Listlessness is Better than Laziness II: Composing Listless Functions

Philip Wadler
Programming Research Group, Oxford University
11 Keble Road, Oxford, OX1 3QD

1. Introduction

This paper is a successor to "Listlessness is Better than Laziness" [Wadler 84a,b]. That work described a method that can automatically transform a program to improve its efficiency, by eliminating all intermediate lists.

The listless transformer applies only to programs from which all intermediate lists can be removed. In [Wadler 84b] this is made precise by defining the class of programs that are subject to bounded evaluation (b.e. programs), that is, programs that can be lazily evaluated in constant bounded space. Unfortunately, the listless transformer only acts as a semi-decision procedure for determining whether a given program is b.e.: it will succeed if the given program is b.e., but may enter an infinite loop if the program is not b.e.

Clearly, it would be desirable to have a simple way of structuring programs which guarantees that they are listless. An obvious rule that comes to mind is *composition*. That is, one would like to be able to say: if `f` and `g` are functions computed by listless programs, then so is `(f ∘ g)`. (As usual, `(f ∘ g) x = f (g x)`.) Unfortunately, this is not true; a counterexample will be given in section 2. However, there is a simple restriction of listless programs, namely *preorder listless programs*, for which this property does hold. Thus, if `f` and `g` are functions computed by preorder listless programs, then so is `(f ∘ g)`.

Further, this paper will present an algorithm that given the preorder listless programs that compute `f` and `g` will return a preorder listless program that computes `(f ∘ g)`. This means that one can transform an expression of the form `(f ∘ g)` in a modular way: first transform `f` and `g`, and then combine the results. This is in contrast to the previous listless transformer, which would need to transform `(f ∘ g)` in a single (possibly large) operation. The advantages of performing the transformation in this way are similar to the advantages of independent compilation as opposed to a single monolithic compilation.

Of course, not all programs are listless. For example, there is no listless program to sort or reverse a list, since in these cases the entire input list must be read (and saved) before the first element of the output list can be written. On the other hand, the function to merge two sorted lists is listless.

In general, few useful programs are listless as a whole, although most programs contain large sections which are listless. This suggests that listlessness will be useful in practice only if one can find a way to combine listless programs with other programs that are not listless. This paper discusses a method for combining general programs evaluated by a reduction machine with listless programs evaluated by a listless machine. With this method, the use of listlessness in an optimizing compiler appears to be a practical possibility.

A method similar to that described in this paper is discussed in [Ganzinger and Giegrich 85], which deals with the composition of attribute grammars.

This paper is organized as follows. Section 2 explains the difference between listless programs and preorder listless programs. Section 3 demonstrates that the composition of two preorder listless programs is a preorder listless program. Section 4 gives a formal description of listless programs. Section 5 describes an algorithm that composes two preorder listless programs to obtain a new listless program. Sections 6 and 7 extend this algorithm to more general forms of composition and to listless programs that include primitives. Section 8 discusses schemas to provide listless programs for library functions. Sections 9 and 10 discuss how listless programs can be interfaced to a general graph reducer. Section 11 discusses a detail related to strong correctness and gives a semantic characterization of preorder traversal. Section 12 presents conclusions. Appendix 1 describes some standard functions used in this paper, and appendix 2 gives definitions of these functions.

2. The importance of being preorder

This section presents a counterexample, to show that the composition of two listless programs is not necessarily listless. Consider the following two function definitions:

```
join (pair xs ys)  =  append xs ys
copy zs            =  pair zs zs
```

Here `pair` is a constructor of arity two (different from the constructor cons) and `append` concatenates two lists. Both `join` and `copy` are listless, that is, can be computed by listless programs. Informally, `join` is computed by first traversing the list `xs`, copying each element to the output, and next traversing `ys`, copying each element to the output; and `copy` is computed by traversing `zs`, copying each element to both lists in the output pair.

The composition of these two functions is

```
(join ∘ copy) zs    =  append zs zs
```

But appending a list to itself is not listless, that is, cannot be computed by a listless program. The only way to compute `append zs zs` without using intermediate lists requires traversing `zs` twice, and this violates the definition of listless program, which requires that each input list is traversed only once.

The problem with composing `join` and `copy` is that `copy` produces its output in a different order from that in which `join` traverses its input. That is, `copy` first creates a pair node; it then provides the first element of both lists in the pair, then the second element, and so on. On the other hand, `join` first examines the input pair node; it then completely traverses the first list in the pair before examining the second list.

This problem arises because the definition of listless program says nothing about the order in which the input or output data structures are traversed. This motivates the concept of preorder listless program. In a preorder listless program, the input and output data structures must each be traversed in preorder from left to right. For example, `copy` is not preorder listless because it produces the two result lists together, rather than completely producing the first result list from left to right, and then producing the second result list. On the other hand, `join` is preorder listless because it traverses the first input list from left to right, and then traverses the second input list; and the output list is also produced from left to right.

Most useful listless programs are in fact preorder. For example, the term

```
        sum (map square (upto 1 n))
```

finds the sum of the squares of the numbers from 1 to `n`, and `sum`, `map square`, and `upto` are all preorder listless. (See the appendix for definitions of `sum`, `map`, and `upto`.) Indeed, the idea that inputs and outputs should be traversed in order is so natural that many people on encountering listless programs make the (mistaken) assumption that they are the same as preorder listless programs.

There is one useful class of functions that are listless but not preorder listless. These are the functions, such as `copy`, that return more than one list. In this case each output list is traversed in order, but the lists must be bundled into a structure, such as the `pair` in `copy`, which is not traversed in order. Another example of such a function is:

```
        split p xs   =  pair (filter p xs) (filter (not ∘ p) xs)
```

which given a predicate `p` and a list `xs` returns a pair of lists, one containing elements in `xs` with property `p` and the other containing elements in `xs` not satisfying `p`. Although each instance of `filter` is preorder listless, the combination of the two into a `pair` is not. This paper will not deal with such functions, but extending the work described here to handle such cases is an obvious area for future research. It seems clear that such an extension is possible.

