Listlessness is Better than Laziness
Lazy evaluation and garbage collection at compile-time

Philip Wadler
Programming Research Group, Oxford University
8-11 Keble Rd., Oxford, OX1 3QD

0. Introduction

One reason for interest in functional programming is that it is well-suited for program development by transformation: begin by writing a clear (but inefficient) program, and transform this to an efficient (but less clear) program. Promising research has occured in this area [Burstall and Darlington 77; Clark and Darlington 80; Feather 82; Manna and Waldinger 79; Scherlis 81] and it is ripe for further exploration.

Most desirable is a transformation system that can always automatically transform a program to optimal form. It is unreasonable to expect this in general, but it may be possible over limited domains.

One important source of both clarity and inefficiency in functional programs is the use of intermediate lists. This paper describes a <u>listless transformer</u> that, where possible, eliminates all intermediate lists from a program.

The source programs for this transformer are expressed in a functional language with lazy evaluation. The target programs are graphs (similar to flowchart schemata) that can be executed by a simple machine (similar to a finite state machine). Because this <u>listless machine</u> needs neither delayed evaluation nor complicated storage allocation, one can say that the transformer performs lazy evaluation and garbage collection at compile-time.

The main drawback is that, of course, not all programs can be transformed to this form. The method has been shown to apply to all programs that can be evaluated in a bounded amount of space, not counting space for input or output. This class has been formally characterized. It includes many useful programs, but excludes many others.

The listless transformer has been implemented, and run on some non-trivial programs including pattern matching, the telegram problem, and some text processing problems inspired by Unix.

This paper is mainly a summary of the author's Ph.D. thesis [Wadler 84], but it also includes a few results that are not in the thesis.

This paper is organized as follows. Section 1 explains why intermediate lists are a source of both clarity and inefficiency. Section 2 gives an example of the transformation method. Section 3 presents the machine for executing transformed programs. Section 4 outlines theoretical results and extensions to the method. Section 5 gives a more realistic example of program transformation, using a text processing method inspired by Unix. Section 6 discusses related work. Section 7 suggests topics for future work and presents conclusions.

1. Intermediate Lists

Consider finding the sum of the squares of the numbers from 1 to n. In a functional style, this might be written

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

A key feature of this style is that it uses operators (upto, map, sum) that encode common patterns of computation ("consider the numbers from 1 to n", "apply a function to each element", "sum a collection of elements"). In contrast, in an imperative program for the same task

```
(2)  s ← 0;
     for i ← 1 to n do
         s ← s + (i * i);
     result ← s;
```

these operators cannot be directly expressed (except for upto, which is built into the language as for). These operators have also been called <u>paradigms</u> [Floyd 79] and <u>cliches</u> [Waters 79].

Intermediate lists are central to this style; they are the "glue" that holds it together. For example, the intermediate list [1,2,...,n] connects upto to map, and the intermediate list [1,4,...,n^2] connects map to sum.

But intermediate lists have a cost in run-time efficiency. In addition to the space occupied by a list, there is also the time required to allocate the list (one cons operation per element), traverse it (one null, head, and tail operation per element), and deallocate it (garbage collection).

Lazy evaluation has been recommended as a method for reducing the costs of this style [Henderson and Morris 76, Friedman and Wise 76]. If program (1) is executed using lazy evaluation, the basic operations will occur in the same order as in program (2). However, although lazy evaluation eliminates the space cost (since the list need not be in memory all at once), it does nothing to lessen the time cost.

In addition, if the program requires multiple traversals of a list, lazy evaluation does not even save the space cost. For example, lazy evaluation of the function

```
average xs  =  sum xs / length xs
```

requires space proportional to the list xs. If the lists used are long (for example, when processing a file as discussed in section 5) this can be a significant problem.

This style of programming also makes heavy use of function calls. For example, executing program (1) requires at least 4n function calls (n calls each to sum, map, square, and upto) as compared to none for program (2). The listless machine makes no function calls, so this cost is also eliminated by the listless transformer.

