Technical Report 1251

Program Improvement by Automatic Redistribution of Intermediate Results

Robert Joseph Hall

MIT Artificial Intelligence Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1990	technical report

4. TITLE AND SUBTITLE

Program Improvement by Automatic Redistribution of Intermediate Results

5. FUNDING NUMBERS

IRI-8616644
N00014-88-K-0487

6. AUTHOR(S)

Robert Joseph Hall

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

AI-TR 1251

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Information Systems
Arlington, Virginia 22217

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

None

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution of this document is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The introduction of function sharing, a universal principle of design optimization, allows eliminating costly structure by using other structure to perform the function of the original. This approach is particularly applicable to eliminating the inefficiencies that arise naturally in reusing general software components in specific contexts. In this research, I have studied a restricted form of function sharing, redistribution of intermediate results, which is characterized by the fact that each optimization can be implemented using only introduction of new local variables, additional function arguments, and additional function return values. Examples are given showing that these optimizations can capture many well-known types of optimizations, such as copy elimination, generalized loop fusion, identical value sharing, and data

(continued on back)

14. SUBJECT TERMS (key words)

artificial intelligence
function sharing
software engineering
program optimization
reuse

15. NUMBER OF PAGES
275

16. PRICE CODE
$14.00

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED

Block 13 continued:

invariant suspension. I have decomposed the problem into two subproblems: first search for a small set of plausible candidates, then certify the correctness of each candidate. My system solves the search/screening subproblem while leaving certification up to one of a variety of possible external approaches. The system performs search and screening by automatically deriving operational filtering predicates from teleological input information obtainable as a natural by-product of the design process. The system uses a novel form of explanation-based generalization to derive these filtering predicates from a program correctness proof. I also discuss how the research sheds light on the certification problem and holds promise for future practical applications.

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I. Technical Report 1251 December, 1990

Program Improvement by Automatic Redistribution of Intermediate Results

by

Robert Joseph Hall

This report was submitted in December, 1990 to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

© Massachusetts Institute of Technology, 1991. All rights reserved.

Program Improvement by Automatic Redistribution of Intermediate Results

by

Robert Joseph Hall

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering and Computer Science.

Abstract

The introduction of function sharing, a universal principle of design optimization, allows eliminating costly structure by using other structure to perform the function of the original. This approach is particularly applicable to eliminating the inefficiencies that arise naturally in reusing general software components in specific contexts. In this research, I have studied a restricted form of function sharing, redistribution of intermediate results, which is characterized by the fact that each optimization can be implemented using only introduction of new local variables, additional function arguments, and additional function return values. Examples are given showing that these optimizations can capture many well-known types of optimizations, such as copy elimination, generalized loop fusion, identical value sharing, and data invariant suspension. I have decomposed the problem into two subproblems: first search for a small set of plausible candidates, then certify the correctness of each candidate. My system solves the search/screening subproblem while leaving certification up to one of a variety of possible external approaches. The system performs search and screening by automatically deriving operational filtering predicates from teleological input information obtainable as a natural by-product of the design process. The system uses a novel form of explanation-based generalization to derive these filtering predicates from a program correctness proof. I also discuss how the research sheds light on the certification problem and holds promise for future practical applications.

Thesis Supervisor: Dr. Charles Rich
Title: Principal Research Scientist

To I-Fan and Kelsey

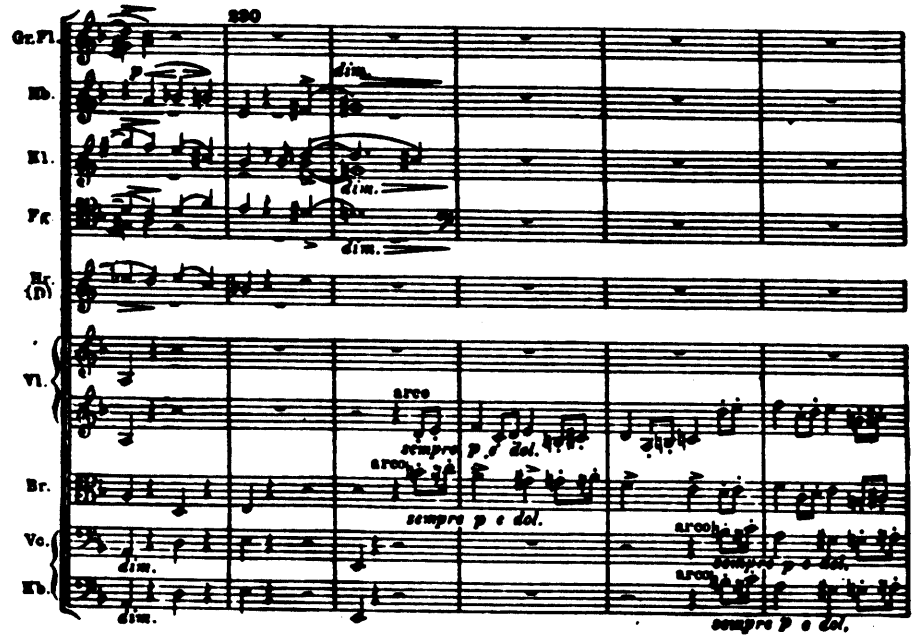
Acknowledgements

Special thanks are due my thesis adviser, Chuck Rich, for many valuable discussions, encouragement, and useful criticism. My thesis readers, John Guttag and Tomas Lozano-Perez, also contributed greatly to the research both in the conceptual and writing stages. I would like to extend thanks as well to Dick Waters, Doug Smith, and Peter Szolovits for discussing the issues presented here, and to David Steier, Ted Linden, and Michael Lowry for reading and commenting on drafts of portions of the document. Thanks to all my friends at the MIT AI Laboratory who, over the years, have contributed in various ways to making my experience there fun and enriching.

On the personal side, I am blessed with the best wife and daughter of all time; I-Fan and Kelsey have provided me with loving support, friendship, and a good excuse to think about the Blocks World again. I would like also to sincerely thank my parents for the many different forms of support they have given me over the years. Thanks also to my brother for the Big Game tickets and all the long-distance phone calls.

Portions of Chapters 1, 2, and 14 appear in my paper "Program Improvement by Automatic Redistribution of Intermediate Results: An Overview" which appears in the book *Automating Software Design*, M. Lowry & R. McCartney, eds. (Menlo Park and Cambridge: AAAI Press / The MIT Press). They are reproduced here with permission.

This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. The lab's artificial intelligence research is supported in part by the National Science Foundation under grant IRI-8616644, Advanced Research Projects Agency of the Department of Defense under Naval Research contract N00014-88-K-0487, IBM Corporation, NYNEX Corporation, and Siemens Corporation. The views and conclusions contained in this document are those of the author and do not necessarily represent the policies, expressed or implied, of these organizations.



Frontispiece. Function sharing in Brahms' *Tragic Overture* (Brahms, 1880). The viola's notes on beat 4 of bar 231 and beat 1 of bar 232 serve both to end the previous phrase in the clarinet and also to begin the next phrase.

Contents

1 Introduction and Overview **12**

1.1 Function Sharing 15

1.1.1 Examples in Design 15

1.1.2 The Adaptation Problem 16

1.2 Significance of the Work 17

1.3 An Example 18

1.4 Optimization Phenomena Captured by Redistribution 27

1.5 Research Overview 28

1.5.1 Key Ideas in the Implementation 28

Key Idea 1: Additional Design Information 29

Optimization Invariants. 30

Explanations. 30

Key Idea 2: Using Incomplete Design Information . . . 31

Relative Optimization Invariants. 31

Proofs of Quasi-specifications 31

Key Idea 3: Candidate Generation 33

Key Idea 4: Candidate Screening 33

Key Idea 5: Approaches to Correctness 34

Compile-time certification. 35

Run-time debugging. 35

1.5.2 Experiments 36

1.5.3 Evaluation 36

1.6 Reader's Guide 37

2 Optimization Phenomena **39**

2.1 Identical-value Redistributions 39

2.2 Copy Elimination 41

2.3 Data Invariant Suspension 41
2.4 Generalized Loop Fusion 43

I Representations and Operations 46

3 Program Structure 47
3.1 Dataflow Programs 49
3.1.1 Program Elements 51
3.1.2 Programs have eager dataflow semantics 53
3.1.3 Conditional execution 53
3.1.4 Showing box implementations 56
3.1.5 Virtual structure 58
3.1.6 Pathnames 58
3.1.7 The causality relation on program elements 60
3.2 Modeling Issues 61
3.2.1 Stores model side effects 61
3.2.2 The series data type models iteration 62
3.2.3 Syntactic program restrictions for modeling 63
Store restrictions. 63
Series restrictions. 65
3.3 Redistributions 66
3.4 Executing Programs 68
3.4.1 Structure of the Interpreter 70
3.4.2 Computational Primitives 71
3.4.3 Program traces 71
3.5 Programs versus Plans 72

4 Program Function 74
4.1 Terms and Clauses 75
4.1.1 Syntax 75
4.1.2 Semantics 76
4.1.3 Ground Evaluation of Clauses 77
Universal Instantiation. 77
Term Evaluation. 78
4.2 Optimization Invariants 79
4.3 An Optimization Invariant Example 81

5 Proofs Connect Structure to Function **83**
5.1 Proof Syntax 83
5.1.1 Proof Nodes: Leaves 84
:TAUTOLOGY Nodes 84
:AXIOM Nodes 85
:FORWARD Nodes 85
:DEFINITION Nodes 85
:PROGRAM-STRUCTURE Nodes 85
5.1.2 Proof Nodes: Internal 87
:SUBSUMPTION Nodes 87
:RESOLUTION Nodes 87
:INSTANTIATION Nodes 87
:DEFINITION-APPLICATION Nodes 87
5.2 Proof Example 88
5.3 Weakened Relative Conditions 90
5.4 Proof-tree Restructuring 95

II The Optimization Algorithms **99**

6 Overview of the Algorithms **100**

7 Candidate Generation **104**
7.1 The Problem 104
7.2 The Solution Method 106
7.2.1 Pseudo-code Description 107
7.2.2 Cost Estimation 109
Program-cost 110
Use-cost 111
Cost Comparison 112
Shortcomings 112
7.2.3 Syntactic Pruning 113
7.3 Discussion 114

8 Candidate Screening I: Inv-Screen **116**
8.1 Pseudo-code Description 116
8.2 Discussion 117

8.2.1 Optimization Power . . . 118
8.2.2 Test Input Evaluations . . . 118
8.2.3 Certification Difficulties . . . 119
8.2.4 Program Checking Difficulty . . . 119

9 Candidate Screening II: EB-Screen **121**
9.1 Pseudo-code Description . . . 121
9.2 Target Condition Theory I: Single Redistributions . . . 122
9.2.1 Computing Target Conditions from Proofs . . . 124
9.2.2 Extensions to Handle Hierarchical Structure . . . 128
9.2.3 Evaluating Target Conditions . . . 131
9.2.4 Quality of Proof Structure . . . 131
9.2.5 Proof Restructuring . . . 133
9.3 Target Condition Theory II: Multiple Redistributions . . . 134
9.4 Target Condition Theory III: Recursive Redistributions . . . 137
9.4.1 The Bad News . . . 138
9.5 Applications to Checking Redistributions . . . 140
9.5.1 Alternative Checking Strategies . . . 142
9.6 Discussion . . . 144
9.6.1 Comparison of Strategies . . . 144
9.6.2 Wrong Answers . . . 145
9.6.3 Routine Redistributions . . . 147
9.6.4 Do We Need Recursion? . . . 149

10 Proofs of Quasi-specifications **151**
10.1 The Default Quasi-specification . . . 151
10.2 Quasi-specifications for List Programs . . . 152
10.3 Quasi-specifications for Set Programs . . . 154
10.4 Proved Quasi-specifications in General . . . 155

III Analysis and Discussion **157**

11 Comparison of Algorithms **158**
11.1 Power . . . 158
11.2 Time and Space Costs . . . 159
11.2.1 Run Time . . . 159

11.2.2 Space 161
11.3 Routine Optimization 161

12 Supplying Design Information **163**
12.1 Optimization Invariants 163
12.2 Explanations (Proofs) 164
12.3 Representative Test Inputs 164
12.4 Program Structure 165

13 Certification **168**
13.1 Compile-time Certification 168
13.2 Run-time Certification 170

14 Literature Review **172**
14.1 Program Improvement and Redesign 172
14.2 Explanation-based Generalization 177

15 Conclusions **179**
15.1 Contributions 180
15.2 Limitations 182
15.3 Future Work and Potential Applications 183

16 References **186**

IV Appendices **190**

A Experimental Domain Knowledge **191**
A.1 Primitive Data Types 191
A.2 Computational Primitives 193
A.3 Noncomputational Functions 193
A.4 Universal Instantiators 194
A.5 Abstract Data Type Implementations 195
A.5.1 LR1 Lists 195
A.5.2 SR Sets 197

B Glossary **198**

C Selected Examples **213**
C.1 Example Programs On Numbers 214
C.1.1 POLY 215
C.1.2 Fibonacci Numbers: FIB 222
C.2 Example Programs On LR1 Lists 227
C.2.1 LR1-CONCAT-C+D (APPEND) 227
C.2.2 LR1-REV (MY-REVERSE) 229
C.2.3 LR1-REM+REVAPPEND 230
C.2.4 LR1-REM+APPEND 234
C.2.5 LR1-MSORT (MERGE-SORT) 238
C.2.6 LR1-SUBSTITUTE+EQUAL? 246
C.3 Example Program On SR Sets 249
C.3.1 SR-ELT?+UNION 249

D Proof Restructuring **257**
D.1 Preprocessing 258
D.1.1 Eliminate Macro Nodes 258
D.1.2 Instantiation Pushing 258
D.1.3 Non-T Tautology Elimination 259
D.1.4 Subsumption Pulling 260
D.1.5 Local Inefficiencies 261
D.1.6 Preprocessing Pseudo-code 262
D.2 The Restructuring Algorithm 262
D.2.1 Removing Unneeded Marked Leaves 262
D.2.2 Pushing Marked Leaves Down 262
D.2.3 Tree Restructuring Pseudo-code 264

E Programmability versus Checkability **265**

F Series Subtleties **270**
F.1 The Current Approach 272
F.2 A More General Approach 274

Chapter 1

Introduction and Overview

General tools make it easy to solve many different problems. Unfortunately, using a general tool to solve a specific problem often yields an inefficient solution. The reason is simple: the general tool was designed to handle harder problems and so may do more work than necessary in a given context.

In some domains of engineering, general tools are used widely. Mechanical engineers, for example, exploit catalogs of standard parts as much as possible to reduce production costs. Since the precise part for the specific job is unlikely to be available in a catalog, the designer must find a standard, more general part that handles the job. Accordingly, mechanical designs often sacrifice performance in order to be able to use low-cost standard parts.

Software engineers, by comparison, seldom reuse standard parts, except low-level ones, such as programming language primitives, library routines, and operating system tools. The benefits of widespread reuse in software engineering are many. It would yield more correct code, because the reused code has already been tested and debugged. Code would be produced faster, because no time is spent on producing reused subroutines. Also, the code would be more readable, because reused components would be modular and well-documented. Readability is an important cost consideration, because debugging and "maintenance" usually dominate the cost of a large software project (Boehm, 1981).

A major reason for the lack of widespread reuse in software engineering is the inefficiency phenomenon discussed above. Software by nature has extremely complex, layered functional hierarchies; hence, using standard parts at many levels of a software design would lead to compounding inefficiencies,

resulting in unacceptable system performance. Thus, programmers currently tailor their code to the specific task, exploiting constraints in the problem context to gain back efficiency. Unfortunately, a tailored solution has many signficant drawbacks: it takes longer to write the code; the resulting programs tend to have tangled and unclear structure; the necessity of producing more new code increases the potential for error; and unreadable programs are hard for others to understand and modify.

The primary contribution of this research has been to invent a way to automatically tailor general data and procedural abstractions to their contexts of use, thereby ameliorating the general-tool-inefficiency problem in software reuse. I have implemented a prototype system capable of automatically performing powerful-but-routine tailoring operations based on the fundamental engineering design principle of *function sharing*. The system is fully automatic in that it runs to completion fairly quickly without human intervention. Full automation has a price, however, since the problem of verifying that a program optimization preserves correctness is in general uncomputable.[1] The optimizations found by the system are guaranteed to preserve correctness only on a set of given test cases; as a result, some form of additional certification must be provided, either at compile-time or run-time.

This *screen/certify methodology* is illustrated in Figure 1.1. It is a fundamental departure from traditional compiler approaches, wherein the safety of a given optimization is guaranteed by virtue of being derived using syntactic transformations, each of which is known to preserve correctness in all situations. There are two primary reasons for this departure. First, there can be other computationally simple methods for demonstrating the safety of a transformation besides simply restricting to sequences of correctness-preserving transformations. For example, a sequence of arbitrary syntactic transformations, none of which preserves correctness alone, may combine to produce an optimization that is simple to prove safe based on the modules' specifications. By explicitly separating the certification and screening subproblems, we allow more flexible approaches to both. The second reason for adopting the methodology is that it allows the encapsulation of the candidate-generation phase of the process for potential use as a separate tool, interesting in its own right. Such a tool would be a kind of "optimization advisor," to be treated as one would a medical expert system: it gives plausible

[1] This follows easily from Rice's Theorem (Hopcroft & Ullman, 1979).

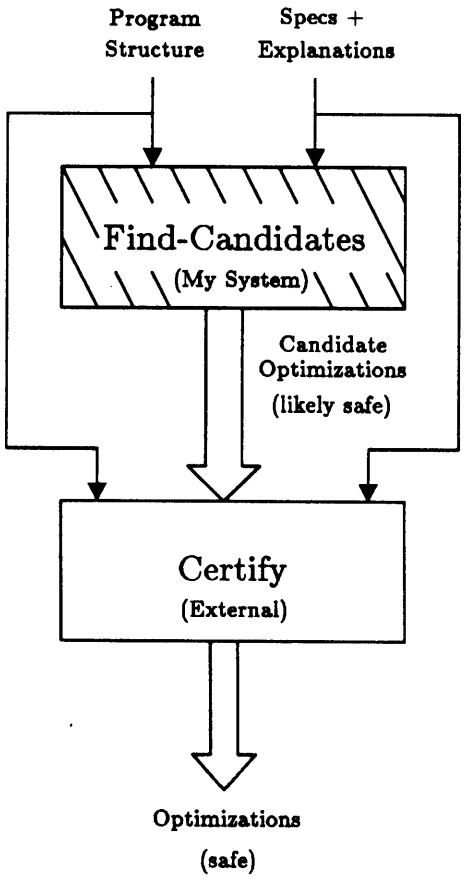


Figure 1.1: The screen/certify methodology applied to program optimization. This diagram illustrates the top-level problem decomposition. It also points out that this research discusses only the implementation of the candidate search phase.

candidates and carries out routine aspects of the optimization process, but leaves the responsibility for safety with the human user, where it currently rests. I will return to these issues in Section 1.5.1, Chapter 13, and in the Conclusion.