3. Composing preorder listless functions

The composition of two preorder listless functions is also preorder listless. This may be intuitively obvious, but the proof can be made more explicit as follows. Consider the preorder listless programs that compute `f` and `g`. To compute `f ∘ g` one simply runs the program for `f` (possibly producing some output) until it requests some input; one then runs the program for `g` (possibly traversing some input) until it produces some output. Since f traverses its input in preorder, and g produces its output in preorder, the input required by f must be the same as the output produced by g, so one may now continue running f. This process is repeated until f and g finish execution.

In other words, f and g can be run as coroutines. No "buffering" is needed between f and g, because f requests input in the same order that output is produced by g. Thus, if f and g can be computed in space bounded by the constants C and C′ respectively, then f ∘ g can be computed in space bounded by C + C′. Or, to put it another way, if the programs for f and g have N and N′ different states respectively, then f ∘ g can be computed by a program with N × N′ different states. (In general, if a listless program has N different states, then its space is bounded by a constant C proportional to log N.)

This shows that f ∘ g is listless. Further, since f produces its output in preorder and g consumes its input in preorder, it follows that f ∘ g is also preorder listless.

An example of the composition of preorder listless functions will be given in the next section.

4. Listless programs

This section re-introduces listless programs, with some changes in notation from the previous work. The syntax of listless programs is as follows:

```
listless program   q  ::=   done                        termination
                          | v ← p ; q                   output step
                          | case v of {pi ⇒ qi}         input step

pattern            p  ::=   c v1 ... vk                 (where c has arity k)
```

Here c is a constructor and v is an input or output variable. The notation (case v of {pi ⇒ qi}) is a shorthand for:

```
case v of
    p1 ⇒ q1
    ...
    pn ⇒ qn
```

Listless programs are graphs, but the above notation treats them as if they were (possibly infinite) trees. In practice, listless programs can be represented in the computer by cyclic structures, which represent the graph directly in the computer memory.

For example, here are the definitions of three functions:

```
nots        =  map not
every       =  rightreduce and true
everynot    =  every ∘ nots
```

The function nots takes a list of booleans, and returns a list with each element in the input replaced by its complement. The function every takes a list of booleans, and returns true if every element in the input list is true. (The functions map, rightreduce, not, and and are defined in the appendix.) The functions nots and every are preorder listless, and hence everynot is also preorder listless.

The three listless programs corresponding to the above definitions are given in figure 1. Here x, xs, z, zs are input variables; y, ys, a are output variables; true, false, nil are constructors of arity zero; and cons is a constructor of arity two. Labels such as NOTS are used to indicate cycles in the listless programs. This is just a notational convenience; as mentioned, the programs can be represented in the computer directly by cyclic structures.

The programs in figure 1 can, of course, be derived by applying the listless transformer. In addition, the program for everynot can also be derived by applying the method of composition, described in section 5, to the programs for nots and every.

Incidentally, if we define

```
some        =  rightreduce or false
notsome     =  not ∘ some
```

and then generate the listless program corresponding to notsome (either directly or by the composition method), then this program will be identical to the program for everynot given in figure 1. This amounts to an automatic proof of a generalized version of DeMorgan's law.

A description of the listless transformer is beyond the scope of this paper; it can be found in [Wadler 84b]. It is easy to modify the transformer described there so that it is restricted to traverse the input and output in preorder. Further, as a result of this restriction, there is only one possible candidate for the input variable at each input step. This means that the expensive breadth-first search of input variables discussed in [Wadler 84b] is not required for this version of the listless transformer.

As mentioned in the introduction, one can define a concept of bounded evaluation (b.e.) that corresponds to a program that can be evaluated in bounded space, not counting space occupied by its input or output. In [Wadler 84b] it is shown that the listless transformer succeeds in converting a functional program to an equivalent listless program if and only if the functional program is b.e. Similarly, one can define a concept of preorder bounded evaluation (p.b.e.) that corresponds to a program that can be evaluated in bounded space, not counting space occupied by its input or output, and traverses its input and output in preorder. One can show that the modified listless transformer succeeds in converting a functional program to an equivalent preorder listless program if and only if the functional program is p.b.e. The precise definition of p.b.e. and the proof are straightforward modifications of the work in [Wadler 84b].

Figure 1: *Listless programs for* nots, every, *and* everynot.

```
{ys ← nots xs}

    NOTS:      case xs of
                   nil         ⇒  ys ← nil;  done
                   cons x xs   ⇒
                       ys ← cons y ys;
                       case x of
                           true    ⇒  y ← false;  NOTS
                           false   ⇒  y ← true;   NOTS


{a ← every zs}

    EVERY:     case zs of
                   nil         ⇒  a ← true;  done
                   cons z zs   ⇒
                       case z of
                           true    ⇒  EVERY
                           false   ⇒  a ← false;  done


{a ← everynot xs}

    EVERYNOT:  case xs of
                   nil         ⇒  a ← true;  done
                   cons x xs   ⇒
                       case x of
                           true    ⇒  a ← false;  done
                           false   ⇒  EVERYNOT
```

5. The composition algorithm

Section 3 showed how two listless programs might be composed by executing them as coroutines. This section will show how two listless programs can be combined directly into a single program. The advantage of doing this, as opposed to simply running the two programs as coroutines, is that the overhead associated with the coroutine linkage is eliminated.

If `q` and `q'` are two listless programs corresponding to `g` and `f`, we write `q >> q'` for the listless program corresponding to `f ∘ g`. That is, `q >> q'` is the program that results from connecting the output of `q` to the input of `q'`. For example, the program `EVERYNOT` in figure 1 can be derived by computing `NOTS >> EVERY`.

Algorithm 1: *Simple Listless Composition.* Given `q` and `q'` one may derive a listless program to compute `q >> q'` by applying the following rules.