2. A simple example

This section explains the listless transformer by tracing its performance on a simple example. Details of the transformer are given in [Wadler 84].

Consider the function definitions

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

not true = false
not false = true

trivial xs = map not (map not xs)
```

Then map applies a function to every element of a list, and trivial is the identity function over lists of booleans:

```
map not [true, false] = [false, true]
trivial [true, false] = [true, false]
```

(Here [true, false] is an abbreviation for cons true (cons false nil).)

This simple problem involves no <u>primitives</u>, that is data types like integers or operations on them like addition. Extension of the transformer to handle primitives is discussed in section 4.1.

For this example, the input to the transformer is the <u>specification</u>

```
(0)  {ys ← trivial xs}
```

The listless transformer will convert this to a program for a listless machine with <u>input variable</u> xs and <u>output variable</u> ys. Given xs, the machine will calculate trivial xs and put the value into ys. In general, a specification may contain more than one input or output variable.

The transformer works by building a graph. Each vertex of the graph is labeled with a specification, like (0) above. Each edge of the graph is labeled with a sequence of <u>input</u> and <u>output</u> <u>steps</u>. The graph derived for this example is shown in figure 1. The next section describes how this graph is interpreted as the program of a listless machine.

The graph is derived as follows. Symbolic evaluation (unfolding equations as much as possible) transforms (0) into

```
(1)  {ys ← map not (map not xs)}
```

Initially, the graph contains only an edge from (0) to (1) labeled with an empty sequence of steps.

Further evaluation cannot take place until more is known about the value of xs. Therefore, an input step must take place. There are two possible cases, represented by the input steps:

```
nil ← xs
cons x xs ← xs
```

These can be thought of as corresponding to a case expression

```
case xs of
     nil:  ...
     cons x xs:  ...
```

The variable xs is called a <u>branch</u> <u>variable</u>. This method of doing a case analysis on a variable that is needed for further evaluation is similar to the use of <u>bomb</u> <u>sets</u> in [Boyer and Moore 75, 79] and <u>driving</u> in [Turchin et al 82].

In the first case, performing the input step nil ← xs transforms (1) to

```
{ys ← map not (map not nil)}
```

That is, the xs in (1) is just replaced by nil. Symbolic evaluation simplifies this to

```
{ys ← nil}
```

At this point an output step ys ← nil may be performed, yielding the new specification

```
(2)  {}
```

Executing this step puts nil in the location pointed to by ys, and the new specification indicates that execution may then terminate. A vertex labeled with (2) is added to the graph, along with an edge from (1) to (2) labeled with the step sequence

```
nil ← xs;  ys ← nil
```

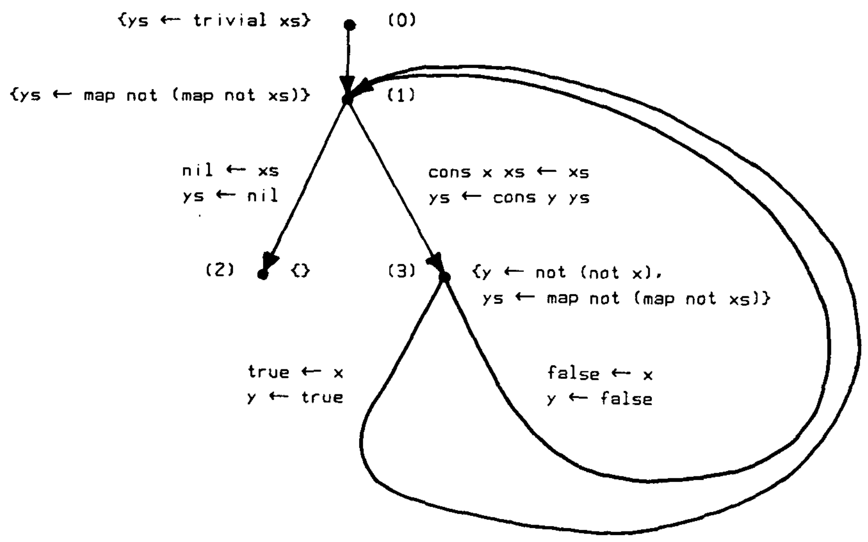
In the second case, performing the input step cons x xs ← xs transforms (1) to