1.1 Function Sharing

Function sharing is when one structural component serves more than one purpose in a design. This research attempts to automate the introduction of function sharing into program designs. Many of the insights gained also carry over to other design domains.

1.1.1 Examples in Design

Function sharing is ubiquitous in engineering problem solving. Ulrich (1988) says this:

> If automobiles were designed without function sharing they would be relatively large, heavy, expensive and unreliable. But because elements like the sheet-metal body perform many functions (electrical ground, structural support, aerodynamic faring, weather protection, and aesthetics among others) automobiles can be manufactured relatively inexpensively and can perform relatively well. (Ulrich, 1988, p. 76)

Ulrich's work explores introducing function sharing into designs of mechanical devices. Here are some examples from other domains:

- In electronic design, fanout is a simple example of function sharing. A bus structure is a more complicated example that saves quadratic wiring costs by using a single set of wires to mediate communication among functional units.
- Architects design the walls of buildings to perform several functions at once: structural support of the building, separation of interior spaces, decoration, and sound insulation.

- In software engineering, well-known optimization techniques such as common subexpression elimination, loop fusion, and memoization introduce function sharing in limited ways.

1.1.2 The Adaptation Problem

To introduce function sharing into a design one must solve two problems: find a candidate for sharing, and adapt the design to make it work. Consider the process of improving the design of a popcorn container to serve not only as the shipping package, but also as the cooking vessel in a microwave oven. The original idea is simply the observation that the same container can, in principle, serve both purposes. From there, however, significant problem-solving remains: in addition to its original requirements of strength, food preservation, and low cost, it must also be capable of expanding as the cooked popcorn fills a larger volume; there should be no metal in it (for use in a microwave oven); and it should withstand a larger range of temperatures. Satisfying these additional constraints requires general design problem solving expertise; I will call this the *adaptation problem.*

By contrast, consider going from the initial design of a car, in which there is a ground wire running from the tail light to the battery, to that in which the metal car body is used for this purpose. Almost no additional design is necessary beyond the initial observation that the car body can serve the purpose of the ground wire. In this case, the adaptation problem is trivial.

In this research, I have restricted my attention to those optimizations requiring only trivial adaptation. By this, I mean that the only adaptation considered is the introduction of dataflow from a new *source* (value producer) to a given *target* (value user), together with elimination of any arc from the old source to the target. This type of adaptation can be implemented using only additional local variables, input arguments, and return values as means of communicating a value from its point of computation to its (shared) points of use. No additional subroutine calls are added, and the only changes to subroutine calls are those required to implement data flow, such as adding extra arguments or extra return values. I term this restricted kind of function sharing in software design *redistribution of intermediate results,* or just *redistribution.*

The decision to study only redistribution rather than general function sharing has both advantages and disadvantages. The key advantage is that

it allows me to study the problem of automating the introduction of function sharing into programs without developing a general theory of automated program design. It also drastically limits the search for optimizations by allowing the system to ignore any optimization that adds new computations. Even with this simplification, however, search control is still a problem. (See Section 1.5.1.)

The obvious disadvantage of studying only redistribution is that the system will be able to perform fewer optimizations. Consider, for example, a program whose original design operates on two different representations (say array- and pointer-based) of the same list, and suppose the program requires both representations of the list to be sorted. Such a design is plausible if one reusable component constructs and operates on the array representation and some other reusable component constructs and operates on the pointer-based representation. Moreover, it is unlikely that the sorted array representation can be directly used by the routines requiring a sorted linked list. However, a system similar to mine that was capable of solving this more general adaptation problem could discover the design that converts the sorted array into a sorted linked list and saves it for later use, thus eliminating a costly sort in favor of a linear-time conversion.

1.2 Significance of the Work

The problem of automatic program improvement is of interest to both the artificial intelligence and software engineering communities. From an artificial intelligence perspective, optimization is a key step in the design process; design, in turn, is an important kind of human problem solving. Furthermore, design optimization is difficult enough that it shares a fundamental property with many other problems studied by artificial intelligence; namely, it requires computational solutions that trade off power for tractability. AI researchers may view this research as a case study in the computational tradeoffs and techniques required to implement an intuitively motivated principle of design optimization, *introduction of function sharing*, in the domain of software. Note, however, that while this approach is motivated by intuitions about how humans optimize designs, I have not attempted to rigorously study or duplicate human cognitive behavior. Finally, I believe that many of the insights gained carry over directly to other design domains.

From a software engineering perspective, automated program improvement is important because better optimizers allow programmers to write programs more quickly and clearly, while worrying about less detail. In particular, better optimizers can perform more of the tailoring required to *reuse* software modules among different applications, thereby saving separate development costs. Software engineers may view this research as an attempt to demonstrate an approach to automatic program optimization that (1) captures qualitatively more powerful optimizations than do conventional approaches and (2) is well-suited to facilitating reuse of software components. Note, however, that the research is still in the exploratory stage, so it is too early to make specific claims about the ultimate usefulness of the techniques. In particular, the implemented system has not yet been engineered for maximum performance.

1.3 An Example

This section illustrates how my prototype system can perform powerful-but-routine optimizations currently done by hand. The system has been applied successfully to several (currently, 26) examples from three different programming domains, including the example below (see Section C.2.2 for added detail). I treat a simple example here for brevity, but it nevertheless illustrates the key features of the approach. More complex examples are shown in detail in Appendix C.

The system requires more input than just the traditional source-language structural representation; the system uses information about the program's function and the connection of its structure to its function as well. This additional input is discussed below and in Section 1.5.1.

Suppose you want to write a Common Lisp program to reverse a list (presumably forgetting that the language provides a primitive to do so). You don't want the program to destructively modify the input list, so you decide to use only side-effect-free Lisp primitives in writing the program. Here is your first pass:

```
(DEFUN MY-REVERSE (L)
  "Reverses the input list nondestructively"
  (IF (NULL L)
      NIL
      (APPEND (MY-REVERSE (REST L))
              (LIST (FIRST L)))))
```

This program is clearly correct as long as the input is a finite Lisp list (i.e., a CDR-chain of cons cells ending in NIL). Not only that, but it was constructed quickly out of pre-existing software components, in this case the standard Common Lisp language primitives. You didn't have to implement any of the subroutines, and you didn't have to worry about side effects. Unfortunately, this implementation uses too much time and space to be practical—both are proportional to the square of the length of the input list, because APPEND copies its first input on every recursive invocation of MY-REVERSE.

The implemented system discovers optimizations that turn this inefficient implementation into one using only linear time and space. The optimized program (shown below) creates only one new cons cell per cell of the input list and maintains a tail pointer for the new list as it is constructed.

```
;; returns two values:
;; reversed list and last-cons of reversed list
(DEFUN MY-REVERSE-2 (L)
  "Reverses the input list nondestructively"
  (IF (NULL L)
      (VALUES NIL NIL)
      (MULTIPLE-VALUE-BIND (REVD-TAIL LAST-CONS)
          (MY-REVERSE-2 (REST L))
        (LET ((NEW-LAST (LIST (FIRST L))))
          (VALUES (APPEND-2 REVD-TAIL NEW-LAST LAST-CONS)
                  NEW-LAST)))))

(DEFUN APPEND-2 (L1 L2 LAST-L1)
  "Performs APPEND within MY-REVERSE-2"
  (NCONC-2 L1 L2 LAST-L1))
```

```
(DEFUN NCONC-2 (C1 C2 LAST-C1)
  "NCONC within APPEND-2.
   If C1 is not NIL, then LAST-C1 must be the last-cons of it"
  (IF (NULL C1)
      C2
      (PROGN
        (RPLACD LAST-C1 C2)
        C1)))
```

Applying the system. A fundamental limitation of the power of traditional compiler-based approaches to optimization is the fact that they accept only a program's structure, expressed in a standard programming language. They cannot exploit any freedoms that might exist in the program's specification, simply because there is no way for the programmer to express them. By and large, the optimized program must compute values identical to those computed by the original. Some approaches, however, can exploit limited, implicit specification information such as the knowledge that newly allocated memory cells are all equivalent.

I have explored two generate and test algorithms for discovering redistributions, each incorporating several stages of filtering. IBR (Invariant-Based Redistribution) operates simply by trying out each candidate redistribution and directly evaluating the program's overall correctness using representative test inputs and given *optimization invariants* (defined below). IEBR (Invariant- and Explanation-Based Redistribution), by contrast, incorporates an additional filtering step based on computing an approximate weakest precondition on each program target, its *target condition*, that is sufficient to guarantee overall correctness. That is, any object satisfying the target condition may be connected to the target and the program will remain correct. A source whose value fails to satisfy the target condition of a target in some test case is eliminated from consideration. These two algorithms differ in their required inputs and in their performance characteristics. I will discuss (Section 1.5.2) experiments comparing them later, but for brevity I will demonstrate only IEBR here.

IEBR requires the following inputs:

- A *structural representation* of the main program and all subroutines. Specifically, the program is represented as a dataflow graph as defined and discussed in Chapter 3. Figure 1.2 illustrates the dataflow

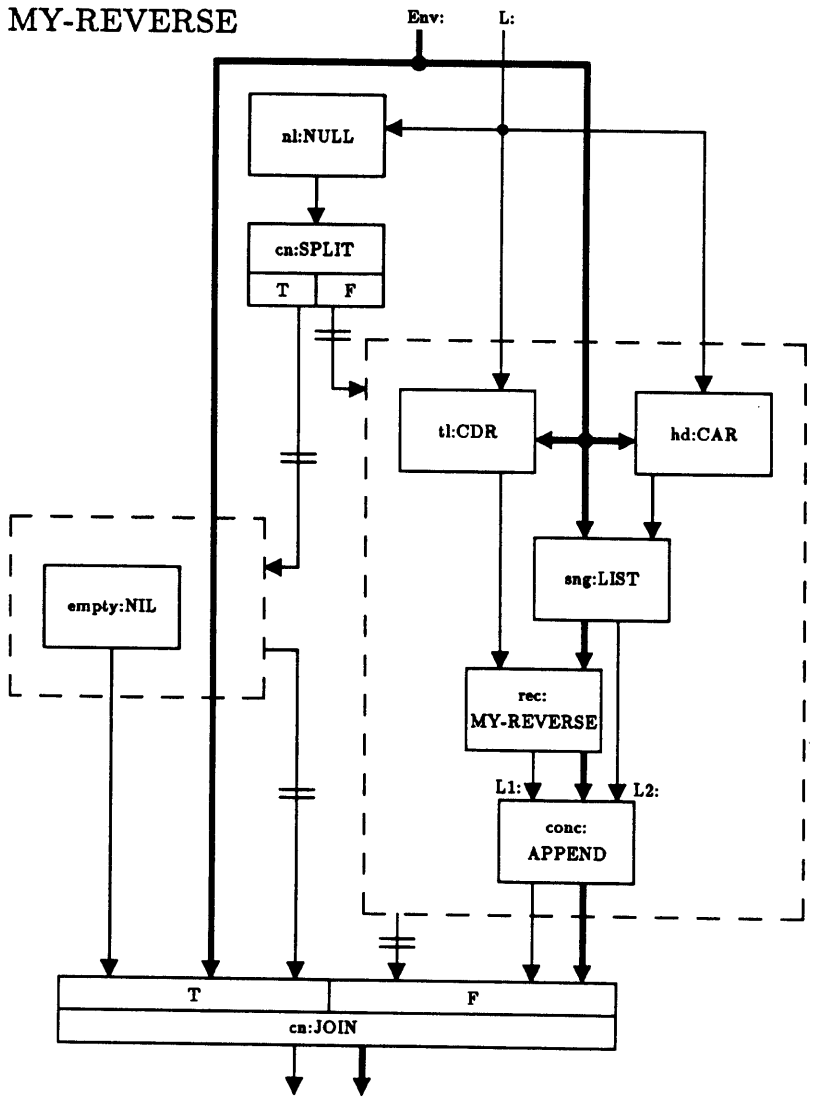


Figure 1.2: Structural representation of the recursive **MY-REVERSE** program. Thin arcs represent standard data flow; thick arcs represent stores; arcs with two short cross segments denote control flow, used here only to indicate conditional execution. This representation is similar to but somewhat different from the *Plan Calculus* as described in (Rich & Waters, 1990). A box is labelled with an instance identifier, a colon, and a type identifier.

representation corresponding to the MY-REVERSE program above, and Figure 1.3 shows those of APPEND and its subroutine NCONC. Note that side-effects are modelled as in standard denotational semantics using explicit stores. Note also that the representation does handle recursive programs.

- A set of *representative test inputs* for the program. The system incorporates an interpreter capable of executing programs on given inputs and recording trace data. It turns out that a single well-chosen test input is sufficiently representative of MY-REVERSE's behavior to allow all and only correct optimizations; one such test input is the cons-cell and store representation of the abstract list `(0 1 2 3 4 5)`.

- A set of *optimization invariants* (see Section 1.5.1 and Chapter 4) that together specify all properties of the program's input/output relation that must be preserved by the optimizer. If available, a ground-evaluable input/output specification would be the ideal optimization invariant(s). However, since a complete formal specification is often difficult to obtain, an optimization invariant may be *relative* in that it may express a relation between the optimized program's outputs and those of the *un*optimized program. The optimization invariants used for MY-REVERSE can be paraphrased in English as follows:

  - For a given input, the output of the optimized program must be the same as that of the original program *viewing both outputs as abstract lists;* that is, corresponding CARs must be identical, but the cells making up the list itself may be different.
  - The program may only modify newly allocated memory cells.

  Note that instead of using the well-known and easily formalized top-level specification of list reversal I use here a relative optimization invariant to demonstrate the technique. In this case, the two are logically equivalent, but in general a relative invariant will be more conservative (allow fewer optimizations), but easier to formalize.

- A *proof* (see Chapter 5) that the program's structure correctly implements its top-level specification based on domain axioms and program structural axioms. Since it is difficult to formalize and prove

program specifications, the system can accept incomplete design information in the form of proofs of *quasi-specifications*. The proof and specification inputs for **MY-REVERSE** were constructed automatically by a quasi-specification proof generator for side-effect-free list programs (see Section 1.5.1).

Intuitively, it is clear where all of the space inefficiency and much of the time inefficiency in **MY-REVERSE** comes from: every recursive invocation of **MY-REVERSE** makes a new copy of the reversed tail within **APPEND**. Of course, in general **APPEND** must copy its first input list to avoid destroying it. Used in **MY-REVERSE**, however, this copy operation is unnecessary, because the reversed tail is always made of "fresh" cons cells, and it is not used after the call to **APPEND**. Thus, we can greatly improve the performance of **MY-REVERSE** by eliminating the unnecessary copy operation. Note that this is exactly the type of thing that might be done by compilers incorporating "copy elimination." Section 1.4, however, shows that redistribution captures many other phenomena as well. Moreover, these manipulations apply equally to user-defined abstractions as well as language primitives.

The system discovers the two redistributions shown in Figure 1.4 that suffice to eliminate the unneeded copy operation. This modification has improved the space usage to exactly one new cons cell per cons cell in the input list. This is the best space usage possible in general for a nondestructive reverse.

The run-time is still quadratic, however. Almost every time **NCONC** calls **LAST**, it must iterate down the cells making up the list to find the last cons cell in the list. It turns out, however, that every time **NCONC** actually needs to know the last-cons of the list, **MY-REVERSE-1** has already computed it in the previous recursive invocation! The output of the **sng:LIST** box one level down in the recursion, after being appended to the reversed tail of the tail is now the last cons. Hence, if we could pass this pointer upward a level to the input of the **rpd:RPLACD** box, we could avoid the costly last-cons iteration. The system eventually discovers this optimization and implements it as in Figure 1.5. Note that this redistribution involves adding new auxiliary output values to the **MY-REVERSE** program.

In summary,

- **MY-REVERSE-2** is linear in both time and space, whereas the original implementation was quadratic in both. Linear is the best possible.

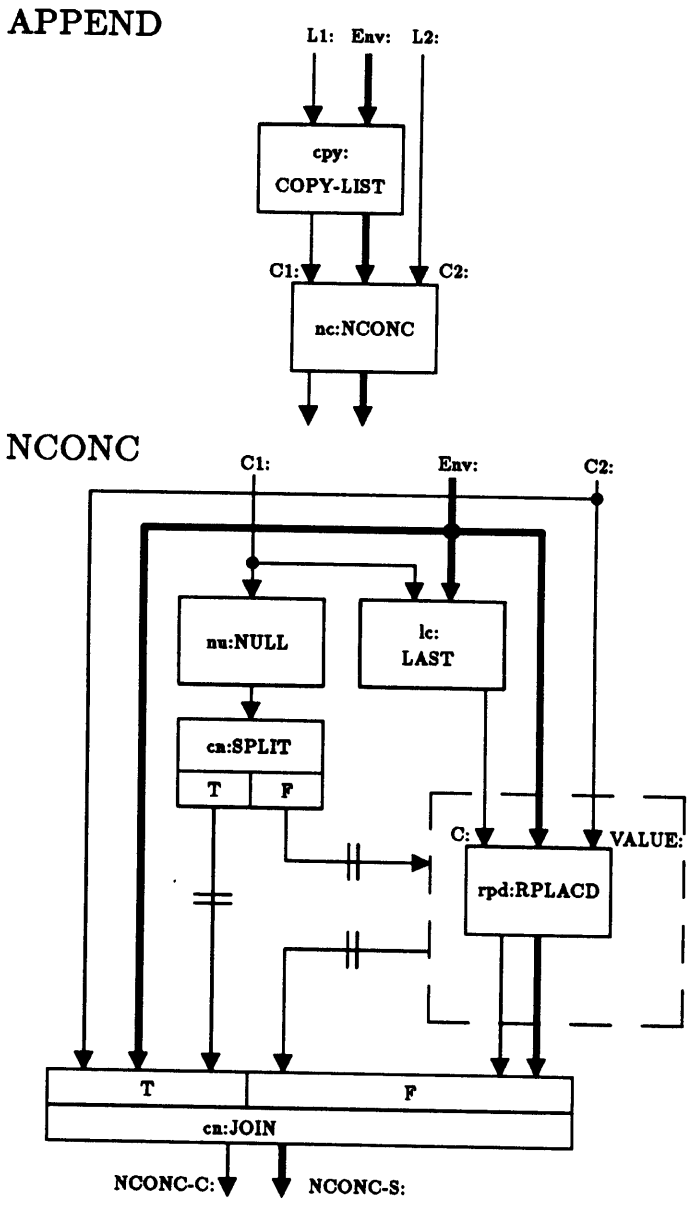


Figure 1.3: Structural representation of the APPEND and NCONC subroutines. These are the structural models used by the system, which do not correspond directly to the way these operations are typically implemented in Lisp. These are in "functionally exploded" form, as discussed in Section 12.4.