If `q'` is done, then so is `q >> q'`:

```
(1)    q >> done   =   done
```

If `q'` is about to perform an output step, then so should `q >> q'`:

```
(2)    q >> (v' ← p' ; q')   =   v' ← p' ; (q >> q')
```

If `q` is about to perform an input step, then so should `q >> q'`:

```
(3)    (case v of {pi ⇒ qi}) >> q'   =   case v of {pi ⇒ (qi >> q')}
```

If `q` is about to perform an output step and `q'` is about to perform an input step, then the output from `q` may be used to select which path is taken in `q'`:

```
(4)    (v ← p; q) >> (case v' of {pi' ⇒ qi'})   =   q >> qj'
```

where `p` has the same constructor as `pj'`

In rule (4) one knows that the output variable `v` of `q` must correspond to the input variable `v'` of `q'`, because outputs of `q` and inputs of `q'` happen in the same order. □

These rules must be applied in the order given; e.g., if both rule (1) and rule (3) apply then rule (1) should be chosen. This corresponds to the use of lazy evaluation: the only actions of `q` that should be performed are those that are needed by `q'`.

The above description says nothing explicitly about loops in the listless program. This is handled implicitly by "memoizing" the corresponding (functional) program that combines the structures representing two listless programs. That is, in order to compute `q >> q'` we maintain a table containing entries of the form `((q, q'), q >> q')`. To compute `q >> q'` we first check the table. If there is an appropriate entry it is returned. Otherwise, q >> q′ is computed using rules (1) -- (4) above, and then a new entry is made in the table. If lazy evaluation is used then computation of q >> q′ can be guaranteed to terminate, and the structure returned will contain appropriate cycles. For a further discussion of memoization see [Hughes 85a]. A similar technique is also used in [Mont-Reynaud 76].

Algorithm 1, together with algorithms 2 and 3 below, has been implemented as a (non-listless) functional program. The programming language used was Orwell [Wadler 85a], augmented with lazy memo functions [Hughes 85a]. It is pleasing to see that the implemented program looks almost identical to rules (1) -- (4) above. The resemblance is particularly close because Orwell allows the user to define new infix symbols (such as >>, ←, and ;).

6. Generalised composition

Section 3 presented an argument to show that if f and g are preorder listless functions then so is f ∘ g. The same argument can be modified to show a more general result, namely, if f and g are preorder listless functions of arity m and n respectively, then h is an preorder listless function of arity m+n−1, where

```
(*)       h x1 ... x(i-1) y1 ... yn x(i+1) ... xm
                    =  f x1 ... x(i-1) (g y1 ... yn) x(i+1)... xm
```

and x1, ..., xm, y1, ..., yn are all distinct variable names. The argument is essentially the same as before: run the listless programs for f and g as coroutines. The program for f is run until it requires an input. If the input is some xj with i ≠ j then the input occurs directly and the program continues. If the input is xi then the program for g is run until it produces an output. This works because the input variables are all distinct, and therefore do not interact with each other.

For example, consider the function

```
        appendnots xs ys  =  append xs (nots ys)
```

Since append and nots are preorder listless functions of arity two and one respectively, it follows from the above result that appendnots is also preorder listless.

It is not hard to modify the algorithm of section 5 to handle this more general version of function composition. The >> notation must be modified to include some extra bookkeeping, as follows. Let q and q′ be two preorder listless programs, each with an arbitrary number of input and output variables. If necessary, rename the variables of q so they are distinct from the variables of q′. Let V be a set of pairs of the form (v, v′) where v is an output variable of q and v′ is an input variable of q′. Then q >>(V) q′ denotes the listless program that results from connecting each output v of q to the corresponding input v′ of q′ for each (v, v′) in V. The inputs of this program are the inputs of q plus the inputs of q′ not in V, and the outputs of this program are the outputs of q′. (Initially, V contains a single pair, but it may contain more than one pair during execution of the algorithm below.)

For example, consider the listless programs APPEND and NOTS with the following specifications:

```
APPEND      =  {xs ← append xs1 xs2}
NOTS        =  {ys ← nots ys1}
```

Let V = {(ys, xs2)}. Then NOTS >>(V) APPEND is equivalent to the listless program

```
APPENDNOTS  =  {xs ← append xs1 (nots ys1)}
```

More generally, let F and G be listless programs with the following specifications:

```
F  =  {x ← f x1 ... xm}
G  =  {y ← g y1 ... ym}
```

Then the function h defined in (*) above is computed by the listless program G >>(V) F, where V = {(y, xi)}. This program can be computed by the following algorithm.

Algorithm 2: *Generalized Listless Composition.* Given q, q′, and V as above, where q and q′ have no names in common, the listless program that computes q >>(V) q′ can be derived by the following rules. As with algorithm 1, the rules should be applied in order, and memoization (on (q, q′, V)) should be used to handle loops in the listless programs.

If q′ is done, then so is q >>(V) q′:

```
(1)    q >>(V) done    =    done
```

If q′ is about to perform an output step, then so should q >>(V) q′:

```
(2a)   q >>(V) (v′ ← p′ ; q′)    =    v′ ← p′ ; (q >>(V) q′)
```

If q′ is about to perform an input step on a variable not in V, then so should q >>(V) q′:

```
(2b)   q >>(V) (case v′ of {pi′ ⇒ qi′})
                     =    case v′ of {pi′ ⇒ (q >>(V) qi′)}

              if (v, v′) ∉ V  for any v
```

If q is about to perform an input step, then so should q >>(V) q′:

```
(3a)  (case v of {pi ⇒ qi}) >>(V) q′   =   case v of {pi ⇒ (qi >>(V) q′)}
```

