

INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

The following explanation of techniques is provided to help clarify notations which may appear on this reproduction.

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.
2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.
3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.*
4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.*

*For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.

UMI University
Microfilms
International

PREVIEW

8605993

Wadler, Philip Lee

**LISTLESSNESS IS BETTER THAN LAZINESS: AN ALGORITHM THAT
TRANSFORMS APPLICATIVE PROGRAMS TO ELIMINATE INTERMEDIATE
LISTS**

Carnegie-Mellon University

Ph.D. 1984

**University
Microfilms
International** 300 N. Zeeb Road, Ann Arbor, MI 48106

PREVIEW

PREVIEW

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ☒.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages ☒
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received ☒
16. Other _____

University
Microfilms
International

PREVIEW

Listlessness is Better than Laziness

**An Algorithm that Transforms Applicative
Programs to Eliminate Intermediate Lists**

Philip Lee Wadler
August 1984

**Carnegie-Mellon University
Computer Science Department**

**Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy**

Copyright © 1985 Philip Lee Wadler

This research was sponsored by the U. S. Army under Contract DAAK-80-C-0573, and by the National Science Foundation.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U. S. Government.

Carnegie-Mellon University

Mellon College of Science

THESIS

submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Title LISTLESSNESS IS BETTER THAN LAZINESS

Presented By Philip Lee Wadler

Accepted by the Department of Computer Science

Al Haberman 15 Aug 84
Major Professor Date

Al Haberman 15 Aug 84
Department Head Date

Approved By Robert F. Schen 21 Jan. 1986
Dean Date

Table of Contents

Abstract	1
Acknowledgements	3
Preface	5
1. Introduction	9
1.1. Listful is tasteful	11
1.2. Listful is wasteful	13
1.2.1. Intermediate lists	13
1.2.2. Multiple traversal	14
1.2.3. Unneeded computation	15
1.2.4. Function call overhead	15
1.3. Listless is wasteless	16
1.4. The listless transformer	18
1.5. Outline of the thesis	21
2. Context	23
2.1. Applicative programming	23
2.1.1. History	23
2.1.2. Languages	24
2.1.3. Operators	25
2.1.4. Type checking	25
2.1.5. Lazy evaluation	25
2.1.6. Inefficiency	26
2.1.7. Unsuitable domains	27
2.1.8. Suitable domains	28
2.2. Program transformation	29
2.2.1. UF transformations	29
2.2.2. Schema-based transformations	30
2.2.3. Loop merging	31
2.2.4. Other related work	32
2.3. Imperative languages and coroutines	33
2.4. Special hardware	33
3. Language	35
3.1. Terms and application	36
3.2. Functions	36
3.3. Structures and constructors	37
3.4. Lists, booleans, and records	38
3.5. Primitives and constants	40
3.6. Lazy evaluation	41
3.7. Infinite structures	42

3.8. Eager evaluation	42
3.9. Lambda, case, let, and where expressions	44
4. Evaluation	47
4.1. Abstract syntax	47
4.2. Maps, matching, and substitution	48
4.3. Normal representation	48
4.4. The evaluator	49
4.5. Call-by-need and call-by-name	51
4.6. Irreducible terms	53
5. Listlessness	55
5.1. Preliminary definitions	55
5.2. Listless form	56
5.3. Compiling listless form	60
5.4. Eager evaluation of listless form	61
5.5. Tail recursion modulo cons	63
6. Transformation: Overview	69
6.1. An example	69
6.2. Another example	74
7. Transformation: Details	83
7.1. Symbolic evaluation	83
7.1.1. Bomb sets	83
7.1.2. Primitives	85
7.1.3. Matching	86
7.1.4. Errors	88
7.1.5. The algorithm	88
7.1.6. Lemmas	90
7.2. The listless transformer	91
7.2.1. And-or graphs and solution sets	91
7.2.2. Rewrite rules	93
7.2.3. Folding and matching	95
7.2.4. The algorithm	96
8. Transformation: Annotations	101
8.1. Languid evaluation	102
8.1.1. Motivation	102
8.1.2. First modification	104
8.1.3. Using annotations to make languid evaluation behave	105
8.1.4. Second modification	106
8.2. Generalization	106
8.2.1. Motivation	106
8.2.2. Method	109
8.2.3. Modification	110
8.2.4. Generalization and <i>let^l-reduce</i>	112
8.2.5. Automating eager annotation	112
8.2.6. Related work	116
9. Examples	119
9.1. Merge	119
9.2. Peano arithmetic	121