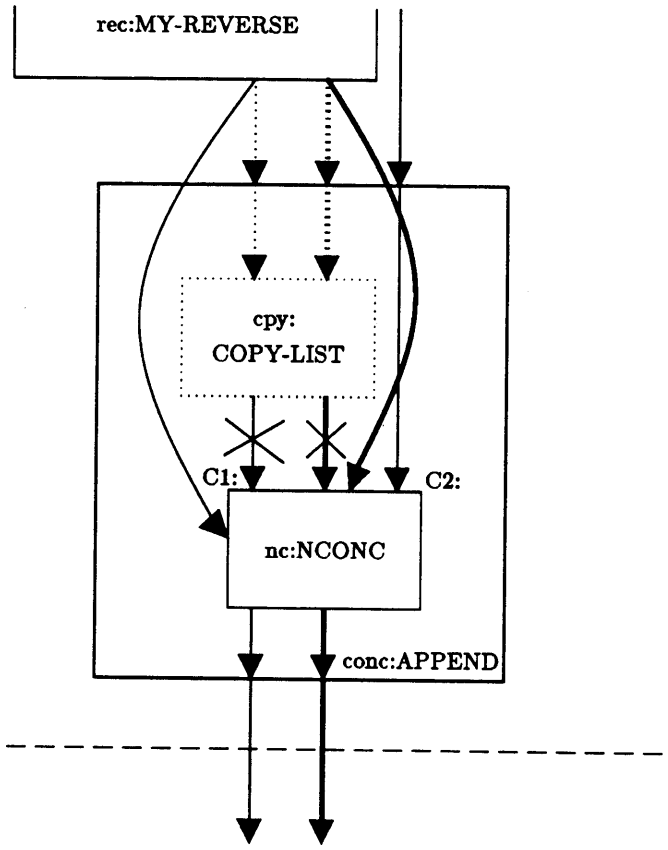


Figure 1.4: First optimization step for **MY-REVERSE**. This diagram shows a closeup of a portion of the **MY-REVERSE** program (see Figure 1.2), with the implementation of the **APPEND** box shown within its boundary. The copy operation has been eliminated by rerouting two flow arcs and eliminating dead code. The Xed-out arcs and the dotted boxes and arcs are unused structure eliminated in creating the optimized version.

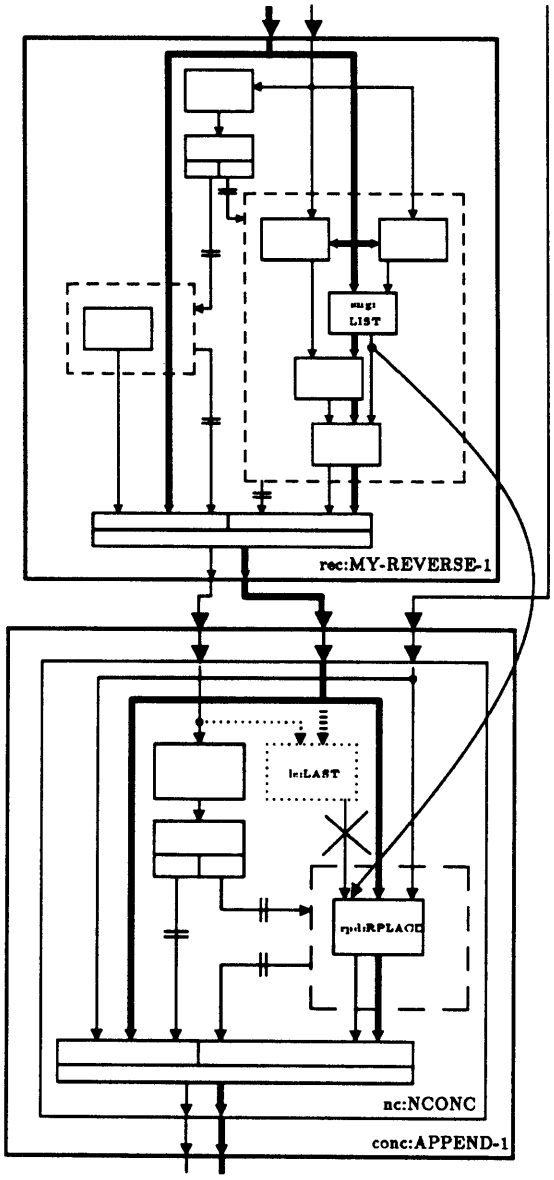


Figure 1.5: The second optimization step for MY-REVERSE. This diagram shows a closeup of a portion of MY-REVERSE-1 including the changes made by the second round of optimization. Implementations of subroutines are shown within solid box boundaries. The redistribution indicated by the curved arrow allows the elimination of the LAST box. It is implicitly applied at all levels of recursion.

- The optimizations introduced by the system consisted only of creating specialized versions of the original subroutines with some unnecessary calls eliminated. The system did not rely on a large library of highly specific program transformations.
- Even though the output of the optimized program is not identical at the cons-cell level to that of the original, the modified program still satisfies the given optimization invariants.
- The code is not very readable or clear, but since it will not be maintained (it can be regenerated automatically), it doesn't need to be. In practice, using an interactive approach to certifying optimizations may require that the system justify optimizations to the user, but in any case this won't require readable source code.

1.4 Optimization Phenomena Captured by Redistribution

A wide range of optimizations can be expressed in terms of redistributions. Recall that a redistribution requires finding a new source to connect to a given target, allowing one to eliminate the old source's computation if the old source is not used elsewhere. One large class of redistributions, called *identical-value redistributions*, are characterized by the new source's value being identical to the old source's value. This class generalizes the common notion of *common subexpression elimination*, wherein the old source and the new source must be computed by syntactically isomorphic subexpressions.

Often, however, a value *not* identical to that of the old source can be substituted for it for the purposes of the target. One large class of this type, *copy eliminations*, is based on eliminating unnecessary copy operations: the new source is simply whatever was to be copied by the copy box, and the old source is the output of the copy box. The eliminated box is the (unnecessary) copy operation.

Yet another class of optimizations, *generalized loop fusions*, includes examples of both identical-value and non-identical-value redistributions. By using a powerful representation for iteration due to Waters (Steele, 1990, Appendix A), loop overhead becomes an explicit structural element that can

be shared. Loop fusing redistributions eliminate these loop-representing elements. Note that a restricted version of this is well known in the traditional compiler world as *loop fusion.* The difference is that far more cases can be covered by redistribution applied to the more powerful representation than can be justified by the simple syntactic criteria used currently.

Finally, another class, *data invariant suspensions*, enables the removal of code whose sole purpose is to maintain unnecessary and costly data invariants. This class is *not* a generalization of any previously known compiler technique, simply because it is dependent on the optimizer having some awareness of the true specification of the program. Traditional optimizers can perform limited instances of this class *for particular data representations provided as primitives in the language.* Unlike my system, they cannot carry out invariant suspensions for user-defined data representations simply because they cannot exploit the necessary freedoms in the program's specification.

Given these observations, the single mechanism of redistribution conceptually unifies many optimization phenomena that are treated separately in the conventional compiler literature. Moreover, by virtue of using more input information, the approach can be applied at all levels of abstraction rather than only at the language level. Chapter 2 discusses these categories in detail and gives an example of each.

1.5 Research Overview

1.5.1 Key Ideas in the Implementation

This subsection summarizes the key ideas and tradeoffs made to obtain a practical system.

Given unlimited time and space resources, a (nonexistent) perfect theorem-prover for the domain, and a (nonexistent) practical theory of program efficiency, it would be easy to find the optimal set of redistributions to improve any program. One could separately test each subset of the quadratically many source/target pairs in the program and evaluate each one for correctness and degree of program improvement, picking the best one. Unfortunately, this approach is impossible. The correctness problem and the problem of evaluating program efficiency are uncomputable, and there are

exponentially many sets of source/target pairs to try.

Recall (Figure 1.1) that I have broken down the top-level problem of optimization into a candidate search phase that is to quickly produce a set of likely-safe redistribution conjectures (source/target pairs) that improves the program and a certification phase that checks the pairs for safety. The system I've implemented performs the candidate search phase; I assume an external agent performs certification. A key result of this research, however, is that it is possible to design the candidate search phase so that it discards candidates that, though they may be safe, would still require too much effort of the certifier.

I have broken down the candidate search problem into two subproblems:

- *Candidate generation:* which of the exponentially many sets of source/target pairs should we consider?
- *Candidate screening:* Given a source/target pair, how can we quickly evaluate its likelihood of correctness?

The two optimization algorithms, IEBR and IBR, use the same approach to candidate generation, based on box cost estimation. They differ in their methods of candidate screening and in the additional input information (beyond the program's structure) they require.

Key Idea 1: Additional Design Information

Requiring more information about the design is an extra burden on the user; thus, it represents a tradeoff of ease of use for increased performance. I will argue that the extra information is not too difficult to provide, particularly if the system is to be used in an integrated program design environment.

There are two principal ways that this research can potentially be applied: either as a stand-alone optimizer used directly by programmers, or as a subprocess of a larger automated design system, such as the Programmer's Apprentice (Rich & Waters, 1990) or KIDS (Smith, 1991). When used in the larger context, the extra design information will be produced as a natural part of the machine-mediated design process and hence represents little or no extra effort. In the stand-alone case, various techniques can be applied to help reduce the additional effort required of the programmer.

Optimization Invariants. Current optimizing compilers are fundamentally limited in power, because they do not have access to the extra information possessed by the human programmer. In the absence of some statement of optimization invariants, such as a top-level specification, the optimizer may only make program changes that can be proved to preserve correctness based on the specifications of the language primitives. For example, can the COPY-LIST in the following program be removed, i.e., replaced by IDENTITY?

```
(DEFUN F (X L)
  "Prepends X to L"
  (CONS X (COPY-LIST L)))
```

The answer depends on what specification the routine F is required to satisfy. The replacement is allowed unless the specification includes the requirement that the output cells be fresh. For example, the replacement would be disallowed if F were used in this program

```
(DEFUN DANGER (X L)
  "Appends X.L to L"
  (NCONC (F X L) L)),
```

but it would be allowed if F were used only here:

```
(DEFUN SAFE (X L)
  "Appends X.L to L"
  (APPEND (F X L) L))
```

A standard compiler is forced to assume the worst and not make the replacement; there is no way to provide it with a weaker specification than the most conservative. My system, however, accepts and exploits a statement of optimization invariants for each program. For F, if freshness is required, the programmer must give this explicitly as an optimization invariant, otherwise the system need not preserve it. Chapter 4 discusses optimization invariants further.

Feather & London (1982) have investigated the issue of exploiting specification freedoms in the context of automatically implementing high-level program specifications.

Explanations. The intuition behind the IEBR algorithm is that it is easier to find a substitute for a value and know that the substitute is adequate if you know the purpose(s) the value serves.

A key idea of this research is that *the purpose description above may be automatically derived from a* proof *of the top-level specification of the program.* Typically, a programmer will know why (he believes that) the program is correct. From this knowledge he can derive descriptions of the purposes of intermediate results and use such descriptions to justify optimizations. IEBR captures this notion using a new form of explanation-based generalization, called *parent-child clause unioning* (PCCU). See Chapter 9 for further discussion of this technique.

Key Idea 2: Using Incomplete Design Information

Both algorithms require extra information that can be difficult to supply. Fortunately, in each case optimizer performance can be traded off for ease of use.

Relative Optimization Invariants. To use IBR, the programmer must provide top-level optimization invariants (ideally, a complete formal specification) for the program. It is well known that complete specifications are difficult to formalize for complex programs. The system, however, can use *relative* invariants, because it has access to the unoptimized program which is assumed correct. For many programs, this makes the job of providing invariants trivial, as the effective specification can simply test for equality to the original outputs. In other cases, such as in `MY-REVERSE`, the original output values can be used to check some abstract properties of the new outputs, but some properties must still be expressed without reference to the original outputs. In still other cases, the original output values provide no help at all in checking the output values of the optimized program. In most instances, however, a relative specification is more conservative than a non-relative specification could be, but easier to formalize.

Proofs of Quasi-specifications To use IEBR, the programmer must provide a proof that the program's structure satisfies a given top-level specification. IEBR would likely be impractical if the user were forced to provide a complete proof of a complete specification for each program to be optimized.[2]

[2]One can, of course, simply choose to use IBR instead of IEBR; the former does not require a proof input.

Two observations address this difficulty.

First, much of the proving and specifying can take place when the library of reusable modules is built, rather than when the modules are used to develop a new program. Thus, the costs for library modules are amortized to effectively zero over all uses of the library.

Second, a tradeoff can be made: the user may sacrifice some optimization performance in return for usability. To support this tradeoff, I define a *quasi-specification*[3] to be any statement true of a program, allowing reference to internal elements (intermediate results) of the program's implementation. A specification must refer only to the inputs and outputs, allowing any implementation; a quasi-specification is not so restricted. Thus, quasi-specifications are really incompletely abstracted specifications, depending as they do on some details of the module's implementation. This allows one to *automatically* produce formal proofs that recapitulate the structure axioms of the program at varying degrees of abstraction. The key idea is that even though one has no real proof for the top-level program, one might still have better proofs for called subroutines (from the library, for example), thereby allowing some freedom to the optimizer. This idea can be extended to capture partial knowledge of the program, such as its side-effect-free nature.

I have implemented algorithms to support the fully automatic production of proofs of quasi-specifications for programs that operate on abstract lists and sets. The proofs and quasi-specifications accurately capture such properties as side-effect behavior and abstract list equality, but are overly restrictive regarding other behavior such as the actual abstract list functionality. Generally, automatic proof and quasi-specification production must be implemented differently for different domains of programs, since different domains have different invariants and properties that must be systematically captured.

In summary, the user need not do any difficult manual proofs, nor even define a complete formal specification to use IEBR; the system can do this automatically. Of course, the system will then miss some optimizations, because it cannot get as much information out of a quasi-specification proof as it could from a proof of a real specification.

[3] An earlier version of this introduction, published as (Hall, 1991), referred to "incomplete proofs" and "incomplete specifications." I apologize for this confusion; I believe the quasi-specification terminology is clearer and more descriptive.

Key Idea 3: Candidate Generation

The system's solution to the candidate generation problem represents a trade-off of search completeness for tractability. Both IBR and IEBR use the same heuristic search control strategy, based on a crude cost estimation technique that considers only those sets of pairs that would allow the system to immediately eliminate a box. Once a box is eliminated, the system then tries to eliminate more boxes, with the iteration terminating when all[4] boxes with cost estimate greater than a threshold (the *box cost threshold*) have been considered. More costly boxes are considered before less costly ones, and boxes within a box are considered after the box itself. The system can occasionally miss the best set of source/target pairs, either because the best set requires eliminating a low-cost box before a high-cost box, or because the best set of pairs cannot be partitioned so that each group is associated with eliminating a box. This is a version of the well-known "local maximum" problem that plagues all hill-climbing algorithms. The heuristic has performed well in practice, however.

Key Idea 4: Candidate Screening

The second aspect of the candidate search problem is candidate screening. A (human or machine) certification procedure for deciding whether a given redistribution maintains program correctness (is safe) is likely to be computationally costly. Thus, the search phase should discard as many faulty candidates as possible, and it should also try to pass only those safe candidates that are likely to be certifiable by a reasoner with limited resources. Of course, if the system is used *without* a compile-time certifier the issue of candidate screening is even more important.

Some candidates can be eliminated simply on "syntactic" grounds—either based on a static type clash or on a causality conflict, i.e., when the proposed source's value is partially determined by the target's value. Such simple tests are not enough, however. In even moderate-sized programs, hundreds or thousands of candidates may remain, of which only ones or tens are actually valid redistributions.

The key idea for solving this problem centers around the idea of using

[4]Recursion complicates the definition of "all boxes" in a program. See Chapter 7 for details.

concrete test inputs: if the program resulting from a source/target redistribution is incorrect on a given test, then the redistribution candidate can certainly be discarded. This represents a tradeoff: the negative aspect is that test inputs for complex programs can be difficult to compute and store. Positive aspects include: it provides excellent filtering; test inputs are usually available in the design environment; and, unlike automated theorem proving, it is easy to compute test outputs, given the program.

I have designed and experimented with two different approaches that exploit this idea. The approach taken in IBR is the simpler of the two: first, carry out the source/target redistribution structurally, and then re-evaluate each test, checking the correctness of the resulting program outputs using optimization invariants. If the results are not correct, the system retracts the structural change.

IEBR, on the other hand, avoids re-executing the tests for most pairs. More importantly, it also screens out many safe candidates that would be difficult to prove safe. It first derives an operational logical expression, called the *target condition* for each target considered. It does this using a new form of explanation-based generalization (DeJong & Mooney, 1986) called *parent-child clause unioning*, or PCCU. It then evaluates whether the source candidate satisfies the target condition in every concrete test. Once a pair passes the target condition test, the redistribution is carried out structurally and the tests are re-executed. Top-level optimization invariants are then checked (as in IBR) to make sure the redistribution hasn't rendered some prior pair invalid.

Variants of IEBR can be made to operate, at the cost of some optimization power, even in the absence of optimization invariants and even without *ever* re-executing a test. A justification of why IEBR restricts to candidates that are easier to prove safe must wait until Chapter 9.

Key Idea 5: Approaches to Correctness

Currently, programmers must solve two problems in optimizing their programs: first, they must find plausible and useful candidates. Then they must convince themselves that the candidates preserve correctness. I have automated the first problem, while cleanly partitioning the second problem.

The system produces a set of source/target pairs that, when used to optimize the program, preserve correctness on the given test inputs. This

does not, of course, guarantee that the program remains correct on every *possible* input.

It is obviously undesirable for an optimizer to introduce errors into the user's program; the user typically introduces enough by himself. Consequently, the system must be used with some external form of *certification* of redistributions. There are two places in the development process at which certification may be performed: compile-time (when the system generates the conjectures) or run-time (when the user runs the optimized program).

Compile-time certification. The idea here is for some external agent to check each candidate redistribution as the system conjectures it. This external agent may be either an automated reasoner, the human programmer, or some combination of the two. Various approaches suggest themselves, ranging from interactive near-term schemes to fully automated ones.

IEBR aids a compile-time certifier much more than does IBR: IEBR only conjectures changes to the program whose justification (if it exists) can be viewed as a perturbation of the original correctness proof. That is, all that needs to be proved is simultaneous satisfaction of the target conditions of the redistribution pairs, rather than a complete proof of the altered program. (See Chapter 9 for more discussion of this point.) Thus, IEBR tends to restrict attention to the more routine, easier to prove redistributions.