If q is about to perform an output step and q′ is about to perform an input step on the corresponding variable in V, then the output from q may be used to select which path is taken in q′:

```
(4a)  (v ← p; q) >>(V) (case v′ of {pi′ ⇒ qi′})   =   q >>(V′) qj′

            if (v,v′) ∈ V
            where  p has the same constructor as pj′
              and  c v1  ... vk   =  p
              and  c v1′ ... vk′  =  pj′
              and  V′  =  V - {(v,v′)} ∪ {(v1,v1′), ..., (vk,vk′)}
```

(For example, if q performs the output step (xs ← cons x xs; qq) and q′ performs the input step (case ys of ... cons y ys ⇒ qq′) and V is {(xs, ys)} then q >>(V) q′ is equivalent to qq >>(V′) qq′ where V′ is {(x, y), (xs, ys)}.) This rule is valid because all output and input variables are traversed in pre-order, so the output steps on v1, ..., vk in q will take place in that order, as will the input steps on v1′, ..., vk′ in q′. □

7. Primitive operations

In order to be practical, listless programs must be extended to include operations on primitive data, such as integers. This section briefly describes such an extension, and presents corresponding extensions to algorithm 2.

Listless machines are extended to include a new class of variables, primitive variables, in addition to input and output variables. Four new kinds of step are added to listless programs, as follows:

```
listless program    q  ::=   ...
                           | u ← v ; q                 primitive input
                           | u ← t ; q                 primitive operation step
                           | v ← u ; q                 primitive output step
                           | case u of {pi ⇒ qi}       primitive case step

primitive term      t  ::=   u | f t1 ... tk           (where f has arity k)
```

Here f is a primitive function and u is a primitive variable. A primitive input step copies data from an input variable to a primitive variable, and a primitive output step copies data from a primitive variable to an output variable. A primitive operation step performs some computation on primitive variables (for example, adding two variables together) and puts the result back into a primitive variable. A primitive case step branches on the contents of a primitive variable; in the primitive case step, each pi must be a constructor ci of arity zero (for example, true and false).

For example, consider the function:

```
sumsq n  =  sum (map square (upto 1 n))
```

which finds the sum of the squares of the numbers from 1 to n. Figure 2 gives listless programs to compute sum, map square, upto, and sumsq. Algorithm 2 is extended to handle primitive steps in algorithm 3 below. The program SUMSQ in figure 2 can be derived by calculating

```
UPTO >>({(ws,xs)}) SQUARES >>({(ys,zs)}) SUM
```

using two applications of algorithm 3.

To define algorithm 3, it is convenient to treat several kinds of steps together. Input steps and primitive case steps are together called case steps, and are written in the form case w of {pi ⇒ qi}, where w is v or u. Similarly, the remaining kinds of steps (except done) are called assignment steps and written in the form w ← r, where w is v or u, and r is p, u, or t.

Algorithm 3: *Generalized Listless Composition with Primitives.* To find q >>(V) q′, where q and q′ may contain primitive steps, algorithm 2 may be modified as follows. Rules (1) and (4a) are left unchanged. Rules (2a), (2b), and (3a) are generalized to case steps and assignment steps by replacing v by w, and p by r, where appropriate. Finally, the following two rules are added.

If q is about to perform an assignment step on a variable not in V, then so should q >>(V) q′:

```
(3b)  (w ← r ; q) >>(V) q′   =   w ← r ; (q >>(V) q′)

           if  (w, v′) ∉ V  for any v′
```

If q is about to perform a primitive output step and q′ is about to perform a primitive input step on the corresponding variable in V, then the output value of q should be copied to q′ (by a trivial primitive computation step):

```
(4b)  (v ← u ; q) >>(V) (u′ ← v′ ; q′)   =   u′ ← u ; (q >>(V′) q′)

           if  (v, v′) ∈ V
           where  V′ = V - {(v, u′)}
```

As a further optimization, the trivial assignment u ← u′ can be eliminated by renaming all instances of u′ in q′ to u. (If this optimization is not performed, then the last line of SUMSQ in figure 2 should be: u3 ← u0; u4 ← u3 × u3; u6 ← u4; u5 ← u5 + u6; u0 ← u0 + 1; SUMSQ′.) □

Figure 2: *Listless programs for* upto, map square, sum, *and* sumsq.

```
{ws <- upto 1 n}

    UPTO:        u0 ← 1;  u1 ← n;  UPTO′
    UPTO′:       u2 ← (u0 ≤ u1);
                 case u2 of
                     false  ⇒  ws ← nil;  done
                     true   ⇒  ws ← cons w ws;
                               w ← u0;  u0 ← u0 + 1;  UPTO′

{ys <- map square xs}

    SQUARES:     case xs of
                     nil          ⇒  ys ← nil;  done
                     cons x xs    ⇒  ys ← cons y ys;
                                     u3 ← x;  u4 ← u3 × u3;
                                     y ← u4;  SQUARES

{a <- sum zs}

    SUM:         u5 ← 0;  SUM′
    SUM′:        case zs of
                     nil          ⇒  a ← u5;  done
                     cons z zs    ⇒  u6 ← z;  u5 ← u5 + u6;  SUM′

{a <- sumsq n}

    SUMSQ:       u0 ← 1;  u1 ← n;  u5 ← 0;  SUMSQ′
    SUMSQ′:      u2 ← (u0 ≤ u1);
                 case u2 of
                     false  ⇒  a ← u5;  done
                     true   ⇒  u4 ← u0 × u0;  u5 ← u5 + u4;
                               u0 ← u0 + 1;  SUMSQ′
```

8. Listless schemas

The preceding sections have developed a method for combining two listless programs to produce another listless program. But where do the initial listless programs come from?

Of course, they could be derived by applying the listless transformer to functions that the user declares to be listless. This will be useful in general for user-defined functions. In practice, however, this generality will often not be needed, because most of the listless functions in a program will be standard library functions, or instances of them.

For example, one would expect to find the functions every and some from section 4 in a standard library, and the library could also store the corresponding listless programs. Another listless library program is merge, which merges two sorted lists of numbers into a single list. (A listless program for every is given in figure 1, and a listless program for merge is discussed in [Wadler 84b].)