```
{ys ← map not (map not (cons x xs))}
```

which evaluates to

```
{ys ← cons (not (not x)) (map not (map not xs))}
```

Figure 1: Graph produced by listless transformer



At this point, it is known that the output variable ys must denote a cons node, so the output step ys ← cons y ys can be performed, yielding the new specification

```
(3)  {y ← not (not x),  ys ← map not (map not xs)}
```

Executing the step ys ← cons y ys builds a cons node in the (old) location pointed to by ys, and the new specification indicates that the head field of this node must be set to contain not (not x) and the tail field must be set to contain map not (map not xs). (The execution of output steps is explained further in the next section.) The vertex (3) is added to the graph, with an edge from (1) to (3) labeled

```
cons x xs ← xs;  ys ← cons y ys
```

To further evaluate (3), an input step must be performed on x. Again there are two cases, x is either true or false. In the first case, the input step true ← x transforms (3) to

```
{y ← not (not true),  ys ← map not (map not xs)}
```

which evaluates to

```
{y ← true,  ys ← map not (map not xs)}
```

The output step y ← true can now be performed, yielding the specification

```
{ys ← map not (map not xs)}
```

This specification is already vertex (1) in the graph, so an edge is added from (3) to (1) labeled

```
true ← x;  y ← true
```

This edge introduces a loop into the graph, corresponding to a loop in the program for the listless machine.

Similarly, analyzing the second case adds another edge from (3) to (1) labeled

```
false ← x;  y ← false
```

The completed graph is shown in figure 1.

In this example, the choice of branch variables was straightforward. In general, this may not be the case. For example, consider transforming the specification

```
{us ← append xs ys,  vs ← append zs xs}
```

If xs is chosen as the branch variable, then the input step cons x xs ← xs eventually leads to the new specification

```
{us ← append xs ys,  vs ← append zs (cons x xs)}
```

Repeatedly choosing xs as the branch variable would generate an infinite graph, whereas choosing zs as the branch variable would lead to a finite graph. To resolve this problem, the listless transformer performs a breadth-first search over all possible branch variables. (The search must be breadth-first, because a depth-first search might get sidetracked down an infinitely long fruitless path, e.g., always choosing xs.) This search is not too expensive, because there are only a few possible branch variables at each point.

3. The listless machine

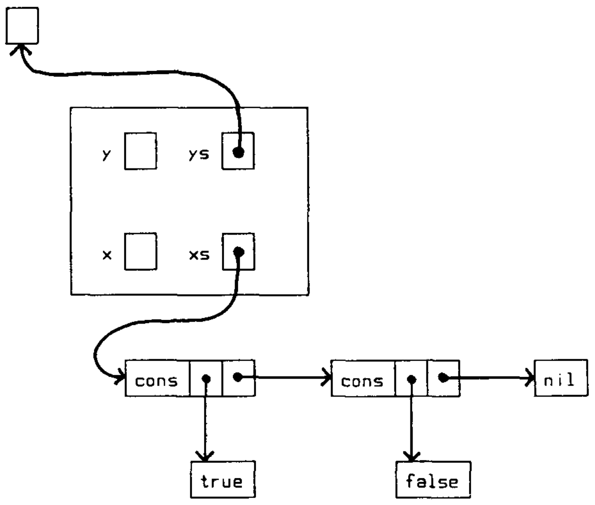
The graph in figure 1 may be written in an equivalent program-like notation:

```
0:  goto 1

1:  case xs of
        nil:  ys ← nil;  done
        cons x xs:  ys ← cons y ys;  goto 3