Run-time debugging. A modification of the notion of *efficient program checking* introduced by Blum and Kannan (1989) provides another approach to certification that does not rely on automated theorem proving at all. Intuitively, a program checker for a problem is a program (assumed correct) that can check the outputs of any program purported to solve the problem. My proposal is to use a program checker to check the optimized program's outputs every time it is run; if it is found to be incorrect, then signal the user and offer to re-optimize the program using the inputs for the faulty run as an additional test input. Of course, the re-optimization can ignore all source/target pairs that were shown incorrect in the previous run. This saves most of the time of re-optimization. The result of the re-optimization will be a program that is more often correct than the original optimized version. This approach is practical as long as the checking is efficient: the run-times of the optimized program and the checker together must be significantly less

than that of the original program alone.

1.5.2 Experiments

Both the IBR and IEBR algorithms have been run on 26 examples. Most of the examples are in the domain of a simple pointer-based representation of abstract lists, similar to but simpler than the list representation in Lisp. `MY-REVERSE` is one such program. Some examples are simple numerical programs and some operate on a list representation of sets, the set representation built as a next layer of abstraction on top of the abstract list representation. The most complex example, i.e., the one that takes the longest to optimize completely, is an implementation of the MERGE-SORT algorithm for sorting lists of numbers. The box-cost threshold was set as low as possible on all examples, in that the only boxes not considered for elimination were zero-cost boxes (constants).

The example programs were improved by large amounts, compared with the typical improvement obtained from a traditional optimizing compiler. Improvements included moving from exponential to linear, moving from quadratic to linear, and decreasing time and space usage by large constant factors.

To give an idea of the speed of the system itself, the complete search and optimization of `MY-REVERSE` took 158 seconds for IEBR and 346 seconds for IBR. For comparison, IEBR required 14980 seconds (4:09:40) to optimize the MERGE-SORT program, while IBR required 59081 seconds (16:24:41). The optimizations carried out included loop fusions and copy eliminations and were somewhat different between IEBR and IBR, though the two result programs were essentially equally efficient. The system is implemented in Common Lisp on a Symbolics 3670.

Chapter 11 and Appendix C discuss the examples in more detail and draw conclusions from them.

1.5.3 Evaluation

Redistribution of intermediate results captures a wide range of powerful optimizations. This is to be expected since it is a limited form of one of the most basic principles in all design optimization, function sharing. This research has investigated the automation of this idea, with the major technical results

being the derivation and use of target conditions to restrict to more routine optimization candidates, and the use of qualitative cost estimation and other heuristics to solve the (restricted) optimization search control problem.

Limitations of the approach exist at many levels of description. While some of these are fundamental, many appear to be interesting topics for future research. The concluding chapter discusses these in detail.

The success of the implemented prototype serves as an initial demonstration of the feasibility of the approach; however, much remains to be done to produce useful tools for programmers. Overall, I believe the techniques here hold significant promise for eventual application.

1.6 Reader's Guide

Chapter 2 indicates the wide range of phenomena captured by redistribution and also shows some examples which the system has optimized successfully.

Part One defines representations for and operations on program structure, program function, and the explanatory structure that connects structure to function. As in designing any representation to be used for human or computer reasoning, it will be important to keep in mind the aspects we wish explicit and those we wish to suppress. The definitions presented in Part One were designed with this in mind, together with a desire to keep things as simple as possible. Thus, certain phenomena may not be captured in full (or even at all); but it is hoped that the techniques will generalize to richer formalisms. Moreover, I do not claim that these representations are optimal for their respective purposes; experience with the implemented prototype has pointed out several improvements that can be made in a second round of implementation. By and large, however, they are good first tries.

Part Two describes the two optimization algorithms, IBR and IEBR. It first discusses the aspects common to both; subsequent chapters discuss the parts that differ. Finally, Chapter 10 describes the quasi-specification techniques used to cope with the difficulty of specifying and proving properties of programs. Due to the size and complexity of the implemented system, it was neither possible nor desirable to give precise descriptions of all algorithms used. However, I have given pseudo-code descriptions of the key procedures, leaving it up to the experience of the reader to fill in the lower level engineering details.

Part Three discusses the power, limitations, and implications of the algorithms. It compares the two algorithms to each other in order to highlight the important characteristics and tradeoffs underlying the general approach; it discusses the issues surrounding supplying the design information necessary to use the system; it discusses the issue of certification and several possible approaches to dealing with it; and it surveys relevant literature both to compare the system with other approaches and to place it in context of design systems in general. Finally, I draw conclusions and point out limitations, future research topics, and potential practical applications.

Part Four is a loosely structured set of appendices. The first three give the domain knowledge and a representative sample of the programs optimized by the system. A key chapter to be aware of throughout the thesis is Appendix B, which is an alphabetical glossary of function and program names with brief descriptions and cross references. Other appendices describe the proof restructuring algorithm; explore the theoretical relationship between computability, checkability, and relative checkability; and discuss some semantic subtleties relating to the use of the series representation.

Chapter 2

Optimization Phenomena

This chapter is intended to indicate the wide range of optimizations that can be expressed in terms of redistributions. It also illustrates the four categories of phenomena discussed briefly in Section 1.4.

Redistribution is particularly well-suited to improving programs constructed from general modules. That is, it is good at finding (and fixing) the types of inefficiencies that arise naturally from the use of general modules in specific contexts. On the other hand, one would *not* expect redistributions to significantly improve programs that are carefully hand-crafted. Each of the examples in this chapter arises naturally when programs are constructed quickly out of reusable (general) modules in a clear and readable fashion.

Details relating to the system's performance on each of these examples appear in Appendix C.

2.1 Identical-value Redistributions

The most obviously correct redistributions are those in which the new source's value is identical[1] to the old source's value. A particularly simple special case of this—where the old and new sources are computed by syntactically identical expressions—is known in the compiler literature as *common subexpression elimination* (Aho, Sethi, & Ullman, 1986).

[1] Unless otherwise noted, I will always mean "identical" and "equal" as synonyms denoting the mathematical sense of equality; this is not the same, for example, as the Lisp relation EQUAL.

There are many occasions when considerations of readability and clarity lead one to code multiple computations of the same value within a program. Consider the following self-evidently correct implementation of the polynomial evaluation formula

$$\text{poly}(C, x) = \sum_{i=0}^{|C|-1} C_i x^i,$$

where C is a finite list of coefficients (in increasing order of subscript) and x is a number:

```
(DEFUN POLY (C X)
  (LET ((SUM 0))
    (DOTIMES (I (LENGTH C))
      (SETQ SUM (+ SUM (* (NTH I C) (EXPT X I)))))
    SUM))
```

Two things cause this to use quadratic time:

- Iteration $i+1$ of the loop in `POLY` calculates x^{i+1} (in time proportional to $i+1$) by recursively calculating x^i and then multiplying by x. But iteration i of `POLY` just computed x^i, so this value could be shared correspondingly at each level of the recursion.

- Iteration $i+1$ of the loop in `POLY` traverses the list `C` from the beginning (within `NTH`, which calls `NTHCDR`) in order to find `(NTH` $(i+1)$ `C)`, taking time proportional to $i+1$. This computation finds `(NTHCDR` $(i+1)$ `C)` by recursively finding `(NTHCDR` i `C)` and then taking the `CDR`. `(NTHCDR` i `C)`, however, was computed in iteration i of `POLY`, so could be shared for each i.

The system discovers and carries out these two optimizations, resulting in a linear time program.

There are many other occasions where identical values may be shared in a program. For example, earlier redistributions can create opportunities for later identical-value redistributions. In `MY-REVERSE`, the first optimization, consisting of two non-identical-value redistributions, enabled the single identical-value redistribution of the second optimization.

2.2 Copy Elimination

Non-destructive operations on structured data types often take the conceptual form of copy-and-modify: the program makes structurally disjoint copies of (some of) the input objects and then performs a destructive operation on the copies to produce the result. The `APPEND` program used in `MY-REVERSE` is an example, as it copies its first argument and then destructively modifies part of it to attach the second input list. Frequent use of operations like `APPEND` often lead to programs that are inefficient in both time and space due to unnecessary copying. Note that this was the problem with the original `MY-REVERSE` that led to the first optimization. This inefficiency is inherent in the *specification* of `APPEND`, not in its implementation. Only by realizing that the specification of a given call to `APPEND` is overly general can the optimizer remove the extraneous copy.

Copy-and-modify operations need not be coded in a style as explicitly structured as the system's `APPEND` was; frequently the copying happens along with other operations, as the result list is built. For example, the standard recursive implementation of `APPEND` has this property. The system is capable of removing these unnecessary copies as well. Moreover, it can eliminate user-defined copy operations on user-defined datastructures at all levels of abstraction in exactly the same way as for copies of lists—by using the extra information in the input teleological structure.

2.3 Data Invariant Suspension

Another general class of redistributions centers around the idea of *data invariant suspension.* Concrete representations of data abstractions are often constructed by defining an abstraction function from a concrete type—a subset of a (possibly mutable, possibly recursive) record type—to the abstract type. The appropriate subset of the record type is defined by a set of constraints on the allowed values of the record fields. Each constraint is termed an *invariant*, because it is a property that must be maintained by any module that operates on the representation. Examples of invariants include

- A linked list used to represent a set contains no duplicate entries.
- A binary search tree representing a set is balanced.

- Each node in a pointer-based representation of a directed graph contains lists of both forward and backward neighbor pointers.

Figure A.1 in Appendix A shows the abstractions I used in experiements. The surrounding text discusses the invariants used to implement them.

Invariants must be maintained by modules that operate on the data structures if and only if their truth is required to prove the correctness of the specification of the program in which they are used. In a given context of use, the optimizer may *suspend* a data invariant (i.e. not maintain it locally) as long as the top-level program remains correct. The optimizer may then eliminate any subcomputations whose only purpose is to maintain the suspended invariant locally. Note that I allow specifications to constrain the overall efficiency of a program as well as its functionality; thus, invariants whose purpose is only to maintain efficiency of other operations, such as keeping a binary search tree balanced, may be suspended in certain contexts. An example of such a context is one in which the efficiency to be gained is only in modules not subsequently used by the program.

As an example of invariant suspension, consider the linked-list representation of a set. The operation of adding a new member to a set (set-add) must check to see if the added element is already an element of the list to avoid duplicating an entry. The only purpose of such a check is to maintain the no-duplicates invariant. The no-duplicates invariant is then assumed true in implementing operations such as set cardinality. However, the usual implementation of the set membership operation will be correct even if the list contains duplicates. Thus, if a program were to do several set-add's followed only by some membership tests and not return any of the set objects as return values, then the no-duplicates invariant could be suspended over that portion of the program. The set-adds would not need to do the extra checking; they could simply push the new element onto the front of the list. Each set-add would cost only a constant amount of time instead of time proportional to the length of the list. Section C.3.1 shows an example where the system performs this optimization.

Note that I am not claiming that *all* invariants are associated with code that is for one purpose only. It is frequently the case that subroutines perform more than one function in a program. Thus, suspending just one of the invariants maintained by such a module would not allow its elimination. The system is most effective at introducing function sharing into designs where

the individual boxes have single (or few) purposes. My observation is that it is usually possible to find such structural models of programs, though these do not always correspond with the most natural implementations. Often it turns out to be better to let the optimizer introduce function sharing in a context-sensitive way, rather than just using a partially shared implementation everywhere.

2.4 Generalized Loop Fusion

In the conventional compiler literature a *loop fusion* (or jamming) is the action of merging two iterations over the same range when there can be proven to be no dataflow conflicts between the two loops. For example, a loop that sums the odd numbers in an array may be fused with one that sums the even numbers, saving one round of incrementation and bound checking. Typically such iterations must be explicitly bounded and the ranges must be identical. Redistribution optimizations can capture more general loop fusions by a judicious choice of program representation.

In this research, loop fusions are distinguished by the fact that the shared "value" is a temporal sequence of values generated by iteration. These general loop fusions do not necessarily involve same-length sequences of identical values, because loops with different termination criteria may have their overlapping ranges fused and corresponding elements may differ.

The system uses a representation of loops based on Waters's Series Expressions (Steele, 1990) which requires significant back-end compiler support (in the form of a macro package). Note that even though the macro package does non-trivial post-processing, the adaptation problem is still solved by my system using only (series) dataflow rerouting. This is an advantage of using a powerful representation.

A *series* data object is a mathematical sequence of Lisp data values, i.e., a function from $\{0 \ldots k\}$, for some $k \leq \infty$, to standard Lisp data values. At the implementation level, each series object is used to represent the sequence of values taken on by some variable on successive iterations of the compiled loop. See the appendix of (Steele, 1990) for explanations of the series functions used here.

As an example of generalized loop fusion, consider the following Common Lisp program, `EQUAL-AFTER-SUBSTITUTION?`, which in this original form has

two loops. (This example appears in Section C.2.6 in a somewhat different nomenclature.) The first consumes linear space in constructing an intermediate list representing the substituted list; this is then iterated over by the EQUAL program. Subsequently, the intermediate list is discarded. Note that both loops are implicit in the series representation.

```
(DEFUN EQUAL-AFTER-SUBSTITUTION? (NEW OLD L1 L2)
  "True iff L2 is EQUAL to the result of substituting
   NEW for OLD in L1"
  (EQUAL (SUBSTITUTE NEW OLD L1) L2))
```

The Lisp primitives are modeled structurally as follows:

```
(DEFUN EQUAL (L1 L2)
  "Lisp EQUAL"
  (LET ((MARKER (MAKE-SERIES (GENSYM))))
    (COLLECT-AND
      (#MEQ (CATENATE (SCAN L1) MARKER)
            (CATENATE (SCAN L2) MARKER)))))

(DEFUN SUBSTITUTE (NEW OLD LIST)
  "Simplified Lisp SUBSTITUTE"
  (COLLECT
    (#MSUBIFEQ (SERIES NEW) (SERIES OLD) (SCAN LIST))))

;;; Helper for SUBSTITUTE
(DEFUN SUBIFEQ (NEW OLD X)
  (IF (EQ OLD X) NEW X))
```

This Lisp code corresponds to the data flow representation used. The models are presented in Lisp for expository purposes; the system accepts only the data flow representation.

Series expressions have the advantage of making iterations explicit structural elements of programs; thus, they become things that can be shared among different users. The iteration implicit in SUBSTITUTE is over the elements of the first list argument, L1, to EQUAL-AFTER-SUBSTITUTION?. The iteration in EQUAL is over both the result list of SUBSTITUTE and the second input to EQUAL-AFTER-SUBSTITUTION?. The system introduces function sharing here by replacing the (SCAN L1) in EQUAL with a series dataflow from the output of (#MSUBIFEQ ...) from SUBSTITUTE. This obviates the COLLECT in SUBSTITUTE, so the intermediate list output of SUBSTITUTE is not created.

Furthermore, the iteration is now controlled by the `COLLECT-AND` which can terminate before the end of the input list is reached. The resulting program has a single loop over the input list, `L1`. The improved version saves substantial time and space.

Part I

Representations and Operations

Chapter 3

Program Structure

The purpose of this chapter is to define the system's representation of program structure. From now on, in speaking of a computational entity, I will use the following uniform terminology: "code" or "source code" always denotes its representation in a certain standard programming language, like Lisp. "Program" is reserved for its representation as a functional dataflow program (as described below). This choice reflects the ontological stance that the latter representation captures something more fundamental, abstracting from unimportant details introduced when rendering a program in a source language.

Programming languages are designed with many considerations in mind, such as efficient execution by machine, code readability, and many others. These considerations directly influence the features available. The optimization task, on the other hand, requires different features; therefore, most programming languages are not appropriate structural representations for this task. For example, it is crucial that dataflow be explicit in a representation for optimization. In most programming languages, local variables are used as notation for communicating intermediate data values among subroutines; variables are familiar from algebra and logic, and such familiarity often leads to better readability. Variables are not explicit enough for optimization, however. Most optimizing compilers compute and operate on some type of dataflow representation of the input programs (Aho, Sethi, & Ullman, 1986).

I chose the program representation based on the following desiderata:

- A key distinction in optimizing by redistributing intermediate results is between *sources*, value-producing entities within a program such as

function call outputs, and *targets*, value-using entities such as function call input arguments. Information can be thought of as "flowing" from sources to targets via explicit *flow arcs*, and redistributions are implemented by introducing new such arcs and deleting old ones. Thus, one goal of the representation is to make sources, targets, and flow arcs within a program explicit. In standard programming languages, a flow arc is implemented implicitly by setting and reading a variable or by an expression appearing in an argument position of a function call.

- The chief way redistributions can improve the speed of a program is by allowing the elimination of subroutine calls whose return values (and side effects) are not used. Thus, every subroutine call must be represented explicitly and independently of every other to maximize the potential for optimization. A subroutine call within a program will be termed a *box*. Boxes have *ports* corresponding to the input arguments and output values of the program to which the box represents a call. In my formalism, free variables must be modeled as explicit input and output ports. Boxes with side-effects must have explicit store input and output ports (see below). A box may be of the same type as the calling program; such a box will be called a "recursion box." For this research, I allow only explicit, self-recursion: programs may not call other programs which in turn call the first program—they may only call themselves directly.[1]

- Types in programming languages provide many advantages. For our purposes, the key advantage will be that we can quickly rule out many potential optimizations on the grounds of conflicting type. Thus, every program's input and output ports are endowed with a type from a user-supplied type hierarchy. The system places only two requirements on this type system:

  - It must contain the type `BOOL`, representing the standard boolean values; and

[1]This restriction is made to simplify the implementation of several algorithms, though I do not believe it is fundamental. In particular, programs with complex recursive structures may always be mechanically converted to ones using explicit self-recursion only, though I do not propose that as the right way to deal with it.

- It must come with a fast implementation of a type-intersection predicate, `STATIC-TYPE-CHECK?`, that returns true if and only if its two type-identifier arguments denote non-disjoint types. The requirement of a fast implementation rules out complex type systems. In particular, the type systems of some programming languages may be too complex for our purposes; this is because while the number of value uses to be checked by a standard compiler grows linearly with program size, my system considers *quadratically* many source/target pairs in the worst case.

- "Stores," similar to those used in standard denotational semantics (Stoy, 1977), are conceptual objects that record mappings from (non-stack) memory cells to values. Side effects must be modeled explicitly using "store flow" arcs so that the system can redistribute store arcs in the same way as other types of arcs. This allows such optimizations as copy elimination and removal of unnecessary datastructure maintenance operations. Note that the explicit flow arc representation allows us to avoid explicitly modeling mutable stack memory locations; thus, stores need only apply to non-stack cells.

- A means must be provided to support conditional execution of boxes within a program.

These are the key features we need to talk about redistributions within programs; thus, almost everything else is left open and must be defined by the programmer through the type system, computational primitives, and other forms of system domain knowledge. I will discuss this knowledge later.

3.1 Dataflow Programs

Directed, acyclic dataflow programs satisfy the requirements listed above. I will adopt a program representation similar in spirit, but not in detail, to *plans* in the *Plan Calculus* (Rich & Waters, 1990). Section 3.5 discusses the differences in detail.