Of more general utility is the fact that many higher-order library functions have instances that are listless. For example, if f has a listless program, then so does map f. Moreover, the listless program for map f can be derived from the listless program for f, using the following schema:

(map schema)
if {y ← f x} has a listless program,
then {ys ← map f xs} has the listless program:

```
MAPF:     case xs of
              nil        ⇒  ys ← nil;  done
              cons x xs  ⇒  ys ← cons y ys;  y ← f x;  MAPF
```

This schema should be applied as follows. Let F be a listless program for f with input variable x and output variable y. Then the listless program for map f is as given above, where the portion of the program that reads y ← f x; MAPF is replaced by F, with MAPF substituted for every instance of done in F.

For example, not can be computed by the listless program:

```
NOT:      case x of
              true   ⇒  y ← false;  done
              false  ⇒  y ← true;  done
```

Applying the map schema to NOT gives the listless program NOTS shown in figure 1.

Figure 3 gives several other schemas, for the functions leftreduce, filter, while, and repeat; these functions are defined in the appendix. These schemas are not quite so general as the map schema, in that they require that their argument functions are primitives (such as + or $\leqslant$). (However, a way of relaxing this restriction is discussed in section 10.)

Figure 3: *Listless schemas.*

(leftreduce schema)
if f is a primitive function,
then {y ← leftreduce f a xs} has the listless program:

```
REDUCEF:    u ← a;
REDUCEF':   case xs of
                nil        ⇒  y ← u;  done
                cons x xs  ⇒  u ← f u x;  REDUCEF'
```

(filter schema)
if f is a primitive function,
then {ys ← filter f x} has the listless program:

```
FILTERF:    case xs of
                nil        ⇒  ys ← nil;  done
                cons x xs  ⇒
                    u ← f x;
                    case u of
                        true   ⇒  ys ← cons y ys;  y ← x;  FILTERF
                        false  ⇒  FILTERF
```

(while schema)
if f is a primitive function,
then {ys ← while f x} has the listless program:

```
WHILEF:     case xs of
                nil        ⇒  ys ← nil;  done
                cons x xs  ⇒
                    u ← f x;
                    case u of
                        true   ⇒  ys ← cons y ys;  y ← x;  WHILEF
                        false  ⇒  ys ← nil;  done
```

(repeat schema)
if f is a primitive function,
then {ys ← repeat f a} has the listless program:

```
REPEATF:    u ← a;
REPEATF':   ys ← cons y ys;  y ← u;  u ← f u;  REPEATF'
```

As an application of these schemas, note that the functions sum and upto can be defined by

```
        sum       =  leftreduce (+) 0
        upto m n  =  while (≤ n) (repeat (+ 1) m)
```

(Here (+ 1) is the function that adds one to its argument, and (≤ n) is the function that returns true if its argument is less than or equal to n.) The programs for SUM and UPTO given in figure 2 can be derived by applying the schemas in figure 3 and using the composition algorithm. For example, SUM is derived by applying the leftreduce schema where f is +, and composing this with the (trivial) listless program for {a ← 0}.

Another example of a listless library function is append. The function append differs from, say, merge, in that append is listless only when its type is known. That is, there exists a listless program to append two lists of numbers, and there also exists a listless program to append two lists of lists of numbers; but these are different programs. If one is using a typed functional language (say, Miranda [Turner 85] or Orwell [Wadler 85a]) then the transformation system can detect when the type of the arguments to append is known, and replace append by the appropriate listless program. (Again, a way of relaxing this restriction is discussed in section 10.)

The same technique used for append may also be used for

```
        reduceappend  =  rightreduce append nil
```

which appends together a list of lists (where each list is itself a list of numbers, or a list of lists of numbers, or so on).

9. Interfacing listless programs with graph reducers

Of course, not all programs are listless. Indeed, very few useful programs are listless as a whole. The main attraction of the listless transformer is that most programs contain significant portions which are listless. This means that if the listless transformer is to be usable in practice, then it is essential that it be possible to combine listless programs with other programs that are not listless. This section explains one method for combining general programs evaluated by a reduction machine with listless programs evaluated by a listless machine. Roughly speaking, the method is to run the reduction machine and the listless machine as coroutines.

It is assumed that the reduction machine behaves as follows. Initially, it is given a pointer to a node representing an expression to be evaluated. There will be various types of nodes, e.g., application nodes, constructor nodes, and so on. The evaluator examines the node to determine its type, and then performs the appropriate action. This action will overwrite the node with a new and equivalent node. This cycle is repeated on the new node, until the node is in an appropriate normal form. For example, a node representing square 3 will be overwritten by a node representing 3 × 3 and this in turn will be overwritten by a node representing 9, which is in normal form. (See [Peyton Jones 86] for a more detailed description of graph reduction.)

To interface a listless program with a reduction machine one simply defines a new kind of node, called an interface node, that corresponds to a value computed by a listless program. The interface node contains a pointer to the corresponding listless machine, which is represented by a vector (say) containing each of its input and output variables and the current state of the listless program. The action taken to reduce an interface node is simply to run the corresponding listless machine until it produces the desired output. While running, the listless machine may perform some input steps. Each input step will cause a recursive call of the reduction machine to evaluate the node pointed at by the input variable.

Recall from [Wadler 84a] that each output variable of the listless machine points at the location where the output is to be placed. This location itself initially contains the interface node that points at the listless machine. Thus, the interface node and the listless machine each point at the other (similar to the way in which two coroutines point to each other). When the listless machine performs an output step to the location, the interface node is overwritten with a constructor node. The fields of the new constructor node (e.g., the head and tail of a cons node) will themselves be interface nodes, containing pointers to the listless machine and pointed at by its output variables.

Using this method one can evaluate, for example, any term of the form f (g (h s)) where f and h are general functions to be evaluated by a reduction machine, g is a function to be evaluated by the corresponding listless program G, and s is a structure. The initial state of the corresponding graph is shown in figure 4.

