the transitive closure and also adds the transitive closures of the new elements. The transitive closures of these are computed by calling the stored function with xs".

Although the gain of efficiency is not so big as in the previous examples the efficiency has indeed improved. The computation of xs' and xs" is done in step one and therefore only once for each function. Each of the functions will then be called several times with these already computed xs' and xs". But since the list of elements that has already been added to the transitive closure changes in each call no further straightforward optimization is possible.

```
--                                          Hans Wolfgang Loidl, 9.5.1991
--
-- CIRCULAR program for computing the transitive closure of a given
-- relation.


import Library

infixr  300  +++

type Object     = Int
type Dependence = (Object, [Object])
type ContDependence = (Object, [Object] -> [Object])
type Relation   = [Dependence]

dec trans  :: Relation -> Relation
dec g      :: [ContDependence] -> Dependence -> ContDependence
dec lookup :: [ContDependence] -> Object -> ([Object] -> [Object])
dec (+++)  :: [Object] -> [Object] -> [Object]
dec writeRel :: Relation -> Relation
```

```
dec applyAll :: [alpha -> beta] -> alpha -> [beta]
dec feed     :: ContDependence -> Dependence

-- p :: [ContDependence] i.e. p :: [(Object,[Object] -> [Object])]

def trans lt = p'
               where p = map (g p) lt      -- 1. create [(elem., fct)]
                     p'= map feed p        -- 2. evaluate above fcts

def g p (x,[]) = (x, fn x = [])
    g p (x,xs) = (x, fn { l = xs' +++ (reduceR (+++) []
                               (applyAll (map (lookup p) xs') xs''))
                               where  xs' = xs \\ l
                                      xs'' = l +++ xs'   } )
 {- Meaning of variables:
     (x,xs) ... element x and all elements that can be reached from x
                in one step.
     xs' ... elements that are in xs but not in the transitive
             closure so far.
     xs'' ... elements that are in the transitive closure and
              elements that can be reached from x in one step
              (xs union l).                                          -}


def lookup []          x = fn x = []
    lookup ((x',ls):p) x = ls          , if x==x'
                         = lookup p x  , otherwise

def [] +++ ys = ys
    xs +++ [] = xs
    (x:xs) +++ (y:ys) = x : (xs +++ ys)       , if x==y
                      = x : (xs +++ (y:ys))   , if x < y
                      = y : ((x:xs) +++ ys)   , otherwise

def applyAll []     x = []
    applyAll [f]    x = [f x]
    applyAll (f:fs) x = (f x) : (applyAll fs x)

def feed (x,f) = (x,f [])

def writeRel []          = []
    writeRel ((x,xs):ys ) = seq
                            writeInt x
                            writeString " : "
                            writeInInts xs
                            writeRel ys

def main =
```

```
let
  r1 = [(1,[3,5]),(2,[4]),(3,[3]),(4,[2,6]),(5,[1,9]),
        (6,[2,6,8]),(7,[]),(8,[2,4]),(9,[1,3,7])]
  r2 = [(1,[2,9]),(2,[1,2]),(3,[4]),(4,[5]),(5,[3,10]),
        (6,[6]),(7,[2,9]),(8,[3,9,12]),(9,[9,10]),
        (10,[4]),(11,[6,12]),(12,[7])]
in
  seq
    writelnString "Given: "
    writeRel r1
    writeString "Result should be 2 sets:"
    writelnString "even and odd nr.<10"
    writelnString "Exceptions: (3,[3]) and (7,[])"
    writeRel (trans r1)

    writelnString "Given: "
    writeRel r2
    writelnString "Transitive Closure: "
    writeRel (trans r2)
```

REFERENCES

[Abelson and Sussman, 1985]
Harold Abelson and Gerald Jay Sussman.
Structure and Interpretation of Computer Programs.
The MIT Electrical Engineering and Computer Science Series. The MIT Press, 1985.

[Allison, 1989]
Lloyd Allison.
Circular Programs and Self-referential Structures.
Software — Practice and Experience, 19(2):99–109, 1989.

[Augustsson, 1987]
Lennart Augustsson.
Compiling Lazy Functional Languages, Part II.
PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Bird, 1984]
R. S. Bird.
Using Circular Programs to Eliminate Multiple Traversals of Data.
Acta Informatica, 21:239–250, 1984.

[Burn *et al.*, 1986]
Geoffrey L. Burn, Chris Hankin, and Samson Abramsky.
Strictness Analysis for Higher-Order Functions.
Science of Computer Programming, 7:249–278, 1986.

[Burn, 1991]
Geoffrey Burn.
Lazy Functional Languages: Abstract Interpretation and Compilation.
Research Monographs in Parallel and Distributed Computing. Pitman, London, 1991.

[Clack and Peyton Jones, 1985]
Chris Clack and Simon L. Peyton Jones.
Strictness Analysis — A Practical Approach.
In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 35–49, Nancy, France, September 16–19, 1985.
Volume 201 of Lecture Notes in Computer Science, Springer, Berlin.

[Field and Harrison, 1988]
Anthony J. Field and Peter G. Harrison.
Functional Programming.
International Computer Science Series. Addison-Wesley, Wokingham, England, 1988.

[Henderson, 1980]
Peter Henderson.
Functional Programming.
International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[Hughes, 1985]
John Hughes.
Lazy Memo-Functions.
In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 129–146, Nancy, France, September 1985.

[Johnsson, 1987]
Thomas Johnsson.
Compiling Lazy Functional Languages, Part I.
PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Kelly, 1989]
Paul Kelly.
Functional Programming for Loosely-coupled Multiprocessors.
Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[Loogen, 1990]
Rita Loogen.
Parallele Implementierung funktionaler Programmiersprachen (Parallel Implementation of Functional Programming Languages) (in German), volume 232 of *Informatik-Fachberichte.*
Springer, Berlin, 1990.

[Peyton Jones, 1991]
Simon L. Peyton Jones.
The Spineless Tagless G-machine: second attempt.

Technical report, Department of Computing Science, University of Glasgow, June 1991.

[Schreiner, 1990]
Wolfgang Schreiner.
ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language.
Master's thesis, Johannes Kepler University, Linz, Austria, October 1990.
Also: Technical Report 90-42.0, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990. Also: Technical Report 91-1, Austrian Center for Parallel Computation, January 1991.

[Sijtsma, 1988]
B.A. Sijtsma.
Verification and Derivation of Infinite-List Programs.
PhD thesis, Rijksuniversiteit Groningen, October 1988.

[Sijtsma, 1989]
Ben A. Sijtsma.
On the Productivity of Recursive List Definitions.
ACM Transactions on Programming Languages and Systems, 11(4):633–649, October 1989.

[Wentworth, 1989]
E. P. Wentworth.
Introduction to Functional Programming using RUFL.
Technical Document PPG 89/6, Parallel Processing Group, Department of Computer Science, Rhodes University, Grahamstown 6140, South Africa, May 1989.

[Wray and Fairbairn, 1989]
S. C. Wray and J. Fairbairn.
Non-strict Languages — Programming and Implementation.
The Computer Journal, 32(2):142–151, 1989.