Figure 3.1 depicts the **APPEND** program. It is copied from Figure 1.3. Recall that it is an abstraction of the following Common Lisp code:

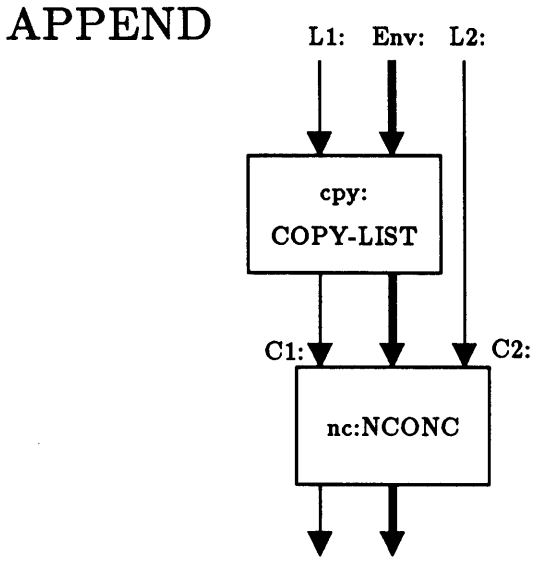


Figure 3.1: The APPEND program. The text explains the graphical notations used.

```
(DEFUN APPEND (L1 L2)
  "APPEND L1 and L2 non-destructively"
  (NCONC (COPY-LIST L1) L2))
```

Each box is labelled with its instance identifier ("role name" in Plan Calculus terminology) in lower case, followed by a colon, followed by its type name[2], designating to which program this box is a call, in upper case. Left-to-right order is never significant in the diagrams. If necessary, explicit port identifiers disambiguate which arcs are bound to which ports. Port identifiers are shown (as necessary) near the port they label, including overall program input and output ports.

Arrows indicate flow of data objects. Thin arcs represent standard data flow. Thick arcs distinguish store flow, communication of store objects. Later, it will be useful to introduce wide, outline arcs ($\Rightarrow$) to distinguish series flow arcs (Section 3.2.2).

Lisp, like most programming languages, treats the store (for mutable objects) as an implicit argument and return value. Thus, a more exact

[2]This use of the word "type" is meant merely in the generic sense of type versus token, not in the more technical sense of Scott domains.

functional rendering of the diagram would look like

```
(DEFUN APPEND (L1 L2 ENV)
  "APPEND L1 and L2 non-destructively"
  (MULTIPLE-VALUE-BIND (COPY CPY-ENV)
      (COPY-LIST L1 ENV)
    (NCONC COPY L2 CPY-ENV)))
```

Note that programs are "functional" in precisely the same sense as programs rendered in FP, pure Lisp, or pure Prolog source code; they are being used here to model "nonfunctional" Lisp programs by promoting the stores to first class status. Stores (see Section 3.2.1) are not first-class objects in Lisp, however; I will therefore always suppress them when showing Lisp code.

3.1.1 Program Elements

A *program* consists of

- a (globally) unique program *type* identifier (i.e., a program name);
- a finite set of statically-typed (as described above) *input ports*, each of which is named with an identifier unique among all identifiers in the program;
- a finite set of statically-typed (as described above) *output ports*, each of which is named with an identifier unique among all identifiers in the program;
- a finite set of *conditionals*, each of which is uniquely identified. Each conditional must have a unique boolean-typed test port, a finite set of output ports and two sets of input ports, each of which is in one-one correspondence with the output port set. One such set, termed the conditional's *true-output port* set, designates the values to return when the test port's value is true and the other, termed the conditional's *false-output port* set, designates the values to return when it is false. Each conditional has a *condition* (possibly empty) that describes which (other) conditional test ports must have which values in order for the conditional to be executed;

- a finite set of *boxes*, each of which is uniquely identified. Each box must have a *box-type* which names a program, P. Further, each box has input ports in one-one correspondence with the input ports of P. Each must also have output ports in one-one correspondence with the output ports of P. Each box has a *condition* (possibly empty) that describes which conditional test ports among those in the program must have which values in order for the box to be executed;

- a finite set of *flow arcs*,[3] each of which connects some source in the program to some target within the program, where

  - a *source* within a program is either a program input port, a box output port, or a conditional output port; and
  - a *target* within a program is either a program output port, a box input port, a conditional test port, a conditional true-output port, or a conditional false-output port.

  Flow arcs are allowed to "fan out;" that is, there may be more than one flow arc with a given source. There may also be sources not connected to any flow arc. However, flow arcs may *not* "fan in;" that is, there must be no more than one flow arc with a given target.

A source's condition will be either (1) empty, if it is a program input, or (2) the same as that of the box or conditional of which it is a port, otherwise. A target's condition is defined similarly, except that conditional true(false)-output ports have the parent conditional's condition conjoined with the truth(falsity) of the parent conditional's test port. If a condition contains a reference to the test-port of some conditional, then the condition of that conditional must be a proper subset of the condition. In other words, conditions (and hence conditionals) must have a last-in-first-out nesting structure.

Programs, viewed as directed graphs with boxes as nodes and flow arcs as edges, must be acyclic. This restriction keeps the operational semantics of programs particularly simple.

[3]By "flow" I will always mean to include "store flow" and "series flow" and any other particular type of arc.

3.1.2 Programs have eager dataflow semantics

Now that I have defined the structure of legal programs, I must also define their meanings (operational semantics). Intuitively, program execution can be thought of as eager dataflow execution. Section 3.4 explains how the system actually executes programs on given data, but that is merely an efficient implementation of the intuitive dataflow operational semantics. I chose eager semantics for simplicity, "intuitiveness," and ease of constructing an efficient interpreter.

Figure 3.2 illustrates the eager dataflow execution of `APPEND` on some inputs. First, associate the input values with the corresponding program input ports (a). Whenever an arc has a value associated with its source end (tail), the value "flows" down the arc and appears at the target (head) end (b, d, f). An arc with more than one target end is an abbreviation for several arcs, each with one target end. Whenever a box has values at all of its input ports, its corresponding type program is recursively executed using those input values to determine the box's output values (c, e). This process continues in parallel throughout the program until the output ports of the program get values (f).

A call to a box may fail to return because of a nonterminating recursion, in which case all of the program's outputs are undefined. If all boxes terminate, however, all program outputs must appear.

3.1.3 Conditional execution

The description above is silent on conditional execution of boxes, so to complete my description of program semantics, I must explain how conditionals are executed.

Conditionals can be viewed as composite objects constructed out of the more traditional dataflow elements *splits*, *joins*, and *control flow*. Even though the system represents conditionals internally as single objects and treats them specially, they behave precisely as if they were constructed out of the skeleton shown in Figure 3.3 (a). Figure 3.3 (b) shows this used in the `NCONC` program. I present this view of them merely as an aid to understanding how conditionals operate in programs.

A split takes in a single boolean value and puts out a single "control token" either to its T output or its F output, depending on whether the input

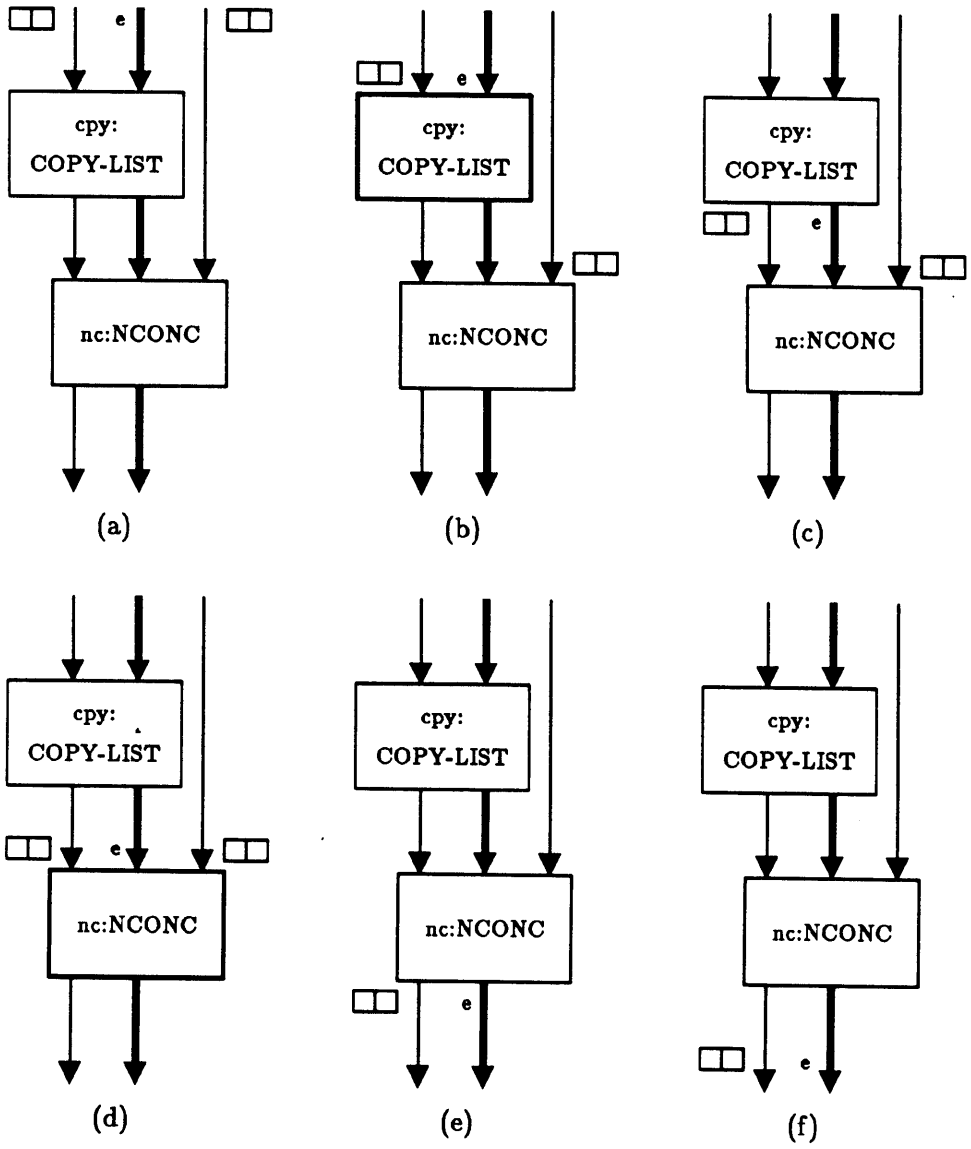


Figure 3.2: An illustration of the execution of APPEND on some inputs. Values "flow" and boxes "fire" as in standard dataflow. Highlighted boxes are those ready to fire by virtue of having all inputs present.

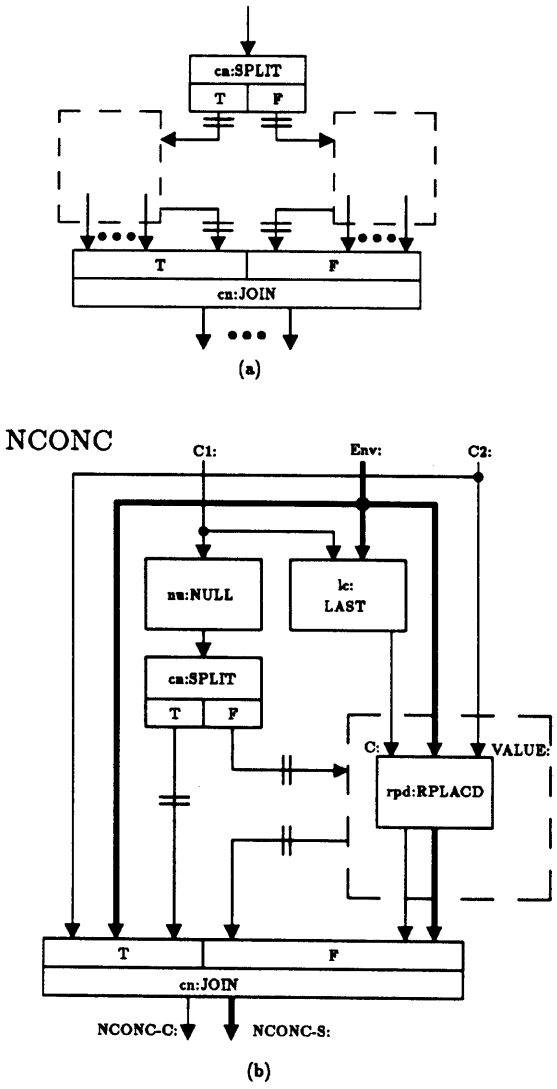


Figure 3.3: (a) The traditional dataflow components representing a conditional. (b) The `NCONC` program. This illustrates the use of a conditional to model conditional execution. The `rpd:RPLACD` box is only executed if the control token arrives from the F port of the split; i.e., if the output of the `nu:NULL` box is false.

is true or false respectively. This is different from regular boxes, because not all outputs appear for given inputs. A dashed rectangle surrounding some boxes is a structural grouping used to indicate that those boxes are called only if the control token appears. This is syntactic sugar for a control flow arc from the split to each of the boxes within the dashed outline. A join takes in two corresponding groups of inputs, each group including a control flow arc, and puts out the non-control values of the group whose control token is present. Outputs are undefined if both control tokens are ever present. A join is not a box, either, since it may fire before all of its inputs appear. Control flow arcs are not treated the same as other types of flow arcs; in particular, they will not take part in redistributions. Ports of type "control" are neither sources nor targets. Note also that the split and join of a given conditional have the same instance identifier; this is to indicate they are conceptually treated as part of the same object.

Note that the nesting of conditions discussed earlier forces conditionals to be nested in last-in-first-out fashion—arbitrary jumps of control flow (eg, `GOTO`s) are not allowed.

3.1.4 Showing box implementations

This subsection and the remainder of Section 3.1 are only loosely connected; each discusses a single issue relating either to the implementation or to the exposition of the material.

Redistribution optimizations can be thought of as introducing new communication paths from internals of one subroutine to internals of another. Thus, to show redistributions, it will often be useful to show the implementation of a program P within a diagram of a parent program Q that has a box of type P. This will be done by nesting the implementation within a rectangle. This may be done to any level of nesting desired. Figure 3.4 demonstrates this for **APPEND**. Note that this does not imply the routines have been coded in-line; this is merely an augmented view of the same structure. The system has no conceptual object corresponding to an "expanded box." Note also that this is not the same as a dashed box surrounding a group of program elements—dashed boxes indicate only structural groupings with no input/output boundary.

Viewing boxes' implementations from the perspective of the calling program is both an expository tool and an important technical idea used by

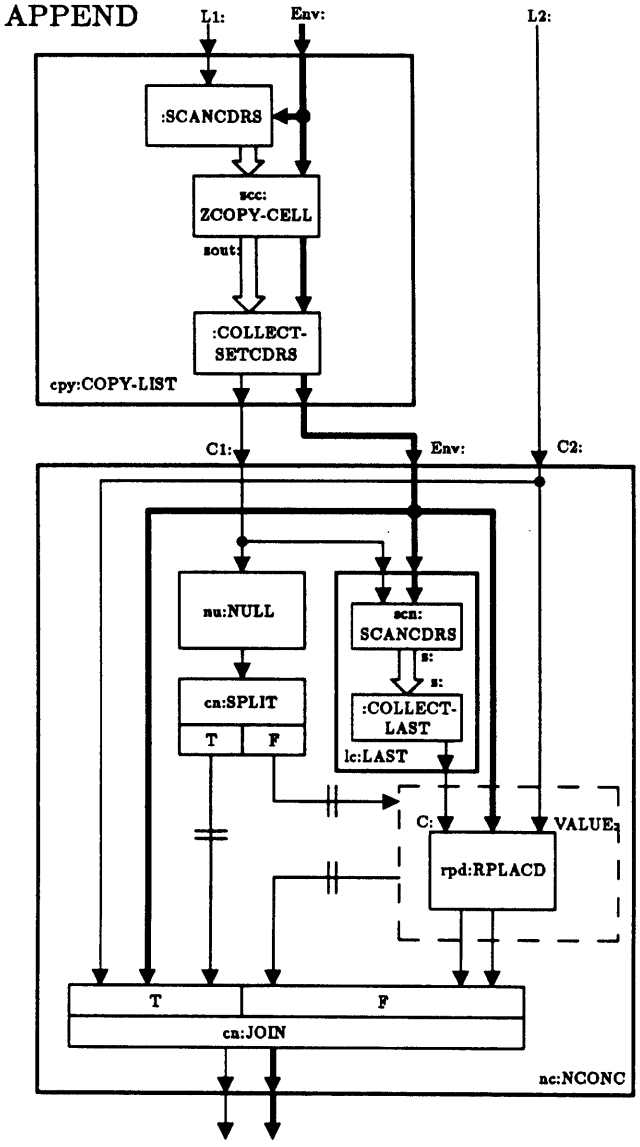


Figure 3.4: Showing box implementations within the box boundary. Here, the `cpy:COPY-LIST`, `nc:NCONC`, and `lc:LAST` boxes have been "opened." (Compare to Figure 3.1.) The instance names of some boxes have been suppressed. For example, the box labelled `:COLLECT-LAST` has instance name `coll`. Outline arrows ($\Rightarrow$) indicate store flow.

the system. To introduce new communication paths into and out of box internals, one must consider the internals of the boxes; one can do this either by having a view mechanism, or else "flattening" the input program, creating a new program with all subroutines in-line. The latter approach is relatively costly, since a flattened program is typically much larger to create and manipulate than the sum of the program definitions of all the boxes.

The system's approach to search control determines when and how deeply to look within boxes.

3.1.5 Virtual structure

Figure 1.4 (Section 1.3) showed a program fragment having a (dotted) box in it whose outputs were no longer connected to anything, due to a redistribution. A back-end processor that actually carries out the optimizations recommended by the system would remove such boxes from the design (dead-code elimination). However, it will be useful to be able to refer to such structural elements even though they do not have a causal effect on the outputs of the program. The reason for this minor paradox is that IEBR develops and uses descriptions of the purposes of the internal targets within a program and these descriptions can refer to a piece of structure even after a redistribution disconnects it. See Chapter 9.

I therefore define a box or conditional to be *physical* if there is a directed flow path from it to a program output. An arc is physical if it lies on such a path. A program input port is physical if it is connected to a physical arc; a box input port is physical if the corresponding program input port of the box's program-type is physical. A program output port is always physical. A box output port is physical if the box is. All other boxes, conditionals, ports, and arcs are *virtual*. Thus, the `cpy:COPY-LIST` box in Figure 1.4 is virtual, as are its input arcs and the `MY-REVERSE` arcs connected to them. The arcs with Xs through them are *not* virtual; they are removed completely by the system. Though the system may remove some arcs completely, it will never remove any boxes—boxes may only change from physical to virtual.