For a second example, consider the Hamming problem: find a list sorted in ascending order such that the list contains the number 1; and the numbers 2 × i, 3 × i, and 5 × i whenever i is in the list; and no other numbers. This problem is a popular example in papers on functional programming, because it has a nice functional solution:

```
ham  =  cons 1
             (merge (map (× 2) ham)
                    (merge (map (× 3) ham)
                           (map (× 5) ham)))
```

The definition of ham is not, of itself, listless. However, it is easy to see that it can be rewritten in the form:

```
ham  =  cons 1 (f ham ham ham)
f xs ys zs  =  merge (map (× 2) xs)
                     (merge (map (× 3) ys)
                            (map (× 5) zs)))
```

where f can be seen to be listless by repeated application of the composition rule. Indeed, one could imagine a compiler performing this transformation automatically. The initial graph to be evaluated by the graph reducer is shown in figure 5.

Figure 4: *Initial configuration to evaluate* `f (g (h s))`, *where* `g` *has listless program* `G`.

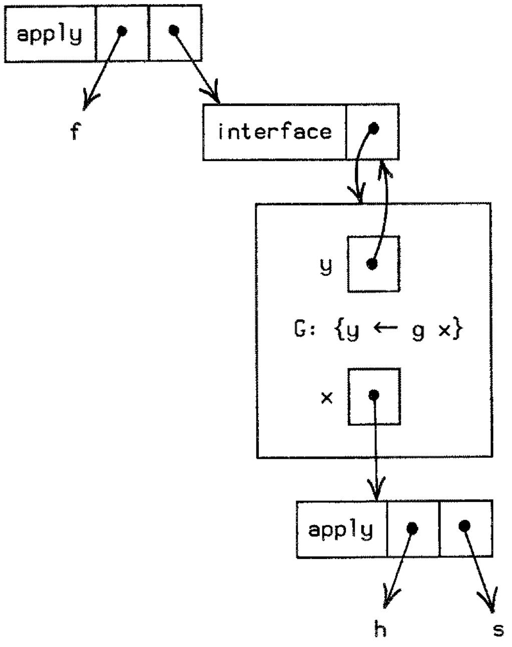
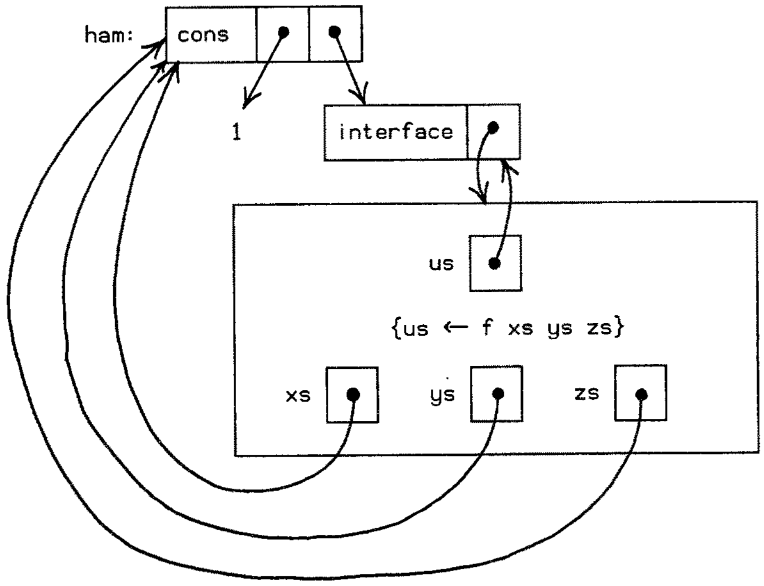


Figure 5: *Initial configuration for the Hamming problem.*



It should be possible to implement this method efficiently. For example, Johnsson and Augustsson's work on the G-machine [Johnsson 84] shows one efficient way of implementing a reduction machine on a conventional architecture, and it appears to be straightforward to extend the G-machine to include interface nodes as described above.

10. Almost listless programs and algebraic rules

The method described so far allows one to embed listless programs inside general graph reduction programs. It is also possible to do the reverse, that is, to embed general graph reduction program inside listless programs. This is done by adapting the listless schemas from section 8.

For example, the map schema shows how to convert a listless program for f into a listless program for map f. Now consider the case that f is a general function to be evaluated by the graph reducer. Then one can treat the program MAPF as an "almost listless" program to compute map f by simply re-interpreting the step y ← f x. This step is now taken as a variation of a primitive step. However, instead of applying f to the value of x and storing the result in y, it just constructs an unevaluated node representing the application f n, where n is the node pointed at by x, and stores a pointer to this node in y. Eventually the value of the node f n may be demanded by the graph reducer, and this will cause f n to be replaced by its value.

All of the schemas given in figure 3 can be re-interpreted in the same way. One can now also have a general almost listless program for append:

```
{ws ← append xs ys}

    APPEND:     case xs of
                    nil        ⇒  APPEND'
                    cons x xs  ⇒  ws ← cons w ws;  w ← x;  APPEND

    APPEND':    case ys of
                    nil        ⇒  ws ← nil;  done
                    cons y ys  ⇒  ws ← cons w ws;  w ← y;  APPEND'
```

Here the steps w ← x and w ← y are interpreted as copying the input node to the output node without yet forcing it to be evaluated. This schema should be used only when the type-checker cannot give more specific information about the type of some instance of append.

The composition algorithm for listless programs with primitives is still valid under this new interpretation of primitive steps. For example, consider a program containing the term map f ∘ map g where f and g are to be evaluated by the graph reducer. Let MAPF and MAPG be the almost listless programs for map f and map g, respectively. One can derive the composition MAPG >> MAPF using algorithm 3. The resulting almost listless program produces an output list containing elements of the form f (g n) where n is the node that is the corresponding element of the input list. In other words, in this case the composition algorithm achieves the same effect as the algebraic rule `map f ∘ map g = map (f ∘ g)`. In fact, the composition rule is more efficient, since it returns elements in the form `f (g n)` instead of in the form `(f ∘ g) n`.