3:  case x of
        true:  y ← true;  goto 1
        false:  y ← false;  goto 1
```

This program is executed by a listless machine with two input registers, x and xs, and two output registers, y and ys. To calculate trivial [true, false] the machine is initialized as follows:



Notice that ys initially points to a location where output is to be placed.

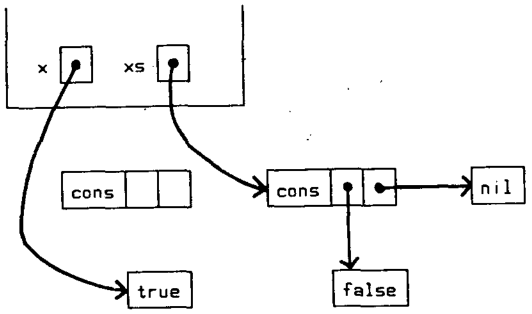
Type declarations for such a machine might be given informally as follows:

```
type bool  =  true: ()  | false: ()
     list  =  nil: ()   | cons: (head:^bool, tail:^list)

var  x:^bool,  xs:^list,  y:^^bool,  ys:^^list
```

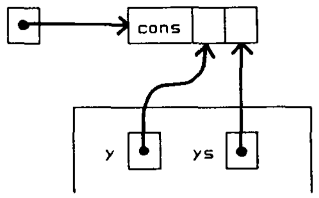
Here ^ denotes "pointer to". Note that the output variables contain one extra level of indirection, they are pointers to pointers.

After executing the input step cons x xs ← xs the new state of the machine is



The cell originally pointed to by xs can be reclaimed as part of this step. All storage can be reclaimed in this way, so there is no need for reference counts or a mark/scan garbage collector.

Next, the machine executes the output step ys ← cons y ys yielding the new state



If this step is unclear, it may help to visualize how it could be written in an imperative language:

```
^ys ← new list cell
y ← @((^ys).head)
ys ← @((^ys).tail)
```

Here @ indicates "address of" and ^ indicates dereferencing.

Readers familiar with Prolog [Kowalski 79] may observe that output variables in the listless machine and logical variables in Prolog are similar in that both specify a location where output is to be placed. Because Prolog allows backtracking, logical variables are more powerful than output variables, but more difficult to implement. Also, programs for the listless machine are related to programs compiled using an optimization called tail recursion, particularly tail recursion modulo cons [Steele 77, Warren 80].

4. Discussion

4.1 Primitives

It is not hard to extend the transformer to handle primitives such as operations on integers. This is done by introducing a new kind of step, called a primitive step, in addition to input and output steps. For example, the result of transforming the sum-of-squares program (1) from section 1 is:

```
0:  s ← 0;  i ← 1;  u ← (i > n);  goto 1

1:  case u of
        true:   result ← s;  done
        false:  x ← i * i;   s ← s + x;
                i ← i + 1;   u ← (i > n);  goto 1
```

This program is essentially identical to the imperative program (2) previously presented.

When primitives steps are not allowed, listless machines are similar to finite state machines. In particular, a listless machine with m input lists and n output lists is like a finite state machine with m read-only, no-backup input tapes and n write-only, no-backup output tapes. When extended to include primitives, listless machines are similar in power to flowchart schemata [Paterson and Hewitt 70].

4.2 Range of applicability

Of course, not all functions can be converted to listless machines. Examples of functions which can be converted are appending two lists, merging two sorted lists, and dividing a list into two lists of odd and even elements.

```
append [1,2] [3,4] = [1,2,3,4]
merge [1,3] [2,4]  = [1,2,3,4]
divvy [1,2,3,4]    = [[1,3], [2,4]]
```

Examples of functions which cannot be converted are reversing a list, sorting a list, or appending a list to itself.