3.1.6 Pathnames

It is useful to have a systematic naming convention, similar to a "coordinate system," with which to talk about sources, targets, and other program

structural elements. Each object in a program corresponds uniquely with a sequence of identifiers, called its *pathname*, as follows. Each pathname begins with a $—a syntactic cue for the reader—and the top-level program name, followed by a sequence of identifiers separated by periods.

- A program of type P has pathname $P;
- Any input or output of the program $P whose identifier is A has pathname $P.A;
- Any box or conditional within P whose identifier is I is denoted $P.I;
- Any input or output port of the box or conditional $P.I whose identifier is A has pathname $P.I.A;
- By convention, the identifier of the test port of every conditional is TEST. Thus, the test port of a conditional $P.CN has pathname $P.CN.TEST;
- By convention, each conditional true-output port is named consistently with its corresponding conditional output port as follows. A conditional $P.CN with output port named $P.CN.RESULT-X has corresponding true-output port named $P.CN.RESULT-X.TRUE;
- Similarly, a conditional $P.CN with output port named $P.CN.RESULT-X has corresponding false-output port named $P.CN.RESULT-X.FALSE;

For example, Figure 3.4 shows a flow arc connecting APPEND's input port $APPEND.L2 to $APPEND.NC.C2, the second cell input port to the box whose pathname is $APPEND.NC. Note also that even though there are two things labelled "Env" in the diagram, the upper one refers to $APPEND.ENV and the lower one refers to $APPEND.NC.ENV.

A useful feature of this pathname convention is that we may talk systematically about structural elements at any level of the structural hierarchy relative to a given program in a natural way. This is done by replacing the program identifier in a pathname with the instance pathname of a box of that type. Thus, for example, to refer to the port $P.I.A within the box $Q.B (assumed of type P), we simply use the pathname $Q.B.I.A.

In Figure 3.1, we can refer to the port labelled VALUE by the pathname $APPEND.NC.RPD.VALUE; the conditional test port has pathname

$APPEND.NC.CN.TEST; and we can refer to the Z output of the scn:SCANCDRS box within lc:LAST within nc:NCONC within APPEND by the pathname $APPEND.NC.LC.SCN.Z.

Source/target ambiguity. Note that every port, except top-level input and output ports, actually names both a source and a target, depending on whether it is viewed as a box port within a parent program or as a program port within a box implementation. For example, in Figure 3.4, $APPEND.NC.C2 can be viewed as a target in that it is a box input port of $APPEND.NC; on the other hand it is also a source when viewed as program input port to (an instance of) NCONC. Since a flow arc emanates from there to $APPEND.NC.RPD.VALUE, it must also be a source. A similar confusion is possible with box/program output ports. This confusion is significant, because the system can consider redistributions, say from $APPEND.NC.C2 to $APPEND.NC.RPD.C.

Unless explicitly stated otherwise, I will use the "outer" interpretation by default; that is, pathnames referring to box input ports will be interpreted as targets and pathnames referring to box output ports will be interpreted as sources. Thus, $APPEND.NC.C2 should be interpreted as a target unless otherwise stated. The system investigates both interpretations when searching for redistributions.

3.1.7 The causality relation on program elements

Since programs may not contain directed flow cycles (by definition; see Section 3.1.1), the elements of a program are partially ordered by the transitive closure of the flow relation. This easily-computable relation, called the *causality relation*, is useful for quickly ruling out the candidate optimizations that would introduce flow arc cycles.

The relation applies to boxes and ports (including conditional ports), but not conditional objects as a whole—different pieces of a conditional are causally separate. I will say that a program element, p_1, is a *causal predecessor* of another program element, p_2, precisely when there is a directed flow path, possibly involving control flow as well, from p_1 to p_2. In this case I will also say that p_2 is a *causal successor* of p_1. Note that since we include control flow in the definition, a box whose condition includes a given conditional test port is a causal successor of the conditional test port.

We may extend the causal relation to include elements within box implementations by adding relationships between elements within a box to elements outside of the box. This is done by making every element outside the box that is a predecessor of the box be a predecessor of every element within the box, and similarly every element outside the box that is a successor of the box be a successor of every element within the box. This procedure is applied at all levels.

Thus, in Figure 3.4 `$APPEND.CPY.ZCC.ZOUT` is a causal predecessor of `$APPEND.NC.LC.COLL.Z`, because the former appears in `$APPEND.CPY`, the latter appears in `$APPEND.NC`, and there is a flow path from`$APPEND.CPY` to `$APPEND.NC`. On the other hand, `$APPEND.NC.LC.SCN.Z` is neither a causal predecessor nor successor of `$APPEND.NC.CN.TEST`.

3.2 Modeling Issues

Having concluded a discussion of the basic structural representation of programs, I must now discuss how to model desired capabilities (like side-effects) within the formalism. While the system is neutral with respect to modeling choices, they still profoundly influence the results achievable. This section discusses some of the choices I've made for experimentation purposes.

3.2.1 Stores model side effects

Lisp, like most programming languages, treats the store (mapping of memory cells to their contents) implicitly. The system, however, must treat the store explicitly in order to be able to reroute "store flow." The ability to redistribute intermediate stores will allow the elimination of store-changing operations such as extra copying and unnecessary data field maintenance. Thick arcs denote store flow in the diagrams. Formally, store ports have a type which is disjoint from all other types. In my system, this is enforced by the `STATIC-TYPE-CHECK?` predicate.

Thus, for example, to obtain the contents of a field of a memory cell, one must call the `SEL` function, giving it the cell, a store, and a field number. To alter the store, a program must take in the current store and put out a new store. Thus, boxes changing the store must be totally ordered by store flow. See Appendix A for further details of modeling side effects using stores.

3.2.2 The series data type models iteration

Wide, outline arrows in diagrams, such as that between `$APPEND.CPY.ZCC.ZOUT` and `$APPEND.CPY.COLL.Z` in Figure 3.4, denote *series* flow arcs. The type identifier for series is `Z` for compatibility Figure 3.4 has examples of this type of arc. The reader is referred to Appendix A of (Steele, 1990) for a full description of the series type. I will summarize here briefly some aspects of the theory.

Formulating in terms of series expressions makes iteration components explicit structural elements of programs (sources), allowing the system to introduce function sharing between distinct loops. The only other way of expressing iteration—through tail recursion—has the effect of "spreading out" the structure of the iteration over the program, with several distinct elements contributing partly to the iteration and partly to other aspects of the program. This makes it difficult to share iterations among several clients.

A "series" data object is a mathematical sequence of data values (i.e., a function from $\{0 \ldots k\}$, for some $k \leq \infty$, to data values). When a series expression is finally rendered into Lisp code and compiled by the Series Macro Package, it becomes a loop. Each series object names the temporal sequence of values taken on by some variable on successive iterations of this loop.[4]

For our purposes, the key advantage of using the series formalism is that Waters's macro package can compile "optimizable" series expressions into efficient loops that do not create intermediate data objects to hold the series. Typically, no series object is ever explicitly created; they are all coded using loops and iteration variables. This yields efficient implementations compared to the analogous expressions using Lisp sequences.

As an example of the use of series functions, the `COPY-LIST` program (see Figure 3.4, within the `cpy:COPY-LIST` box) operates by first calling `SCANCDRS` to produce a series of the conses in the input list; next `ZCOPY-CELL` makes a series of copies of each cell; and finally `COLLECT-SETCDRS` assembles the copies into a single result list. Appendix B has an alphabetical listing of descriptions of primitives, programs, and functions which includes the series primitives.

To be "optimizable" by the series macro package, series expressions must obey certain restrictions; these will be discussed shortly. My system must be integrated with the series package to the extent that it must always produce

[4] In Waters's implementation, there are cases when this is not strictly true. Such cases are irrelevant here, however.

programs that translate into optimizable expressions, assuming the input program corresponded to such an expression. The simplest way of integrating my system with the series macro package is to use the latter as a post-processor: my system performs redistributions, obeying the optimizability restrictions, and then transcribes the final program into Lisp using series expressions where it can be processed by the macro package.[5] More complicated integrations are possible, but they have not been implemented. For example, one could apply the system first to the series formulation, apply the (underlying technology of the) macro package to get a new representation of the program, and then apply the system to the result, possibly getting even more optimization.

3.2.3 Syntactic program restrictions for modeling

Unfortunately, due to the requirement of executability on a real machine, some data types come with restrictions on the way they can be efficiently used in legal programs. That is, programs violating the restrictions may be legal (i.e. operationally meaningful) yet be impossible to implement efficiently in code. Both stores and series have such restrictions. I assume that the input program structure has no unoptimizable configurations in it (see below); thus, the system must only keep from introducing any during the optimization process.

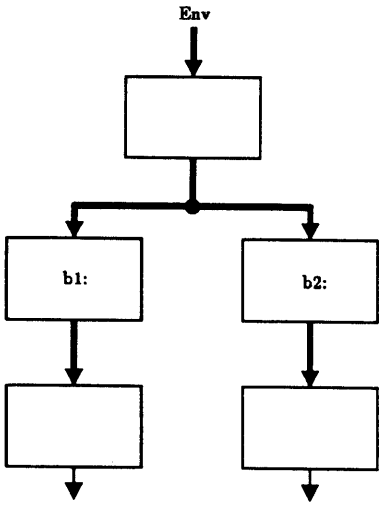
I further assume that whenever the programmer uses a given restricted type in a program, he has supplied an effective and efficient predicate that returns whether a given program satisfies the legality restrictions for that type.

Store restrictions.

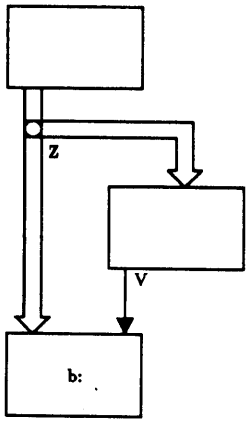
Implementing two different simultaneous stores would be prohibitively inefficient, since these correspond to the memory of the machine; therefore, boxes that alter the store must do so sequentially.

While a programmer would not ordinarily write a program that violates this condition, it is possible for my system to consider "optimizations" that do. Thus, it is important to be able to rule out such candidates, because

[5]The back-end transformation into Lisp-with-series expressions has not been implemented as yet, but is straight-forward.



(a)



(b)

Figure 3.5: (a) An unoptimizable program involving store arcs. Both boxes `b1` and `b2` may alter the same store. (b) An unoptimizable program involving series expressions. Non-series value `V` can only be computed after having seen all the elements of the series `Z`; thus a costly intermediate datastructure must be built to represent `Z` for use by the call to box `b` which can only occur after `V` is computed.

even if they eliminate boxes from the design, they are unlikely to result in an overall improvement.

I conservatively assume that any box with a store output may alter "the store," i.e. it may put out a store that is not equal to its input store. On the other hand, a box that puts out no store cannot alter "the store." Note that a box may take in a store without putting one out. Consider Figure 3.5 (a). This program schema is unoptimizable (hence illegal) because it allows both boxes `b1` and `b2` to alter the same store.

Here are the *store restrictions*:

- If a pair of physical boxes with store output ports have inputs connected to the same store source, then they must have conflicting conditions; i.e., there must exist some conditional test port true in one condition and false in the other. This is a conservative way of guaranteeing that boxes modifying the store during a given execution must do so sequentially.
- A box may have at most one physical store input port.
- A box or conditional may have at most one physical store output port.

It is straight-forward to code a checker for these conditions.

Series restrictions.

Series flow arcs have more complex optimizability restrictions, but there are efficiently checkable restrictions that work for many cases. I will summarize the issues here; fuller discussions appear in Appendix F and in the discussion of Waters's series macro package in Appendix A of (Steele, 1990).

A programmer may inadvertently write programs that violate the series restrictions, so the checking serves to detect unoptimizable input programs as well as to avoid optimizations that result in unoptimizable programs. Waters points out that most user programs violating the restrictions can be simply rewritten to obey the restrictions.

There are basically two restrictions defined by Waters that are significant to my system (I have restated them somewhat):

- The conditions of source and target of a series flow arc must be the same. This is because the Series Macro Package requires that series expressions not contain conditional branches.

- Waters defines a *constraint cycle* as "a closed oriented loop of data flow arcs such that each arc is traversed exactly once and no non-series arc is traversed backward (Steele, 1990, p. 946)." The optimizability restriction is that the program may have no constraint cycles passing through either non-series ports or "off-line" ports. Intuitively, a port is off-line if its values cannot be consumed in lock-step fashion relative to the other series ports around it. This is a static property analogous to a port's static type.

Figure 3.5 (b) shows a program schema that violates the second criterion. The cycle contains a non-series flow arc. I have not as yet incorporated automatic checking into the system, though such would be a straight-forward re-implementation of Waters's techniques. Moreover, the checks are easily computable, so would not significantly impact the run-time of the system. I have hand-checked the redistributions produced by the system, and only one (out of hundreds) was illegal, violating the first criterion.

Note that the term "data flow" used above does not include store flow, so such arcs are not included in the constraint cycle calculations. This leads to some subtle semantic issues which I discuss in Appendix F.

3.3 Redistributions

Given the definition of the structural representation of programs, it is now possible to define "redistribution" precisely. This single type of structural program change is expressed as a rerouting of flow arcs.[6] Such an operation will be termed a *redistribution*; an example appears in Figure 3.6. Here, I've shown the redistribution, from the source `$APPEND.CPY.ZCC.ZOUT` to the target `$APPEND.NC.LC.COLL.Z`, as a single arc for clarity of exposition. When carrying out this redistribution, however, the system actually installs several arcs, some new output ports, and some new input ports. The actual altered program appears in Figure 3.7. The notation in Figure 3.6 is simply "syntactic sugar" for Figure 3.7 and will usually be preferred. It will be useful to call the collection of new arcs and ports introduced by a redistribution an *extended* flow arc.

[6] "Flow arc" is generic for data flow, store flow, and series flow. It does not include control flow, as such arcs are not explicit in the system's representation.

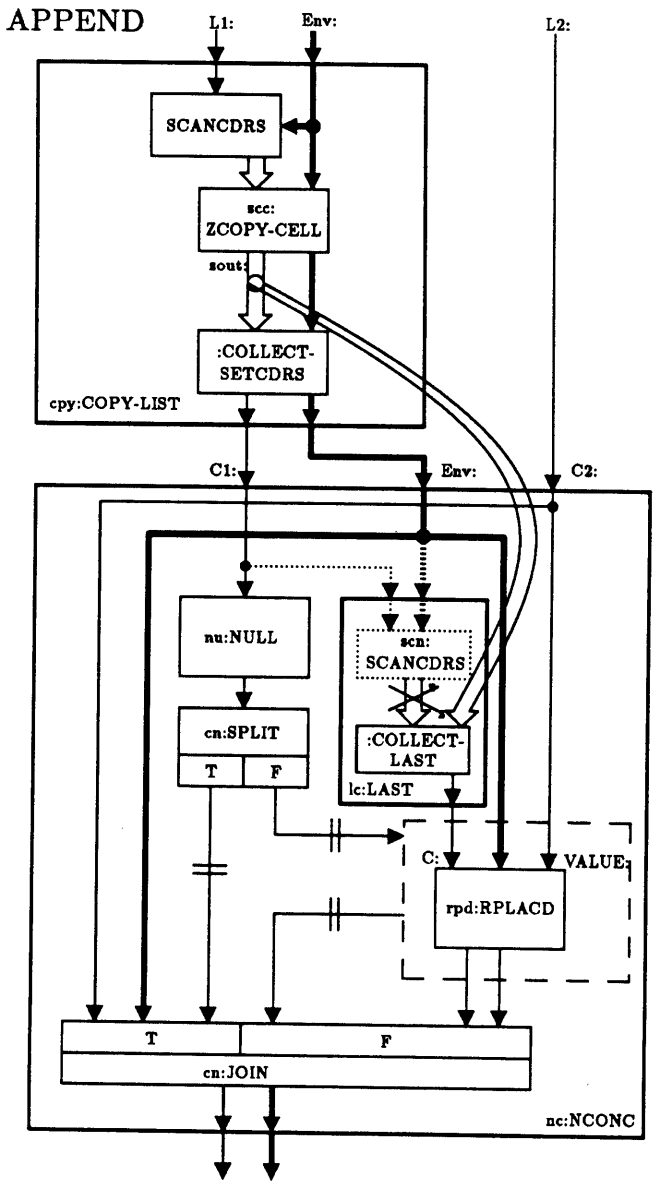


Figure 3.6: A redistribution example. In this case, a series-valued source, `$APPEND.CPY.ZCC.ZOUT`, is shared with the distant target `$APPEND.NC.LC.COLL.Z`. In the process, an old arc is removed and some structural elements become virtual.

An alternative to adding several extra ports and arcs would be to extend the program formalism to allow arbitrary flow arcs to enter, leave, and cross program bodies, thus allowing any redistribution to be implemented by a single arc. Such an approach would, however, always be equivalent in this system to adding a new port, etc. Thus, it would not save any effort (conceptual or otherwise) and would make the definition of program more complicated.

Note that the alterations made by the system take effect only in local copies of the routines effected. In particular, if APPEND (or any other program) were to have another box of type NCONC it will not be effected by the changes to the version corresponding to $APPEND.NC. The exception to this rule is that any *recursive* program effected by the change is altered at all levels of the recursion, not just the top-level.[7] Thus, the optimizations in MY-REVERSE took effect at all levels of the recursion, rather than being restricted to the first level of recursion.

Here are the steps that go into carrying out a redistribution from source s to target t, assuming such has been checked for consistency with store and series constraints:

- Remove the old arc ending at t.
- Implement the new extended arc from s to t, introducing new arcs and ports as needed, including those needed to support recursion.
- Mark structural elements virtual as necessary at all levels of the program.

3.4 Executing Programs

In order to evaluate the truth of the optimization invariants that determine whether a program change is allowed, the system must calculate new output values for the optimized program on the given test cases. To find the new output values, the system must be able to evaluate programs on data values. More than this, however, the system needs to be able to record intermediate values computed along the way for later use by IEBR in evaluating target conditions. Thus, the system contains an interpreter that operates directly on

[7] Actually, even this exception can be turned off when desired.