The composition algorithm can achieve the effect of other algebraic rules as well. For example, [Wadler 81] presents four rules for composing the functions `map`, `leftreduce`, and `generate`; one of these is the rule for `map f ∘ map g` given above. All four of these rules, and many others beside, are subsumed by the listless schemas in section 8 combined with the composition algorithm. Further, in [Wadler 81] the function `generate` has to be treated as a single entity, but using the approach described here it can be defined in terms of simpler functions:

```
        generate f g h  =  map f ∘ while g ∘ repeat h
```

In addition, the composition algorithm handles many cases that cannot be handled easily by algebraic rules. For example, there are no simple algebraic rules to simplify terms of the form `while f ∘ map g` or `append xs (map f ys)`, because there are no simple, standard functions that correspond to these compositions. But the composition algorithm, or course, does apply to these terms. (See [Bellegarde 85] for other examples of algebraic rules.)

These results might be summarised by the motto: "listless composition reaches the parts algebraic rules cannot reach". On the other hand, algebraic rules also reach some parts that listless composition cannot reach. For example, `reverse ∘ reverse = identity` is a useful algebraic law for finite lists. The listless composition algorithm cannot achieve the effect of this law, since the function `reverse` is not listless.

11. Strong correctness

This section explains a small detail important to strong correctness, which has been ignored so far in this paper. In doing so, it also provides a semantic characterization of preorder traversal.

As usual, let $\bot$ denote an undefined value (e.g., the result of executing an infinite loop). When a listless machine is interfaced to a graph reduction machine, as described in sections 9 and 10, it is possible that the input to the machine may contain $\bot$. In this case, the output of the listless machine may be less defined than the output of the corresponding functional program.

For example, consider the listless program `NOTS`, defined in figure 1, which corresponds to the functional program `map not`. If `map not` is applied to the input list `[false, ⊥, true]` it will return the output list `[true, ⊥, false]`. On the other hand, if `NOTS` is applied to the same input list it will construct the output list `(cons true (cons y ys))`, where `y` and `ys` are output variables still to be bound, before it examines the $\bot$ in the input. Replacing the unbound variables by $\bot$, we have that

```
        cons true (cons ⊥ ⊥)  ⊑  [true, ⊥, false]
```

that is, the output returned by NOTS is less defined then the output returned by map not.

The difference between the lists returned by NOTS and map not will only be apparent if the general graph reducer examines, say, the third element before the second element. No problem can arise so long as all outputs and inputs to listless programs are traversed in preorder. In order to guarantee preorder traversal we define a new function pre that is equivalent to the identity function but forces its argument to be traversed in preorder.

First, we require a precise definition of structure. A structure s is either ⊥ or has the form c s1 ... sk where c is a constructor of arity k and s1, ..., sk are structures. Here are three examples of structures:

```
        cons true (cons false nil)
        cons true (cons ⊥ (cons false nil))
        cons (cons ⊥ nil) nil
```

(here cons is a constructor of arity 2, and nil, true, and false are constructors of arity 0). The first structure does not contain ⊥, and the second two do.

Preorder traversal of c s1 ... sk requires that s1 is traversed in preorder, then s2 is traversed in preorder, and so on up to sk. If ever the value ⊥ is encountered, then the traversal stops. So if si contains ⊥ then preorder traversal cannot distinguish between the two structures c s1 ... si s(i+1) ... sk and c s1 ... si ⊥ ... ⊥.

This motivates the following definition of pre. The function pre is defined such that for any constructor c of arity k,

```
        pre (c s1 ... sk)  =  c s1' ... sk'
```

where si' is ⊥ if there is any j < i such that sj contains ⊥, and si' is (pre si) otherwise. Here are three examples:

```
   pre (cons true (cons false nil))           =  cons true (cons false nil)
   pre (cons true (cons ⊥ (cons false nil)))  =  cons true (cons ⊥ ⊥)
   pre (cons (cons ⊥ nil) nil)                =  cons (cons ⊥ ⊥) ⊥
```

It is easy to see that pre s ⊑ s, and that pre s = s if s does not contain ⊥. Further, the preceding discussion shows that preorder traversal cannot distinguish between s and pre s for any structure s. Indeed, pre s is the largest structure (under the ⊑ ordering) that is indistinguishable from s using preorder traversal.

It is easy to see that the listless machine NOTS is strongly equivalent to (pre ∘ map not ∘ pre), and in general that the listless machine MAPF is strongly equivalent to (pre ∘ map f ∘ pre), where MAPF and f are as in the (map schema) in section 8. More generally, strong equivalence with the listless machine can always be guaranteed by applying pre to the input and output of the corresponding function.

In many cases, strong correctness is guaranteed even without the use of `pre`. For example, let `f` be a primitive function (so `f x` returns $\perp$ whenever `x` contains $\perp$, that is, `f` is hyper-strict). Then the function `(filter f)` is guaranteed to traverse its input and output in preorder, or more formally,

```
filter f  =  pre ∘ filter f ∘ pre
```

Similarly, `(leftreduce f a)`, `(while f)`, and `(repeat f a)` are also guaranteed to be preorder whenever `f` is primitive; and `some`, `every`, and `merge` also preorder. Further, if `f` is primitive then

```
pre ∘ map f  =  map f ∘ pre  =  pre ∘ map f ∘ pre
```

This means that many compositions of `map f` with other listless functions may automatically be deduced to be preorder. The functions `append` and `reduceappend` have a similar property.

In the remaining cases where functions cannot be guaranteed preorder, two other methods may be used. One is to use functions which are preorder by definition, such as `map'`, where

```
map' f  =  pre ∘ map f ∘ pre
```

and to only replace instances of `map'`, not `map`, by listless programs. A second method is to apply strictness analysis ([Hughes 85b, Wadler 85b]) to determine cases where preorder traversal is guaranteed. For example, if an instance of `map` takes as input and produces as output a list that is strict in the head, then preorder traversal is guaranteed.