TABLE OF CONTENTS

III

9.3. Breaking a file into lines and fields	121
9.3.1. Functions for lines manipulating files	121
9.3.2. Definition of <i>lines</i> and <i>fields</i> : an example of listful style	127
9.3.3. Equality and case analysis	128
9.3.4. Transformation to listless form	129
9.4. The telegram problem	130
9.4.1. The problem	130
9.4.2. Equality and case analysis	138
9.4.3. Other solutions	139
9.4.4. The transformed program	139
9.5. Implementation notes	141
10. Conclusions	145
10.1. Contributions of the thesis	145
10.2. Future work	146
Function definitions	147
References	149

Appendix: Bounded Evaluation and the Listless Machine

1. Introduction	1
2. Language	2
3. Bounded Evaluation	3
4. Evaluation Graphs	11
5. Listless Machines	19
6. The Listless Transformer	32
7. Relation to Body of Thesis	47
8. Conclusion	54

PREVIEW

List of Figures

Figure 1-1: Listless and non-listless functions	17
Figure 1-2: Definition of <i>sum-squares</i> in listful style.	19
Figure 1-3: Definition of <i>sum-squares</i> in listless form.	19
Figure 4-1: Abstract syntax of Toy.	47
Figure 4-2: Trace of evaluation of <i>map square</i> (upto 1 2)	52
Figure 5-1: Summary of listless form	58
Figure 5-2: Definitions of <i>map-square</i> and <i>split-odd</i> .	59
Figure 5-3: Definitions of <i>map-square</i> and <i>split-odd</i> in listless form.	59
Figure 5-4: Trmc applied to <i>map-square</i> : adding an output parameter.	65
Figure 5-5: Trmc applied to <i>map-square</i> : doing the tail recursion.	65
Figure 5-6: Trmc applied to <i>map-square</i> : cleaning up.	66
Figure 5-7: Trmc applied to <i>split-odd</i> : doing the tail recursion.	67
Figure 6-1: Beginning the search of the and-or graph	76
Figure 6-2: First level of and-or graph	78
Figure 6-3: First and second level of and-or graph	79
Figure 6-4: Definitions corresponding to graph of figure 6-3	80
Figure 8-1: Listless transformation of <i>insert</i> , with lazy evaluation	103
Figure 8-2: Listless transformation of <i>insert</i> , with languid evaluation	104
Figure 8-3: Listless transformation of <i>needs-generalized</i> , without generalization	107
Figure 8-4: Listless transformation of <i>needs-generalized</i> , with generalization	108
Figure 8-5: Definition of <i>sum</i> .	113
Figure 8-6: Listless transformation of <i>sum</i> , without generalization	113
Figure 8-7: Listless transformation of <i>sum</i> , with generalization	113
Figure 8-8: Definition of <i>insert-sort</i>	114
Figure 8-9: Listless transformation of <i>insert-sort</i>	114
Figure 8-10: Definition of <i>two-sums</i>	117
Figure 8-11: Two ways of annotating <i>two-sums</i>	117
Figure 8-12: A less clear way of annotating <i>two-sums</i>	117
Figure 9-1: Definition of <i>merge</i>	122
Figure 9-2: Listless definition of <i>merge</i>	122
Figure 9-3: State-diagram of <i>merge</i>	123
Figure 9-4: Definition of <i>last</i>	124
Figure 9-5: Listless definition of <i>last</i>	124
Figure 9-6: State diagram of <i>last</i>	125
Figure 9-7: Definitions of Unix utility functions	131
Figure 9-8: Character operations for Unix utility functions	131
Figure 9-9: Listless definition of <i>normalize</i>	132
Figure 9-10: State-diagram of <i>normalize</i>	133
Figure 9-11: High-level state-diagram of <i>normalize</i>	134