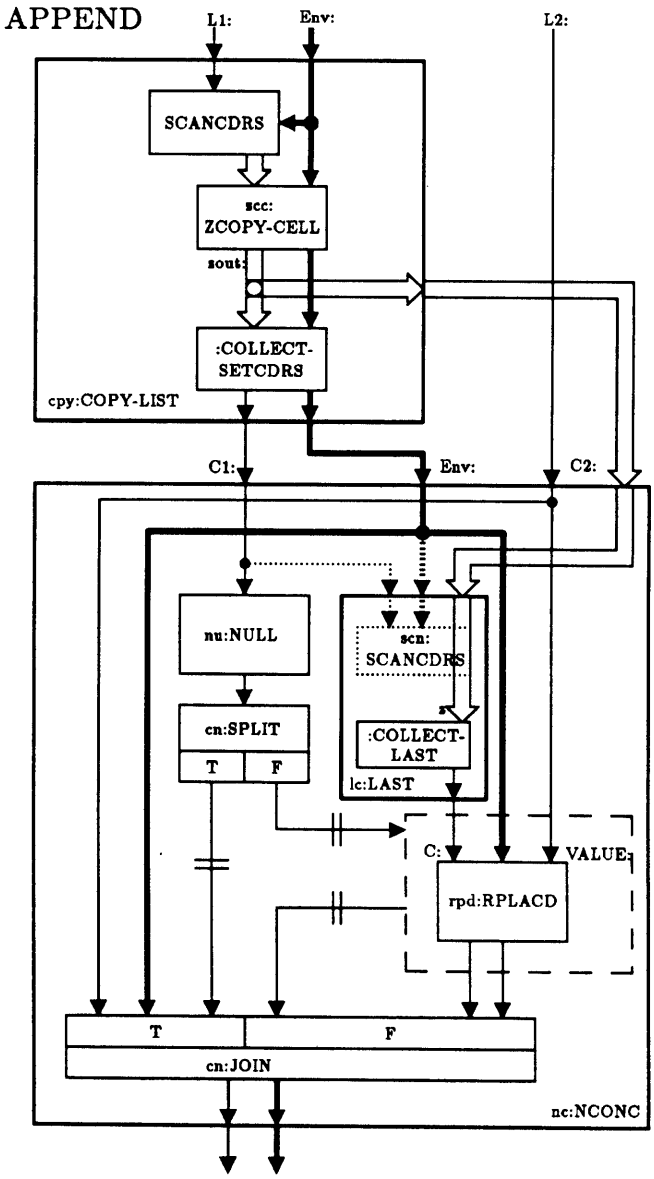


Figure 3.7: This is the actual structure produced when the system carries out the redistribution indicated in the previous figure. Removed arcs have been removed; virtual structure remains but is now dotted.

the internal program representation. It is capable of recording intermediate program values in a data structure called a *program trace.* This capability forces the system to represent all data objects explicitly; in particular, series and stores must now have first class representations that can be stored. This means, of course, that the interpreter will be much slower than the compiled version of the program, but we are using the two for different purposes. The interpreter is used only for ground reasoning about test cases; the "production" version of the optimized program will be produced by translating to Lisp and running various compilation routines such as the series macro package and the Common Lisp compiler.

3.4.1 Structure of the Interpreter

To interpret program `P` on some input values, `P` is first preprocessed to determine an *execution order* for the boxes and conditionals. The boxes and conditionals are topologically sorted in a manner consistent with the causality relation, with causal predecessors before successors. The execution order is determined once and only recomputed if the program is altered during optimization.

Once the execution order is known, the inputs are initialized to their values and then the boxes and conditionals are evaluated in order.[8] Boxes are evaluated one of two ways: if the box's type is a *computational primitive* (see next subsection) then the corresponding primitive evaluation function is called; otherwise the box's type program is executed recursively using the input values read from the box's input ports. A conditional is evaluated as follows: if the test port evaluated (earlier) to true, then the conditional outputs are copied from the conditional's true-output ports, otherwise the conditional outputs are copied from the false-output ports. Note that since a box's condition must be true before it can be evaluated, boxes on the "wrong side" of the conditional are not executed, because their conditions are not satisfied. Once the output ports of a box or conditional are determined, the values are (implicitly) copied to the other ends of all flow arcs connected to them.

Note that virtual elements are evaluated in precisely the same way as

[8]Note that, because stores are treated explicitly, this is equivalent to the intuitive parallel execution model given earlier.

physical elements; the fact that their outputs are not used is irrelevant to the interpreter. It is only when final code is produced for the fully optimized program that virtual elements are eliminated.

3.4.2 Computational Primitives

The recursion in the interpreter must bottom out in primitive programs for which the system is given algorithms. This set of primitives is one way in which domain knowledge is encoded in the system. Appendix A lists those I used in experiments. They include programs for allocating, accessing, and modifying memory cell arrays, arithmetic operations, and series primitives.

3.4.3 Program traces

If the interpreter's recording option is selected, then each value computed is recorded in a *program trace*, a table keyed by global pathname. Elements at lower levels of the structure hierarchy (within box implementations) are recorded using their unique global pathnames referred to the top-level program, rather than their pathnames inherited from their direct program parent. Thus, a program trace for APPEND could store the entry

$$\texttt{\$APPEND.NC.LC.SCN.Z} \longrightarrow \texttt{\#Z<cell-2 cell-1 cell-0>}$$

but *not* the entry

$$\texttt{\$LAST.SCN.Z} \longrightarrow \texttt{\#Z<cell-2 cell-1 cell-0>}.$$

Trace depth. A program trace also records a small amount of extra information. One important such datum is the maximum depth of recursion reached by the program on that input. When IBR blindly tries each redistribution, it is in great danger of creating a non-terminating program. The interpreter can be set to terminate abnormally when updating a trace if the recursion level ever exceeds the previously stored recursion level. IBR then knows that it should avoid the redistribution causing this because, even if the computation should eventually terminate normally, it will probably not be any more efficient than the original. This heuristic has worked out well in practice.

3.5 Programs versus Plans

To complete my description of the program structure representation, I will briefly compare it to the most similar such representation in the literature, the Plan Calculus (Rich & Waters, 1990).

The differences between a "program" in my terminology and a "plan" are listed below.[9]

- Flow arcs in programs imply only "shallow" equality, whereas such arcs in plans imply "deep" equality. The difference shows up in the treatment of mutable objects, such as memory cells. In my formalism, a dataflow equality between two cells implies only that the pointers are identical—it says nothing about the values bound to the cell. In the Plan Calculus, the equality implies not only equality of pointers but also that the abstract objects represented by the cells are equal. Shallow equalities have several practical advantages, the most important of which is that it is always easy to translate from programs with shallow flow arcs to source code and back again. It can be arbitrarily difficult to translate deep equalities into efficient code due to the difficulty of deciding whether copying must be done to maintain the deep properties across intervening destructive operations. A point in favor of deep equalities, on the other hand, is that they capture the designer's intent more closely, facilitating cliche recognition for example.

- Plans allow logical constraints among port values that are not implied by either dataflow or box input/output specifications. For example, two different box outputs may be constrained in the plan to be equal, even if the specifications of the boxes are logically insufficient to imply it. Programs do not allow such "extra" constraints. Disallowing such constraints has the benefit of guaranteeing the useful property that "any implementation of a box that provably satisfies the box's specification may be used to implement it while still maintaining correctness of the overall program." Several algorithms in the system depend on

[9]These are with respect to the version of Plan Calculus reported in (Rich & Waters, 1990). Since the publication of that book, the authors have modified the semantics of plans to allow more expressive power. My system most closely resembles the original, eager-semantics form.

this property for correctness. Plans do not have this property. The extra constraints allowed in plans are intended to capture partial designs more flexibly.

- Control flow arcs are allowed in full generality in plans; programs use them only in implementing conditionals. I have not yet investigated rerouting explicit control flow arcs, though it may allow added generality.

- Plans use "situations" to model control flow ordering, mutable object state, and non-termination. By contrast, the only control flow in programs is implicit in conditionals; stores model mutable object state; and non-termination can be modeled as a separate ("bottom") type.

Chapter 4

Program Function

By "program function" I mean logical properties that are true of the computation performed by the program. These properties may refer to intermediate values within the computation as well as top-level input and output values. The system uses representations of program function to help find and to help justify optimizations. Here are some informal examples of program function statements:

- `COPY-LIST` does not change the contents of any cells allocated in its input store.
- The value produced at the internal port `$MY-REVERSE.CONC.L2` is a fresh, length-one Lisp list whenever the (value of) `$MY-REVERSE.L` is a non-empty, finite Lisp list.
- The output list of `COPY-LIST` must be abstractly equal to the input list.

The system accepts, constructs, and uses representations of statements such as these. This chapter defines the system's representation of them. The formalism is designed both to have sufficient expressive power to represent partial correctness properties of programs and to support the manipulations performed by the system, the most important of which is proof generalization (discussed in Chapter 5).

4.1 Terms and Clauses

Program function is expressed as a collection of *clauses*, each of which expresses a constraint among some program port values. A clause is a set of boolean-valued terms, representing the logical disjunction of the terms. Terms represent standard logical terms from predicate calculus.

4.1.1 Syntax

The system's logical notation is essentially second-order predicate calculus without existential quantifiers. Except for free variables in the function position, it is similar to the logic of the Boyer-Moore theorem prover (Boyer & Moore, 1988). I present the details both for completeness and to fix notation.

A *term* is either *atomic* or *compound*. Compound terms are formed syntactically by surrounding a sequence of terms with parentheses. The terms making up a compound term are called its *subterms*. Atomic terms have no subterm structure, and their names fall into three categories: function symbols, free variables, and pathname terms.

Function symbols are character strings not containing spaces and *not* beginning with a question mark. Examples include `NIL`, `+`, `=`, and `A?`. The single-character identifier `$` is not allowed as a function symbol.

Free variables are written as alphanumeric identifiers whose leading character is a question mark, such as `?N` and `?4`. Note that free variables may appear in the function position of a compound term.

Pathname terms are in one-one correspondence with program structure pathnames, and are denoted in exactly the same way as pathnames.

A *negated term* is a term of the form `(NOT t)` where `t` is another term. The *negation* of a term is `(NOT t)`, except that multiple negations are canonicalized so that not more than one `NOT` may appear in a row.

A *clause* is a set of terms (some possibly negated), written in standard set notation. Here are some examples of clauses:

- `{?X}`
- `{(NOT (CHI-C ?C)), (> (CSIZE ?C) (ZERO))}`
- ```
  {(NOT (CHI-C ?C)),
   (NOT (A? (V ?C $P.ENV))),
   (= (Q (V ?C $P.ENV)))}
  ```
  ```

4.1.2 Semantics

I will define the meaning of a clause by giving a mapping M from clauses to sentences of first-order predicate calculus. I will then assume that the reader is familiar with the standard semantics of such sentences.

Define the action of the mapping M on a term or clause as follows:

- A compound term is mapped to the corresponding logical term obtained by mapping each of the subterms using M and applying the first mapped subterm to the rest. Thus,
  $M(\texttt{(t1 t2 t3)}) = M(\texttt{t1})(M(\texttt{t2}), M(\texttt{t3}))$.
- Function terms are mapped to distinct logical function constants.
- Free variable terms are mapped to distinct logical variables.
- Pathname terms are mapped to constant symbols with names functionally derived from the pathname. These must be disjoint from the constant symbols used for function symbols.
- A clause is mapped to the universal closure of the disjunction of the mappings of its constituents.

Here are the mappings of the three clause examples in the previous subsection:

- $\forall x\ x$
- $\forall c\ [\overline{\chi_C(c)} \vee (\mathrm{CSIZE}(c) > 0)]$,
  which is commonly written $\forall c\ .[\chi_C(c) \Rightarrow (\mathrm{CSIZE}(c) > 0)]$.
- $\forall c\ [\overline{\chi_C(c)} \vee \overline{\mathrm{A?}(V(c, \boxed{\texttt{\$P.ENV}}))} \vee Q(V(c, \boxed{\texttt{\$P.ENV}}))]$,
  which is commonly written
  $\forall c\ [\chi_C(c) \wedge \mathrm{A?}(V(c, \boxed{\texttt{\$P.ENV}})) \Rightarrow Q(V(c, \boxed{\texttt{\$P.ENV}}))]$

$\boxed{\texttt{\$P.ENV}}$ denotes the logical constant corresponding to `$P.ENV`. From now on, I will always use the unboxed form for pathnames, as no confusion can arise.

4.1.3 Ground Evaluation of Clauses

The system needs to be able to evaluate *ground* instances of clauses in a way consistent with the semantics just given. In particular, it needs to be able to decide whether a given clause is true, false, or unknown when all pathname terms are replaced by concrete data values calculated by a computation. One application of this is in IBR, where the top-level specification is evaluated on the inputs and outputs of a test-case in order to see whether a proposed optimization maintains correctness on that test-case. Another application is when IEBR evaluates target conditions using trace data to evaluate a proposed optimization. The system, therefore, includes a *clause evaluator* that is capable of performing this evaluation. The clause evaluator takes in the clause and a program trace and puts out true, false, or unknown.

The clause evaluator consists of a universal instantiator and a term evaluator. The universal instantiator attempts to convert a clause containing free variables into an equivalent finite set of clauses free of variables having the property that the clause is true if and only if every set member is true. If it fails to do so, the entire clause evaluates to unknown. To evaluate a variable-free clause, the system calls the term evaluator on each term of the clause until one returns true. If one is true, then the clause is true; if none are true, but some are unknown, then the clause is unknown; otherwise the clause is false.

The clause evaluator encodes domain knowledge in the term evaluation functions and universal instantiators; this is another place that domain knowledge must be entered by the user.

Universal Instantiation.

User-supplied instantiation experts perform the instantiation of the universally quantified free variables in a clause. Each expert examines the clause, possibly calling the term evaluator on variable-free terms, and decides whether it knows how to instantiate one or more of the variables in it. If so, then the expert returns a mapping of variable(s) to value-list(s) that defines the set of concrete clause instances to evaluate. If more than one expert tries to instantiate the same variable, one is chosen arbitrarily. Once all the variables are eliminated by some collection of experts, the variable-value maps are composed and returned to the clause evaluator, which then tries

each combination of values sequentially.

It is impossible (or impractical) to instantiate a quantification over an infinite or large domain. Thus, a clause like {(= (+ ?N ?N) (* 2 ?N))} cannot be handled soundly by the system. Fortunately, bounded quantification over relatively small domains is often sufficient to express the necessary constraints.[1] Here are some examples in both English and clause form, where I've highlighted portions corresponding to the bounding of the quantification:

- Every *member of the input list* is a member of the output list:

```
{(NOT (MEMBER? ?X $P.Lin)), (MEMBER? ?X $P.Lout)}
```

- Every *cell allocated in the input store* maintains its binding in the output store:

```
{(NOT (CHI-C ?C)),
 (NOT (A? (V ?C $P.ENV))),
 (= (V ?C $P.ENV) (V ?C $P.P-ENV))}
```

- The kth entry of the list is a number for every number[2] *k less than the length of the list*:

```
{(NOT (< ?K (LENGTH $P.Lin))), (CHI-N (NTH ?K $P.Lin))}
```

In the first example, an expert would notice the form of the first term of the clause, instantiating `?X` by all the members of the list to which `$P.Lin` evaluates in the current test-case. In the second example, a different expert would notice the form of the second term and instantiate `?C` with all cells allocated in the input store. In the third, the system would instantiate `?K` by the integers $0 \ldots l-1$ where l is the length of the list.

Term Evaluation.

Term evaluation is performed recursively. An atomic term is evaluated according to its type: a pathname term is looked up in the current test-case

[1] When large domains are required, instantiation experts can be coded that return a list of representative values rather than all values. This hasn't been necessary for the examples I've run so far.

[2] In the domain knowledge given the system for the experiments the only "numbers" are the nonnegative integers.

trace; a function term evaluates to an effective subroutine that can compute it via lookup in a user-supplied table of primitive *term evaluation functions*; a free variable evaluates to unknown, because it should have been instantiated away before term evaluation. A compound term is evaluated by first evaluating its subterms and then applying the subroutine corresponding to the evaluation of the function subterm to a list of the evaluated other subterms. If the function evaluates to unknown, then the entire compound term does as well. Term evaluators may return unknown.

Note that the data values are the same as those produced by the program evaluator. Also, if a function symbol term has the same name as a computational primitive program, then the primitive's (user-supplied) program evaluator is called. This allows for greater consistency and sharing between the program evaluator and the clause evaluator.

4.2 Optimization Invariants

Having defined a representation for program function statements, I will now define, in terms of this representation, one of the key items of information that must be input by the user of the system, the optimization invariant. Later (see Chapter 9) I will define another class of function statement, target conditions, that is derived and used internally by the system.

Informally, a program's *specification* is a precise statement of the problem it is required to solve. Formally, the system represents a program specification as a collection of clauses referring only to program input ports and output ports.[3] If for all input values the program's computation always satisfies all clauses of the specification, I will say that the program is correct with respect to that specification.

A specification is a special case of *optimization invariant*, a logical statement constraining the input/output functionality of a program. More generally, optimization invariants may state constraints among a program's inputs, its *original* (i.e. before it was changed by the optimizer) outputs, and its *current* outputs (after the change). Typically, optimization invariants capture the constraints required of a program for correctness in some larger context. As such, these are what must be preserved by the optimization process.

[3]In Chapter 10, it will be useful to define the notion of a *quasi-specification*, which *is* allowed to refer to values of internal elements of the program.

Representing optimization invariants for a program is crucial, because a designer (in particular, an optimizer) that knows them has more freedoms than does one ignorant of it. An optimizer may change the original design even to the extent that the program gives different output values for given inputs as long as the program is still correct with respect to the invariants. As discussed earlier, traditional optimizers accept only the program's structure as input, having no access to more liberal constraints. This severely limits the range of optimizations allowed, because the program must remain provably correct with respect to all possible specifications satisfied by the program's structure.[4]

Note that optimization invariants expressed in terms of the original outputs of the program give one a way to express *incomplete* specification information, by avoiding the need to formalize some properties of the outputs in terms of the inputs alone.

The system represents optimization invariants as a collection of clauses (implicitly conjoined). A special pathname syntax allows one to refer to the original output values of a program: `$P.0.ORIGINAL` refers to the original (unoptimized) output value corresponding to the program output `$P.0`.

There are both theoretical and empirical reasons to believe that it is significantly easier to produce relative optimization invariants than those that may not refer to the original outputs, which are essentially full formal specifications. In general, any problem that represents a single-valued function, such as any decision problem, is trivial to check using the original outputs—simply compare the new outputs to the old outputs for identity. On the other hand, one can show that for decision problems, the checking problem (without reference to original inputs) is equivalent to computing the problem; of course, there are decision problems of arbitrary complexity. Appendix E discusses the theoretical issues in more detail.

In addition to these theoretical considerations, practical experience indicates that relative optimization invariants are much easier to produce and use, as well. All of the `LR1` examples (see Appendix C) run on the system used essentially *the same* optimization invariants, modified only to reflect different input and output names! Even with this somewhat restrictive information,

[4]This latter condition is not the same as "... must keep exactly the same output values." Language primitives only guaranteed to satisfy underdetermined specifications, such as random functions or Lisp's *union*, allow some variation in the output even in complete ignorance of the program's true specification.

the system was able to find the optimizations of interest. A similar story applies to all SR examples, as well.

4.3 An Optimization Invariant Example

COPY-LIST takes in a pointer to a memory cell and a store and returns a memory cell and a store. Informally, it must be side-effect free, and whenever its output is a list of memory cells they must be fresh (newly allocated) with respect to the input store. Furthermore, when the input is a NIL-terminated CDR-chain, i.e. a legal Lisp list, the output must represent the same list, that is, corresponding CAR pointers must be identical. Here are the clauses, together with English paraphrases.