12. Conclusions

One of the pervading principles of computer science is modularity: the ability to decompose a problem into parts which can be solved seperately. Indeed, one of the fundamental advantages of functional programming is that it provides modular ways of creating programs, such as function composition. (See [Hughes 85c] for a further discussion of modularity in functional programs.)

Therefore, it was disquieting that the listless transformer, as originally described in [Wadler 84a,b], could only deal with programs in a monolithic rather than a modular fashion. The major advance of this work is to introduce methods that allows one to construct and transform listless programs in a modular way. This paper has discussed two different kinds of modularity: the ability to compose two listless programs into a single listless program, and the ability to combine listless programs with general graph reduction programs.

Of course, modularity is desirable not just because it is elegant in theory, but because it is essential to practice. It is precisely because they support modularity that the techniques introduced in this paper bring listlessness much closer to practical use.

Acknowledgements

Richard Bird, Geoff Barrett, and John Hughes made useful comments on earlier versions of this paper. This work was performed while on a research fellowship funded by ICL.

Appendix 1. Standard functions

The call (map f xs) applies f to every element of the list xs.

```
map square [1, 2, 3]          =  [1, 4, 9]
```

The calls (rightreduce f a xs) and (leftreduce f a xs) combine elements of the list xs using the binary function f, using a as the value for the empty list; they associate to the right and left respectively.

```
rightreduce (+) 0 [1, 2, 3]   =  1 + (2 + (3 + 0))
leftreduce  (+) 0 [1, 2, 3]   =  ((0 + 1) + 2) + 3
```

The call (filter p xs) returns all elements of xs that satisfy p, and the call (while p xs) returns the initial segment of xs that satisfies p.

```
filter odd [3, 1, 4, 5, 2]    =  [3, 1, 5]
while odd  [3, 1, 4, 5, 2]    =  [3, 1]
```

The call (repeat f a) returns the infinite list [a, (f a), (f (f a)), ...].

```
repeat (+ 1) 0                =  [0, 1, 2, 3, ...]
```

The call (upto m n) returns the list of numbers from m up to n.

```
upto 1 3                      =  [1, 2, 3]
```

The function append appends two lists, the function merge merges two sorted lists into a single sorted list, and the function reduceappend appends together a list of lists:

```
append [1, 2, 3] [4, 5, 6]    =  [1, 2, 3, 4, 5, 6]
merge [1, 3, 4] [2, 5, 6]     =  [1, 2, 3, 4, 5, 6]
```

Appendix 2. Function definitions

```
map f nil                         =  nil
map f (cons x xs)                 =  cons (f x) (map f xs)

rightreduce f a nil               =  a
rightreduce f a (cons x xs)       =  f x (rightreduce f a xs)

leftreduce f a nil                =  a
leftreduce f a (cons x xs)        =  leftreduce f (f a x) xs

filter p nil                      =  nil
filter p (cons x xs)              =  cons x (filter p xs)      IF p x
                                     filter p xs               OTHERWISE

while p nil                       =  nil
while p (cons x xs)               =  cons x (while p xs)       IF p x
                                     nil                       OTHERWISE

repeat f a                        =  cons a (repeat f (f a))

merge nil ys                      =  ys
merge (cons x xs) nil             =  cons x xs
merge (cons x xs) (cons y ys)     =  cons x (merge xs (cons y ys))  IF x < y
                                     cons y (merge (cons x xs) ys)  OTHERWISE

not true                          =  false
not false                         =  true

and true  x                       =  x
and false x                       =  false

or  true  x                       =  true
or  false x                       =  x

every                             =  rightreduce and true
some                              =  rightreduce or false
```

References

[Bellegarde 85] Bellegarde, F. Convergent term rewriting systems can be used for program transformation. This volume.

[Ganzinger and Giegrich 84] Ganzinger, H. and Giegrich, R. Attribute coupled grammars. *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction. SIGPLAN Notices*, 19(6). June, 1984.

[Hughes 85a] Hughes, R.J.M. Lazy memo functions. *Conference on Functional Programming Languages and Computer Architecture*, Nancy, France. September, 1985.

[Hughes 85b] Hughes, R.J.M. Strictness detection on non-flat domains. This volume.

[Hughes 85c] Hughes, R.J.M. Why functional programming matters. Internal report, Programming Methodology Group, Chalmers Institute of Technology, Gothenburg, Sweden. 1985.

[Johnsson 84] Johnsson, T. Efficient compilation of lazy evaluation. *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction. SIGPLAN Notices*, 19(6). June, 1984.

[Mont-Reynaud 76] Mont-Reynaud, B. Removing trivial assignments from programs. Stanford University, Dept. of Computer Science technical report STAN-CS-76-544. March, 1976.

[Peyton Jones 86] Peyton Jones, S. *Implementing Functional Languages by Graph Reduction*. To appear.

[Turner 85] Turner, D. Miranda: a lazy functional language with polymorphic types. *Conference on Functional Programming Languages and Computer Architecture*, Nancy, France. September, 1985.

[Wadler 81] Wadler, P.L. Applicative style programming, program transformation, and list operators. *Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire. October, 1981.

[Wadler 84a] Wadler, P.L. Listlessness is better than laziness: lazy evaluation and garbage collection at compile-time. *ACM Symposium on Lisp and Functional Programming*, Austin, Texas. August, 1984.

[Wadler 84b] Wadler, P.L. Listlessness is Better than Laziness. Ph.D. Dissertation, Carnegie-Mellon University. August, 1984.

[Wadler 85a] Wadler, P.L. An introduction to Orwell. Internal report, Programming Research Group, Oxford University. 1985.

[Wadler 85b] Wadler, P.L. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). To appear.