Figure 9-12: Listless definition of <i>wc</i>	135
Figure 9-13: State-diagram of <i>wc</i>	136
Figure 9-14: Original definition of h_1 of <i>normalize</i>	137
Figure 9-15: Definition of functions for Telegram Problem	142
Figure 9-16: Character operations for Telegram Problem	142
Figure 9-17: High-level state-diagram of <i>telegram</i>	143
Figure 9-18: Low-level state-diagram of <i>telegram</i>	144

PREVIEW

Abstract

This thesis is about a style of applicative programming, and a program transformation method that makes programs written in the style more efficient. It concentrates on a single, important source of clarity and inefficiency in applicative programs: the use of structures to communicate between components of a program.

One means of increasing clarity is to divide a program into independent components. In *listful style* these components communicate by passing intermediate data structures. Usually, but not always, these structures are lists; hence the name. Listful programs are typically composed out of standard list-manipulating functions, such as *map*, which applies a function to each element of a list to form a new list. Several researchers have advocated a programming style using such functions, but few have emphasized the importance of intermediate lists.

Unfortunately, listful style can be very inefficient. This is due to the space required to store the intermediate lists, and the time required to traverse them. Transforming a program into *listless form*, in which no intermediate lists (or other structures) are allowed, eliminates all inefficiency caused by listful style. The transformation method is an algorithm for applying Burstall and Darlington style transformations, based on a simple case analysis. This *listless transformer* is completely automatic. It might be incorporated as part of an optimizing compiler, or as part of a more sophisticated transformation system.

Several researchers have advocated the use of *lazy evaluation* with applicative languages. Lazy evaluation makes it possible to write certain elegant programs that would be undefined or impractical under eager (normal) evaluation. However, because of its high overhead cost, lazy evaluation is seldom used. Among other things, the listless transformer converts programs with lazy evaluation semantics into programs that can be executed by an eager evaluator. This "lazy evaluation at compile-time" makes available the advantages of lazy evaluation without the costs. Thus, when it is applicable, the listless transformer is superior to lazy evaluation.

However, the listless transformer is not always applicable. Given a function definition, the transformer will succeed in finding an equivalent definition in listless form if and only if the original definition can be evaluated using bounded intermediate storage. If the function cannot be converted to listless form, the user may annotate the program to indicate places where intermediate structures may appear.

"Why, anybody can have a brain. That's a very mediocre commodity. Every pusillanimous creature that crawls on the earth, or slinks through slimy seas has a brain! Back where I come from we have Universities - seats of great learning - where men go to become great thinkers. And when they come out, they think deep thoughts, and with no more brains than you have. But - they have one thing you haven't got -- a diploma!"

-- The Wizard, to the Scarecrow

(From the MGM release "The Wizard of Oz", copyright 1939 by Loew's Incorporated. Copyright renewed 1966 by Metro-Goldwyn-Meyer Inc.)

Acknowledgements

I'd like to thank the members of my thesis committee, Nico Habermann, Guy Steele, and Bill Scherlis, for their help. All three shared the tasks of asking me the right questions, regenerating my flagging enthusiasm, and keeping me intellectually honest.

Jim Morris, my external committee member, first introduced me to functional programming and lazy evaluation. I have great admiration for the way in which he brings abstract ideas to life. It was he who coined the term "lazy evaluation", without which this thesis would have a less interesting title.

The Programming Research Group at Oxford was my home when the appendix to this thesis was written. I am indebted to Bernard Sufrin and Tony Hoare for allowing me the freedom to do this.

Trina and Nathaniel Borenstein provided invaluable help when I had to prepare the final version of this thesis remotely from England.

Everyone who's been here says that the Computer Science department at C-MU provides a superb academic and social environment for research. They're right. Everyone who hasn't been here says that Pittsburgh must be an awful place to live. They're wrong. My thanks to C-MU CS and Pgh.