```
reverse [1,2,3,4]   =  [4,3,2,1]
sort [3,1,4,2]      =  [1,2,3,4]
self-append [1,2]   =  [1,2,1,2]
```

The last function cannot be made listless because it requires two traversals over the input to produce the output, and a listless machine may only traverse its input once.

The listless transformer only applies to data that is list-shaped. For example, the input data may be integers, lists of integers, or lists of lists of integers, but it cannot be trees or arbitrarily deeply nested lists. Trees cannot be handled because, even if the tree need not be kept in memory, a stack of pointers into the tree will be needed, and this requires unbounded internal storage. One interesting topic for future research is to extend this work to include trees, perhaps by adding a stack to the listless machine.

In some ways, the listless transformer is similar in power to a lexical analyzer generator (such as Lex [Lesk 75]). Both are suitable for roughly the same sort of problems, namely those that can be processed by finite-state machines. Also, both produce as output programs that are efficient, but may be larger or different in structure from those an unaided programmer might write. Of course, the input language of the listless transformer is at a much higher level than the input language of a lexical analyzer generator.

4.3 Decidability

In general, a recursive function can be converted to a listless machine if and only if it is subject to bounded evaluation, that is, if the function can always be evaluated in constant bounded space, not counting space occupied by the input or the output. This notion has been formalized, and the theory of bounded evaluation has been used to motivate the development of the transformer and demonstrate its correctness [Wadler 84].

Does there exist a decision procedure for determining if a recursive function is subject to bounded evaluation? Unfortunately, no. It is not hard to show that this problem is equivalent to the halting problem. (I am indebted to Alan Staughton of Edinburgh for the proof.) Further, if the function may include primitives, then results from the theory of program schemata show that there is not even a semi-decision procedure [Paterson and Hewitt 70].

Given this, the listless transformer performs as well as one could ask. If the function has no primitives, the transformer will terminate if and only if the function is subject to bounded evaluation (b.e.). If primitives are allowed, then the transformer may fail to terminate for some b.e. functions. However, in practice I have encountered no b.e. functions that cause the transformer to loop, except for some pathological functions created for that purpose.

4.4 Evaluation order

Some useful functional programs, which intuitively should be evaluable in bounded space, cannot be evaluated in bounded space by a normal-order evaluator. For example, the `normalize` program discussed in section 5 has this property. This problem could be resolved by designing the listless transformer to search over all evaluation orders, but this would be costly. An alternative, adopted by the current implementation, is to always perform all possible reductions, except where the user indicates that normal-order reduction must be used to preserve termination.

This problem is not restricted to the listless transformer. It has been discovered independently by Hughes, in the context of efficient evaluation of lazy functional languages [Hughes 84]. He suggests annotating programs to indicate where parallel evaluation and synchronization are needed in order for the program to evaluate in bounded space. Hughes and the author are collaborating on designing a version of the listless transformer that accepts programs annotated in this way.

5. A larger example

As a more realistic example of this style of programming and the performance of the listless transformer, consider the following text processing problem, motivated by some features of the Unix operating system [Ritchie and Thompson 74].

In Unix, a file is regarded as a sequence of lines (terminated by newline characters, '↲') where each line is a sequence of fields (separated by one or more space characters, '␣'). Taking a file as a list of characters, it is easy to define functions that break a file into lines and a line into fields.

```
lines "we␣␣like↲␣lists␣↲"  =  ["we␣␣like","␣lists␣"]

fields "we␣␣like"  =  ["we","like"]
fields "␣lists␣"   =  ["lists"]
```

(Here "abc" denotes a list of characters ['a','b','c'].) Note that `fields` discards information about how many spaces appear before or after a field.

One nice property of this style of programming is that the reader need not know the exact definitions of `lines` or `fields`. But for the curious, complete definitions are given in appendix I. This appendix provides an extended example of programming in a functional style with higher-order functions and intermediate lists.

It is now easy to define a function `ix` that processes an entire file. The file is broken into lines, and each line is broken into fields.