I want to thank all the friends who have helped keep me happy, sane, and growing during this time, especially: Nathaniel Borenstein, Trina Borenstein, Valerie Henderson, Christine Hohman, Steven Miller, Jill Smudski.

My uncle and aunt, Herb and Cele Sorkin, and my grandfather, Jack Wadler, provided me with a second home in New York, for which I am deeply grateful.

This thesis is dedicated to my mother Marlene Wadler and my sister Janyce Wadler.

PREVIEW

Preface

This thesis is about a style of functional programming, and a transformation method that automatically optimizes programs written in this style. The thesis is written in two parts. The body of the thesis introduces the style of programming, discusses one version of the transformer, and presents some examples. The appendix presents a formal characterization of the class of programs that can be transformed in this way, and presents a simplified version of the transformer and a proof of its correctness.

In the best of all possible worlds, the two parts of this thesis would be integrated into a single presentation. Time considerations have prevented doing this, and I beg the reader's pardon for any difficulties this may cause. This preface attempts to lessen the problem, by describing the relationship between the two parts, and providing a guide for reading the thesis.

In the course of proving the correctness of the transformer, I achieved a much better understanding of how it works. (This supports the claim that "theoretical" concerns such as formalism often lead to practical benefits as well.) Thus, the appendix provides a description of the transformer that is perhaps more intuitive and easier to understand. On the other hand, the transformer described in the body is an extension of the simple transformer described in the appendix; the extensions are necessary for dealing with the examples discussed. Further, the description of the transformer in the body is more detailed, and follows more closely the implementation used to test the examples.

One method of reading the thesis is as follows. Chapter 1B provides an overall introduction, and a description of the style of programming (listful style) that the thesis is concerned with.¹ Chapter 2B discusses background and related work. Chapter 3B introduces the language used in the thesis. The reader may then skip forward and read the appendix, which assumes no knowledge of body of the thesis except for chapter 1B. After the appendix, the reader may skip back and read the examples in chapter 9B and the conclusions in chapter 10B. (The examples use a more complicated formalism than that in the appendix, but the information in this preface and chapter 7A should allow the reader to follow them.) Finally, the reader interested in more details can look at the development of the transformer in chapters 4B to 8B.

The remainder of this section discusses in more detail the relationship between the material in the body and the appendix. Further aspects of the relationship are discussed in chapter 7A.

¹Within this preface, chapters in the body will be followed by B, and chapters in the appendix will be followed by A. Thus, chapter 1B is the first chapter in the body.

Language

Chapter 3B introduces the language, Toy, used in the body. Chapter 4B describes an interpreter for Toy, as a way of giving a more rigorous definition of Toy, and because this interpreter is later modified to form a part of the transformer (the symbolic evaluator).

Chapter 2A describes the language, T, used in the appendix. T is a subset of Toy. Chapter 2A also describes an evaluation model for T in terms of reduction. The model is basically simple, but some complication is needed to define normal-order reduction for equations that contain constructors on the left-hand side. This reduction model serves as the basis of the development in chapters 3A and 4A.

Thus, Toy is described by an interpreter model, and T is described by a reduction model. In each case, the model determines the approach taken in the succeeding development. The reduction model has better theoretical foundations (e.g., the Church-Rosser theorem), and this aids the more formal approach of the appendix.

The most important difference between Toy and T is that Toy contains *primitives* (that is, data types such as integers and operations on them such as addition), whereas T does not. Of course, one could implement integers in T using, e.g., Peano arithmetic. But this would not take into account the fact that in practice computers represent integers in a special way, and have special hardware for operating on them. Including primitives in Toy greatly increases its practicality.

Chapter 3B is more tutorial than chapter 2A, so it is more suitable as an introduction. But chapter 2A should also be read, since it is a necessary prerequisite for chapters 3A and 4A.

Bounded Evaluation

Chapter 3A describes bounded evaluation. Roughly speaking, a program is *subject to bounded evaluation* (b.e.) if it can be evaluated in constant bounded space, not counting space used by the input or output. More precisely, the notion of a *partition* is defined. A partition breaks a term into an input part, an internal part, and an output part. Reduction on partitions is defined, analogous to reduction on terms. A program is b.e. if there exists a constant N such that for every input there exists a partition reduction to normal form where the size of the internal part of each partition is bounded by N .

In the body of the thesis, there is no formal concept equivalent to bounded evaluation. There is only an informal suggestion that the programs of interest can be evaluated in bounded space.

Chapter 4A describes *evaluation graphs*. It is proved that a program is b.e. if and only if it has a finite evaluation graph. Further, it is proved that if a program has an evaluation graph, then it must have an evaluation graph with certain properties (canonical, useful, deterministic).

Again, there is no equivalent to the material in chapter 4A in the body.

Graph programs and listless form

Chapter 5A describes *graph programs*, which are simply a program-like notation for deterministic evaluation graphs, and *listless machines*, which can execute such programs. Listless machines only access input structures and create output structures, hence they use no intermediate lists.

Chapter 5B defines *listless form*, which is a subset of Toy. Programs in listless form only access input structures and create output structures, hence they use no intermediate lists. (The situation is slightly more complicated than this, because programs in listless form do use intermediate structures to pack arguments and results of functions and primitives; but this is different from using intermediate lists.)

Clearly, graph programs and listless form are closely related. In particular, input steps in graph programs correspond to *case*-expressions in listless form, and output steps in graph programs correspond to *where*-expressions in listless form. If graph programs were extended to include primitive steps (as discussed in section 7.4A), these would correspond to *let*-expressions in listless form.

There are some important differences between graph programs and listless form. First, the definition of graph programs follows naturally from the definition of evaluation graphs (and the results in chapter 4A about such graphs). In contrast, the definition of listless form may at first sight appear to be a bit arbitrary.

Second, graph programs are in a form that can easily be executed by a listless machine. In contrast, in order to execute programs in listless form efficiently, one must have a compiler that recognizes certain patterns of recursive function calls (namely, calls that are tail recursive modulo cons) and compiles them efficiently.

Third, listless form is a subset of Toy, so the transformer in the body performs source-to-source transformation in one language, Toy. In contrast, graph programs are not programs in T, so the transformer in the appendix transforms from one language (T) into another (graph programs). The greater simplicity of graph programs suggests, in retrospect, that trying to view the listless transformer as a transformer from Toy into Toy was a mistake.

The transformer

In the body, chapter 6B introduces the workings of the transformer through two examples. Chapter 7B describes the transformer algorithm in detail. The appendix describes a somewhat simpler transformer, in chapter 6A.

Both transformers are very similar in structure. Input steps in the appendix transformer correspond to *case*-extraction in the body transformer, and output steps in the appendix transformer correspond to *where*-extraction in the body transformer. If the appendix transformer were extended to include primitive steps (as discussed in section 7.4A), these would correspond to *let*-extraction in the body transformer. Note that the relationship between the two transformers is very similar to the relationship between program graphs and listless form.

Also, the routines used for breadth-first search and for evaluating expressions are very similar in both transformers.

The correctness of the transformer in chapter 6A follows almost directly from the results proved in chapter 4A. The structure of the transformer is simplified by transforming directly into an evaluation graph rather than into listless form. The close relationship to the theory helps to keep the structure of the transformer simple and clear.

However, the transformer in chapter 6B and 7B is much more powerful, because it also deals with primitives (e.g., data types such as integers and operations such as addition). In particular, rewrite lists were introduced in order to deal with primitives. As mentioned in chapter 7.4A, it appears that the transformer in the appendix could be extended to deal with primitives.

Chapter 8B describes some extensions to the transformer. Again, it appears that these extensions could be adapted to the transformer in the appendix, at the cost of further complication.

Examples

Chapter 9B presents some examples of use of the transformer, including some examples inspired by Unix and the Telegraph Problem. These examples suggest the practical power and limits of the transformer. The examples depend on the use of primitives. This shows that the transformer presented in the appendix, although useful because it clarifies the underlying ideas, must be extended to include primitives before it is of practical value.