```
ix  =  map fields . lines
```

For example,

```
ix "we␣␣like↲␣lists␣↲"  =  [["we", "like"], ["lists"]]
```

(Here . denotes function composition, (f . g) x = f (g x).)

Similarly, there are functions that put fields together into lines, and lines together into files:

```
unfields ["we","like"]  =  "we␣like"
unfields ["lists"]      =  "lists"

unlines ["we␣like","lists"]  =  "we␣like↲lists↲"

unix  =  unlines . map unfields
```

Using these operators, it is easy to write many simple text processing operators. For example,

```
normalize  =  unix . ix
```

removes redundant spaces from a file, and

```
wc xs  =  [length xs,
           length (fields xs),
           length (lines xs)]
```

corresponds to the Unix word count utility, which counts the number of spaces, words, and lines in a file.

Appendix II shows the listless program for wc generated by the listless transformer. The structure of the program is similar to the structure of a finite state machine for recognizing files. The different states correspond to being at the beginning of a line, being in a group of spaces between fields, or being in a field. The program for normalize has a similar structure.

The interested reader may find it an illuminating exercise to attempt to write an imperative, structured program for wc that makes as few tests as the listless program. I suspect that it is impossible unless one uses goto's. The programs for normalize and for merging two sorted lists also seem to require goto's for maximum efficiency. (See [Knuth 74] for a discussion of such programs.)

Further, when writing imperative programs for wc and normalize, one may be tempted to use routines that store a line or a word of text at a time, in order to have a clear program structure. In contrast, the listless programs need only store one character at a time.

These examples suggest not only that the listless transformer allows one to write programs in a clear, abstract style, but also that the programs produced may sometimes be more efficient than the programs an imperative programmer might write.

The listless transformer has also been applied to pattern matching and the Telegram Problem.

6. Related work

Most program transformation systems use one of two methods. The first method is based on folding and unfolding (e.g., see [Burstall and Darlington 77, Feather 82]). The second method is based on transformation schemas, that is applying instances of laws such as map f . map g = map (f . g) (e.g., see [Backus 78, Kieburtz and Schultis 81, Wadler 81]). The listless transformer can be thought of as using a new, third method.

The three methods have complementary advantages: the fold-unfold method requires the most work from the user, but is also most general; the schema method and the listless transformer require less work from the user, but are restricted to limited domains.

One can draw an analogy between program transformation and theorem proving. Transforming a program using the fold-unfold method is like proving a theorem from axioms. Transforming a program using the schema method is like proving a theorem by applying algebraic laws. Transforming a program using the listless transformer is like proving a theorem by applying a (semi-)decision procedure.

A previous work by this author used the schema method [Wadler 81]. That work used a set of four rules to transform combinations of the higher-order functions reduce, map, and generate. The hope was to extend the set of rules to handle additional functions, but this proved difficult. The listless transformer, by using a uniform procedure rather than a set of canned rules, handles a much wider range of functions.

Feather is one of the few researchers to consider transformation of large programs [Feather 82]. Most of the transformations performed by Feather are of the same general type as those performed by the listless transformer. In particular, most applications of his combination strategy are concerned with eliminating intermediate lists, and most applications of his tupling strategy are concerned with eliminating multiple traversals of lists. The listless transformer does not apply directly to most of the programs tranformed by Feather, because they are not subject to bounded evaluation. However, Feather's work suggests that if integrated with other methods the listless transformer could handle a significant proportion of the transformations required for developing realistic programs.

7. Conclusions

This paper has discussed a style of functional programming based on operators that communicate by passing intermediate lists. A listless transformer was presented that converts programs written in this style into an efficient form. The transformed programs are executed by listless machines, which are related to finite state machines and flowchart schemata. The transformer has been implemented and applied to some non-trivial examples.

The key idea in discovering this transformer was to concentrate on a limited domain, namely those programs that are subject to bounded evaluation. A formal characterization of this class of programs exists, and has been used in developing the transformer and showing it correct.

The source of strength of the listless transformer is also its major drawback: because it is designed for a limited domain, there are many useful programs that it cannot transform. Two approaches can be taken to lessening this problem, both of which are worth exploring.

First, one can extend the domain handled by the listless transformer. Just as listless machines are analogous to finite state machines, there may be listless stack machines analogous to deterministic pushdown automata. Such machines would be useful for processing trees and parsing (deterministic) context-free grammars.

Second, one can integrate the listless transformer with other approaches. One possibility is to allow users to annotate programs to indicate which intermediate lists should be eliminated by the transformer, and which should be allowed to remain. Another idea is to develop a user-directed program transformation system, and provide the listless transformer as one of a range of tools available to the user for manipulating programs.

Appendix I. Definition of text processing functions

In this appendix, : is infix for cons, and ++ is infix for append. For example,

```
1:[] = cons 1 nil = [1]
[1,2] ++ [3,4] = [1,2,3,4]
```

The notation used is similar to KRC [Turner 81].

```
parse takefun dropfun  =  map takefun . repeat dropfun

repeat f []      =  []
repeat f (x:xs)  =  (x:xs) : (repeat f (f (x:xs)))

lines       =  parse takeline dropline
takeline    =  take (not . linesep)
dropline    =  drop-one . drop (not . linesep)

fields      =  parse takefield dropfield . drop fieldsep
takefield   =  take (not . fieldsep)
dropfield   =  drop fieldsep . drop (not . fieldsep)

linesep '↵'      =  true
linesep '␣'      =  false
linesep other    =  false

fieldsep '↵'     =  true
fieldsep '␣'     =  true
fieldsep other   =  false

take p []        =  []
take p (x:xs)    =  x : take p xs,    if p x
                 =  [],               otherwise

drop p []        =  []
drop p (x:xs)    =  drop p xs,        if p x
                 =  x:xs,             otherwise

drop-one []        =  []
drop-one (x:xs)    =  xs

unlines []         =  []
unlines (xs:xss)   =  xs ++ ['↵'] ++ unlines xss

unfields []        =  []
unfields [xs]      =  xs
unfields (ys:(xs:xss))  =
                      ys ++ ['␣'] ++ unfields (xs:xss)
```

Appendix II. Listless program for word count

Initial specification:

```
{cc ← length xs,
 ww ← length (fields xs),
 ll ← length (lines xs)}
```

Transformed program:

```
0:  {initialize}
    c ← 0;  w ← 0;  l ← 0;  goto 1

1:  {get next character, at beginning of line}
    case xs of
        nil:  cc ← c;  ww ← w;  ll ← l;  done
        cons x xs:  c ← c + 1;  l ← l + 1;  goto 2

2:  {examine next character, at beginning of line or
     between fields}
    case x of
        '↵':  goto 1
        '␣':  goto 3
        other:  w ← w + 1;  goto 4

3:  {get next character, between fields}
    case xs of
        nil:  cc ← c;  ww ← w;  ll ← l;  done
        cons x xs:  c ← c + 1;  goto 2

4:  {get next character, inside a field}
    case xs of
        nil:  cc ← c;  ww ← w;  ll ← l;  done
        cons x xs:  c ← c + 1;  goto 5