Chapter 1 Introduction

One eternal frustration of programmers might be expressed like this: cleanliness is and is not next to godliness. Cleanly written programs have the desirable properties of being easier to understand, show correct, and modify. But a divinely clean program can also be hellishly inefficient.

This is no accident. A major design technique for achieving clarity is *modularity*: breaking a problem into independent components. But modularity can often lead to inefficiency, because of the overhead of communicating between components, and because it precludes potential optimizations across components boundaries. Such optimizations include eliminating computations repeated in multiple components, and one component taking advantage of the representation used by another.

According to this analysis, some current trends in programming methodology (such as structured programming) are inherently limited, because they attempt to develop a program that is simultaneously clear and efficient. This thesis is concerned with an alternate methodology that has attracted much interest: Use *applicative programming* to write clean (but inefficient) programs, and use *program transformation* to convert them to more efficient (but less clean) equivalents. (See [Feather 79] and [Scherlis 80] for more detailed versions of these arguments.)

Applicative programming is simply the doctrine "side effects should be avoided" carried to its extreme: once bound, the value of a variable is never changed. Examples include pure Lisp, the λ -calculus, Backus' functional language FP, and a large subset of APL.

Applicative programming is well suited to the program transformation methodology for two reasons. First, it is good for writing clear initial programs, because it supports a powerful and elegant programming style. Second, it is good for performing transformations, because of its nice mathematical properties.

Typically, program transformation consists of a sequence of steps, each of which is *source-to-source* (the program is rewritten in the same high-level language) and *equivalence preserving* (the semantics of the program does not change). The most widely-known transformation method is the UF (fold-unfold) method, which was developed independently by Burstall and Darlington [Burstall 77a] and Manna and Waldinger [Manna 75, Manna 79].

Neither applicative programming nor program transformation has achieved widespread, practical use. Applicative programs for realistic tasks can be notoriously inefficient. Transforming such programs requires either intractable heuristic searches or tedious human guidance. (Program verification and automated theorem proving suffer from similar problems.) Although there have been promising results, many research problems remain.

This thesis is concerned with one kind of applicative programming style, and a transformation technique that optimizes programs written in this style. In this style, the components of a program communicate by passing large intermediate data structures. Since these structures are often lists, I call it *listful style*. Listful style is an important source of clarity in applicative programs, and also a major cause of inefficiency. Transforming programs into *listless form*, where the intermediate structures are eliminated, removes the inefficiency (at least from this source). The transformation method is an algorithm for applying Burstall and Darlington style transformations, based on a simple case analysis. This *listless transformer* is completely automatic. It might be incorporated as part of an optimizing compiler, or as part of a more sophisticated transformation system.

Incorporating the transformer in a compiler should allow a wider range of applicative programs to be compiled efficiently. This, in turn, should contribute to gaining practical experience that can be used to evaluate the utility of applicative programming. (I believe that there are too many claims made about the desirability of applicative programming, and too few tests of these claims in practice.)

Incorporating the listless transformer into a more sophisticated transformation system should free the user from various "low-level" transformation concerns. The listless transformer is concerned mainly with issues of "plumbing": how values are communicated between the components of a program. Providing tools which deal with this concern frees the user to concentrate on deeper issues of algorithm design.

One technique that has been advocated in connection with applicative programming is *lazy evaluation* [Henderson 76, Friedman 76]. Lazy evaluation is so named because expressions are not evaluated until needed. Typically in listful style, one function produces a list that another function consumes. Lazy evaluation causes the two functions to behave like coroutines, where the producer computes each element only when the consumer requires it.

Lazy evaluation semantics has several advantages over the eager semantics assumed by most programming languages. These advantages are particularly well suited to listful style, as will be explained later. For this reason, the thesis assumes a lazy evaluation semantics for user programs.

On the other hand, lazy evaluation implementations have both advantages and disadvantages with respect to traditional eager implementations. The major disadvantage is a high overhead cost, which usually outweighs the advantages of lazy evaluation.

As will be seen, the listless transformer eliminates all of the inefficiency costs of listful style,