5:  {examine next character, inside a field}
    case x of
        '↵':  goto 1
        '␣':  goto 3
        other:  goto 4
```

References

[Backus 78] Backus, J. Can programming be liberated from the VonNeuman style? A functional style and its algebra of programs (Turing award lecture). Communications of the ACM, 21(8):613-641, August, 1978.

[Boyer and Moore 75] Boyer, R. S., and Moore, J. S. Proving theorems about Lisp functions. Journal of the ACM, 22(1):129-144, January, 1975.

[Boyer and Moore 79] Boyer, R. S., and Moore, J. S. A Computational Logic. Academic Press, New York, 1979.

[Burstall and Darlington 77] Burstall, R. M., and Darlington, J. A transformation system for developing recursive programs. Journal of the ACM, 24(1):44-67, January, 1977.

[Clark and Darlington 80] Clark, K., and Darlington, J. Algorithm classification through synthesis. Computer Journal, 23(1):61-65, February, 1980.

[Feather 82] Feather, M. S. A system for assisting program transformation. ACM Transactions on Programming Languages and Systems, 4(1):1-20, January, 1982.

[Floyd 79] Floyd, R. W. The paradigms of programming (Turing award lecture). Communications of the ACM, 22(8):455-460, August, 1979.

[Friedman and Wise 76] Friedman, D. P., and Wise, D. S. Cons should not evaluate its arguments. In Michaelson and Milner (editors), Automata, Languages, and Programming, 257-284, Edinburgh University Press, 1976.

[Henderson and Morris 76] Henderson, P., and Morris, J. H. A lazy evaluator. In 3'rd Symposium on Principals of Programming Languages, 95-103, ACM, Atlanta, GA, 1976.

[Hughes 84] Hughes, J. M. The design and implementation of programming languages. D.Phil. Thesis, Oxford University, 1984.

[Kieburtz and Schultis 81] Kieburtz, R. B., and Schultis, J. Transformations of FP program schemes. In Conference on Functional Programming Languages and Computer Architecture, 41-48, ACM, October, 1981.

[Knuth 74] Knuth, D. E. Structured programming with go to statements. ACM Computing Surveys, 6(4):261-302, December, 1974.

[Kowalski 79] Kowalski, R. Algorithm = Logic + Control. Communications of the ACM, 22(7), July, 1979.

[Lesk 75] Lesk, M. E. Lex - a lexical analyzer generator. Technical report, Bell Labs, Murray Hill, NJ, 1975.

[Manna and Waldinger 79] Manna, Z., and Waldinger, R. Synthesis: Dreams => Programs. IEEE Trans. Software Engineering, SE-5(4):157-164, July, 1979.

[Paterson and Hewitt 70] Patterson, M. S., and Hewitt, C. E. Comparative schematology. Project MAC Conference on Concurrent Systems and Parallel Computation, Wood's Hole, MA, 1970. Also, AI Memo 21, AI Lab, MIT.

[Ritchie and Thompson 74] Ritchie, D. M., and Thompson, K. L. The Unix time-sharing system. Communications of the ACM, 17(7), July, 1974.

[Scherlis 81] Scherlis, W. L. Program improvement by internal specialization. In 8'th Symposium on Principals of Programming Languages, 41-49, ACM, Williamsburg, VA, January, 1981.

[Steele 77] Steele, G. L., Jr. Debunking the expensive procedure call myth, or, Procedure call implementations considered harmful, or, LAMBDA: the ultimate goto. In Proc. ACM Annual Conference, 153-162, Seattle, WA, October 1977.

[Turner 81] Turner, D. A. Recursion equations as a programming language. In Darlington, et al (editors), Functional Programming and Its Applications, Cambridge University Press, 1981.

[Turchin et al 82] Turchin, V. F., Nirenberg, R. M., and Turchin, D. V. Experiments with a supercompiler. ACM Symposium on Lisp and Functional Programming, Pittsburgh, PA, August, 1982.

[Wadler 81] Wadler, P. L. Applicative style programming, program transformation, and list operators. In Conference on Functional Programming and Computer Architecture, 25-32, ACM, October, 1981.

[Wadler 84] Wadler, P. L. Listlessness is better than laziness. PhD thesis, Carnegie-Mellon University, 1984.

[Warren 80] An improved Prolog implemenation which optimizes tail-recursion. Dept. of AI Research Report 141, University of Edinburgh, 1980.

[Waters 79] Waters, R. C. A method for analyzing loop programs. IEEE Trans. on Software Engineering, SE-5(3):237-247, May, 1979.