- ```
  {(NOT (CHI-C ?C)),
   (NOT (A? (V ?C $COPY-LIST.ENV))),
   (= (V ?C $COPY-LIST.ENV) (V ?C $COPY-LIST.ENV-OUT))}
  ```

  This says that for every cell allocated in the input store of COPY-LIST, its binding in the input store must equal its binding in the output store. CHI-C is a characteristic predicate for the memory-cell type. The function V dereferences a memory cell pointer in a store, returning a tuple (in this example, a pair) if and only if the cell is allocated. The predicate A? decides whether the argument is a tuple or not; this is used for telling whether a cell is allocated. $COPY-LIST.ENV is the input store of COPY-LIST, while $COPY-LIST.ENV-OUT is its output store.

- ```
  {(NOT (CHI-C ?C)),
   (NOT (SPINE? ?C $COPY-LIST.C-OUT $COPY-LIST.ENV-OUT)),
   (NOT (A? (V ?C $COPY-LIST.ENV)))}
  ```

  This is the freshness condition. It says that every cell in the spine of the output list, i.e. all those accessible via a sequence of CDR operations from the output list pointer, must not be allocated in the input store.

- ```
  {(NOT (LR1? $COPY-LIST.C $COPY-LIST.ENV))
   (= (LR1 $COPY-LIST.C $COPY-LIST.ENV)
      (LR1 $COPY-LIST.C-OUT $COPY-LIST.ENV-OUT))}
  ```

  This expresses the abstract list function of the program. LR1? denotes a predicate that is true if and only if the input cell and store pair represents a finite list; LR1 (no trailing question mark) is an abstraction
  ```

function that maps the cell and store pair to the corresponding abstract list. Thus, the clause says that if the input pair represents a list, then the abstracted output must be equal to the abstracted input. Note that the list elements are not abstracted; hence the abstraction is not the same as that captured by the Lisp function EQUAL.

Note that the last property above could easily have been represented relatively by replacing it with the following.

- ```
  {(NOT (LR1? $COPY-LIST.C $COPY-LIST.ENV))
   (= (LR1 $COPY-LIST.C-OUT.ORIGINAL $COPY-LIST.ENV-OUT.ORIGINAL)
      (LR1 $COPY-LIST.C-OUT $COPY-LIST.ENV-OUT))}
  ```

  This expresses that the output, viewed as an abstract list, must equal the original output, also viewed abstractly. For COPY-LIST, whose list functionality is trivial, this is not obviously easier than the direct formalization. Consider, however, replacing COPY-LIST with REMOVE-DUPLICATES. Using the relative version of the third property, one could simply convert the above using textual substitution. The absolute formalization of the third property, however, would have to be completely different and much more complicated.
  ```

Chapter 5

Proofs Connect Structure to Function

The IEBR algorithm (see Chapter 9) exploits the intuitive idea that a human programmer uses knowledge of how a program's structure implements its function to help suggest and reject candidate optimizations. One way to represent this knowledge is as a correctness proof demonstrating that the program's structure implies its specification. This is certainly not the only possible representation, and it will probably not capture all of the programmer's teleological knowledge, but it does have the following advantages:

- Proofs are well-defined formal objects, familiar to almost everyone.
- A proof contains enough information to derive approximate target conditions (see Chapter 9).
- There is a relatively inexpensive way, using a kind of explanation-based generalization, to derive target conditions from proofs.

The proof formalism defined below was designed to be powerful enough to express interesting properties, yet simple enough to yield a relatively simple generalization algorithm.

5.1 Proof Syntax

A *proof* is a finite, rooted tree. Each legal proof-tree represents a proof as in first-order predicate calculus. Each node has a clause associated with it

that states the assertion proved by that subtree. Each node also has a type ("inference type") that indicates which inference step was used to deduce the clause from those of its children, or what basis was used for assuming the clause, if the node is a leaf. Each node may also have additional type-dependent information, as discussed below.

If a node's clause does follow from its children by the stated inference step, or if it is an allowable leaf clause, I will term the node *legal*. If each node in a tree is legal, then the clause at the root follows logically from (is entailed by) the conjunction of the clauses at the leaves. Thus, to define the proof-tree representation, I need only define the node types and the legality criteria.

Some nodes must have a *proof identifier* which is a symbolic name for the proof, used to access the *proof library*, a mapping from proof identifiers to proof trees. Other nodes may or may not have identifiers. Proof identifiers are provided both for convenience and documentation purposes.

5.1.1 Proof Nodes: Leaves

There are five types of leaf nodes.

:TAUTOLOGY Nodes

:TAUTOLOGY nodes may only contain clauses that are logically valid *in the theory of propositional calculus with ground equality*. Such clauses are easy to check by standard congruence closure techniques (Downey, Sethi, & Tarjan, 1980). I will term such tautologies *PE-tautologies*, for Propositional-plus-ground-Equality. Here are some examples of PE-tautologies:

- $a \vee \bar{a}$
- $(a \neq b) \vee a \vee \bar{b}$
- $(a \neq b) \vee (f(a) = f(b))$

Any node whose clause represents a PE-tautology is automatically converted to a :TAUTOLOGY node by the system. This type of node should have no identifier.

:AXIOM Nodes

:AXIOM nodes state domain facts, such as "1 is the successor of 0." The only legality checking done at this type of node is to ensure that the clause is not a tautology and that it has an identifier for documentation purposes. If it is a tautology, the node is converted to a :TAUTOLOGY type node.

:FORWARD Nodes

:FORWARD nodes state arbitrary facts. They are checked for legality in precisely the same way as :AXIOM nodes; I use this node type to represent facts that are believed true, either to be proved later or simply assumed because the user's input program is assumed correct. The primary use for this node type is in incomplete proofs, where we assume certain facts about the user's program because they are too hard to prove.

:DEFINITION Nodes

:DEFINITION nodes are used to define (predicate) names to stand for clauses. Thus, they have extra information in addition to the clause: there is a (new) predicate identifier and an ordered list of formal arguments. The clause is that for which a predicate application stands. For example, suppose we have a :DEFINITION node with predicate identifier P, argument list (?X, ?Y) and clause {(NOT (A ?X)), (B ?Y)}. This means that for any terms t_1 and t_2 the term (P t_1 t_2) is logically equivalent to the clause {(NOT (A t_1)), (B t_2)}. These nodes can only be used as children of :DEFINITION-APPLICATION nodes. There is no legality checking other than that the fields are all present.

:PROGRAM-STRUCTURE Nodes

These nodes provide the connection between programs' structure and clauses. There are three subtypes of :PROGRAM-STRUCTURE based on the three different ways of connecting values in a program functionally. The three are distinguished by extra node information as follows:

- Flow arc constraints in a program can be introduced into a proof through the :FLOW subtype. To do this, one must give the source pathname and target pathname. The pathnames must belong to the same

program and the program's structure must contain a flow arc from the source to the target. This information determines the clause completely as follows: disjoin the term (= *source target*) to the negations of each of the terms in the target's condition. For example, if the source is `$P.I.Y` and the target is `$P.J.A` and the target's condition is {`(NOT $P.CN1.TEST)`, `$P.CN2.TEST`}, then the clause is
{`$P.CN1.TEST`, `(NOT $P.CN2.TEST)`, `(= $P.I.Y $P.J.A)`}.

- Constraints introduced by boxes can be used in a proof through the `:BOX` subtype. To do this, one must give the box's pathname and the proof identifier naming the root of the proof-tree that proves some clause of the specification of the box's program type. Note that this must be a specification clause, i.e. it may not refer to internal pathnames of the box's type. Note also that it may not be a relative clause; that is, it may not refer to the "original" outputs. Furthermore, the proof identifier must be declared to be one of the box's specifications. Such declaration is done when the box's type program is defined. This prohibits one from accidentally using unadvertised properties of the box's implementation. As an example, suppose the freshness property of `COPY-LIST`'s output list (see Section 4.3) is proved and the proof has identifier `COPY-LIST-FRESHNESS-SPEC`. Then suppose we wish to use the freshness property in a proof about `APPEND`. We must give the box's pathname `$APPEND.CPY` and the proof identifier `COPY-LIST-FRESHNESS-SPEC` as additional information. This then fully determines the clause of the proof node to be the same as that of `COPY-LIST-FRESHNESS-SPEC`, except the pathnames in the latter (all referred to the program `COPY-LIST`) must be renamed in the former to their global names referred to `APPEND`. Here is the renamed clause:

```
{(NOT (CHI-C ?C)),
 (NOT (SPINE? ?C $APPEND.CPY.C-OUT $APPEND.CPY.ENV-OUT)),
 (NOT (A? (V ?C $APPEND.CPY.ENV)))}
```

- The other subtype of `:PROGRAM-STRUCTURE` allows proofs to refer to the relationship between a conditional's true-output (false-output) and output ports. The only information needed here is the pathname of the desired true-output (false-output) port. The clause is then either
{*test-port*, (= *false-output-port output-port*)}

or
{(NOT *test-port*), (= *true-output-port output-port*)}.

:PROGRAM-STRUCTURE nodes are conventionally without identifier.

5.1.2 Proof Nodes: Internal

There are four types of internal proof nodes. These are (mostly) familiar proof rules from first-order predicate calculus.

:SUBSUMPTION Nodes

A :SUBSUMPTION node is legal if its clause is a superset of the clause of its single child. No other information is necessary. This is sound, as $A \vee B$ obviously follows from A.

:RESOLUTION Nodes

A :RESOLUTION node represents a logical resolution. Let c_1 and c_2 be the clauses of the node's two children. I will say they *resolve* if there is a term t (the *resolvent*) such that t appears in c_1, and the negation of t appears in c_2. Then the :RESOLUTION node is legal if (1) its two children resolve with resolvent t, and (2) its clause is $(c_1 - t) \cup (c_2 - \bar{t})$. Note that it is legal for there to be more than one resolvent, but the resulting clause is always a tautology. Resolution is sound and complete for propositional reasoning.

:INSTANTIATION Nodes

Clauses referring to free variables are supposed to be true with the variable bound to any individual. The :INSTANTIATION node allows one to conclude the specific instance from the clause with the free variable. More precisely, this type of node is legal if and only if there exist terms $t_1, \ldots, t_k$ and distinct free variables $v_1, \ldots, v_k$ such that the clause of the node is equal (as a set) to the clause obtained by substituting each t_i for v_i in the clause of its child.

:DEFINITION-APPLICATION Nodes

These allow replacing a subclause by a single predicate term. This is useful for introducing shorthand notations and for negating clauses. Such a node

has two children, exactly one of which must be a :DEFINITION node. Such a node is legal if there exists a collection of terms $t_1, \ldots, t_k$ corresponding to the k formal arguments of the :DEFINITION's predicate such that if we call the substitution of the terms for the arguments in the clause of the :DEFINITION the *substituted definition*, the following holds: the clause of the :DEFINITION-APPLICATION node must both be a superset of the set-difference of the clause of the non-:DEFINITION child and the substituted definition, and it must contain the single term (P $t_1, \ldots, t_k$), where P is the name of the defined predicate.

As an example, suppose the definition-child is as given previously: predicate name is P, formal arguments are (?X, ?Y), and clause is {(NOT (A ?X)), (B ?Y)}. Further suppose the clause of the non-definition child is {(NOT (A (ZERO))), (B (ZERO)), (G (ZERO))}.
Then a definition-application with these children could legally have any of the following clauses:

- {(P (ZERO) (ZERO)), (G (ZERO))}
- {(P (ZERO) (ZERO)), (B (ZERO)), (G (ZERO))}
- {(P (ZERO) (ZERO)), (NOT (A (ZERO))), (B (ZERO)), (G (ZERO))}

but it could not have any of these clauses:

- {(G (ZERO))} – (missing (P (ZERO) (ZERO)));
- {(P (ZERO) (ZERO))} – (missing (G (ZERO)));

5.2 Proof Example

Figure 5.1 shows a proof of a lemma used by the system. It illustrates six of the nine proof node types. The predicate APURE? is true of a pair of stores if and only if all cells allocated in the first have the same bindings in the second. This allows allocation and alteration of newly allocated cells in the second store. The axiom (used twice) identified as APURE?-def is the "converse" of the definition: whereas the :DEFINITION-APPLICATION replaces the clause by the predicate, the "converse" replaces the predicate by the clause. Conventionally, every defined predicate has an associated converse.

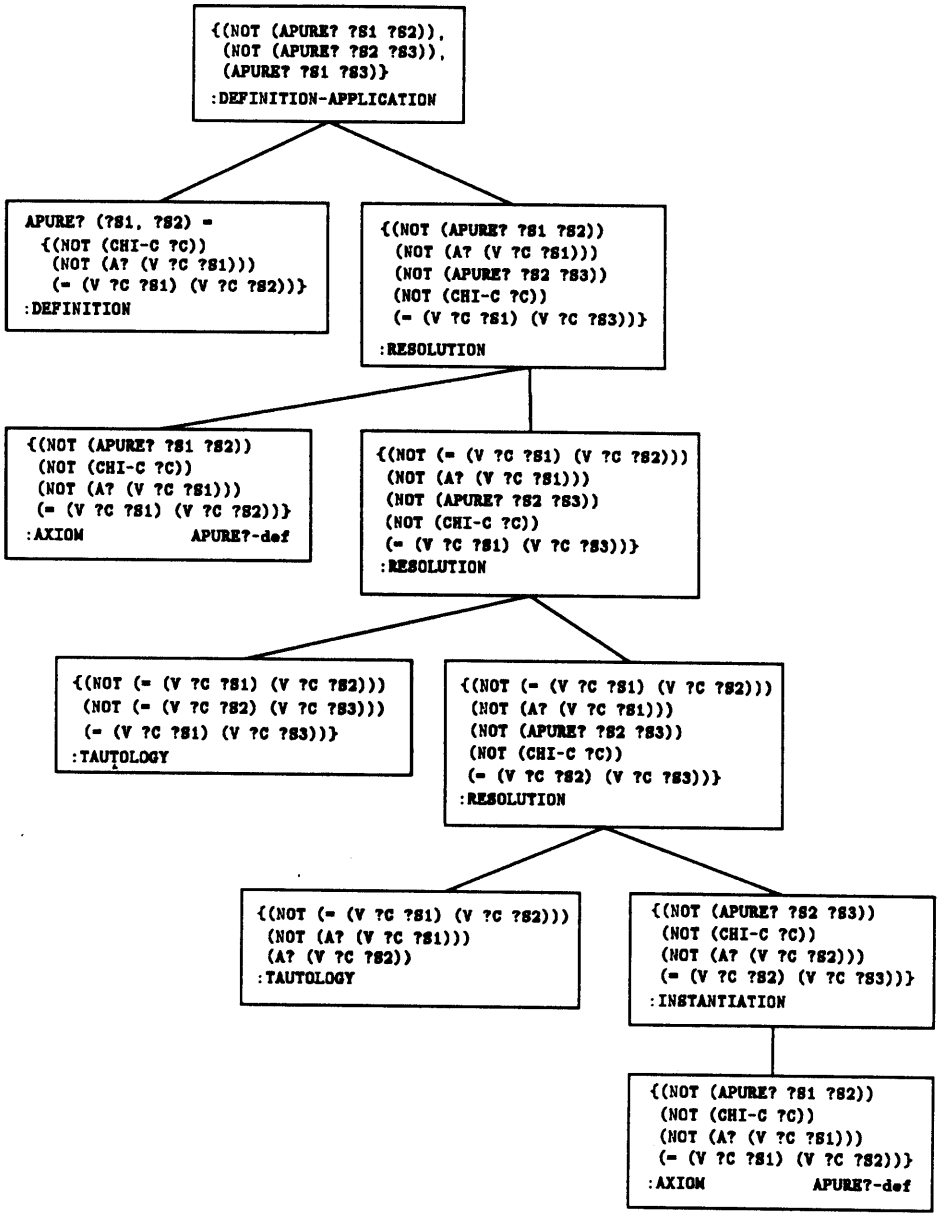


Figure 5.1: A proof of the transitivity of APURE?. This proof illustrates six of the nine node types. APURE? expresses a relationship between stores intuitively stated as "the second is obtained from the first through only allocations and alterations of newly allocated cells."

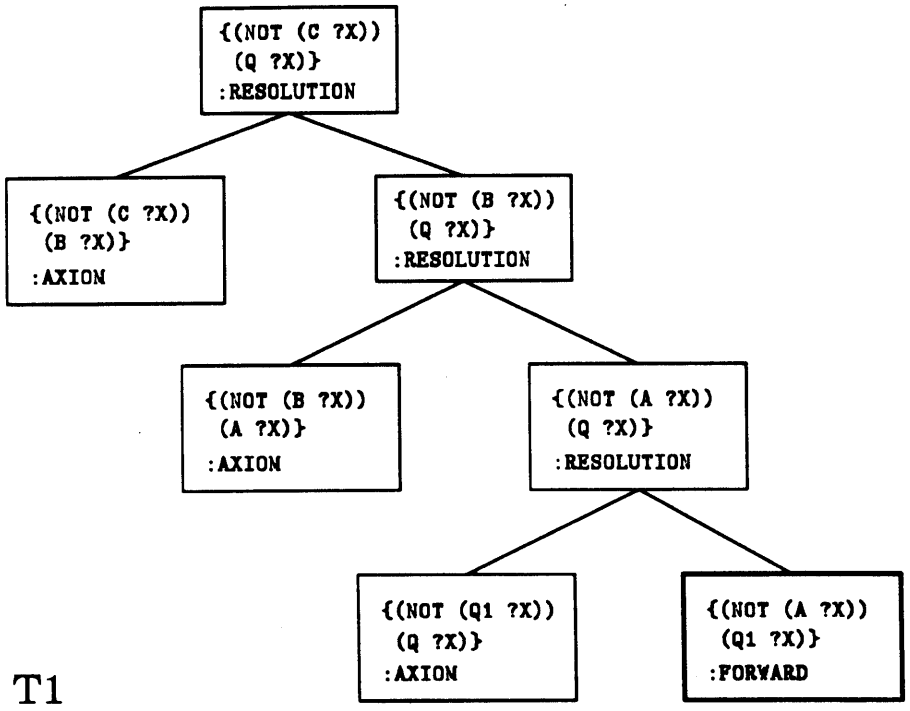


Figure 5.2: A proof, T_1, of the clause "for all x, if $C(x)$ holds then $Q(x)$ also holds". The proof depends on three axioms and a "forward" (assumed) node which is highlighted.

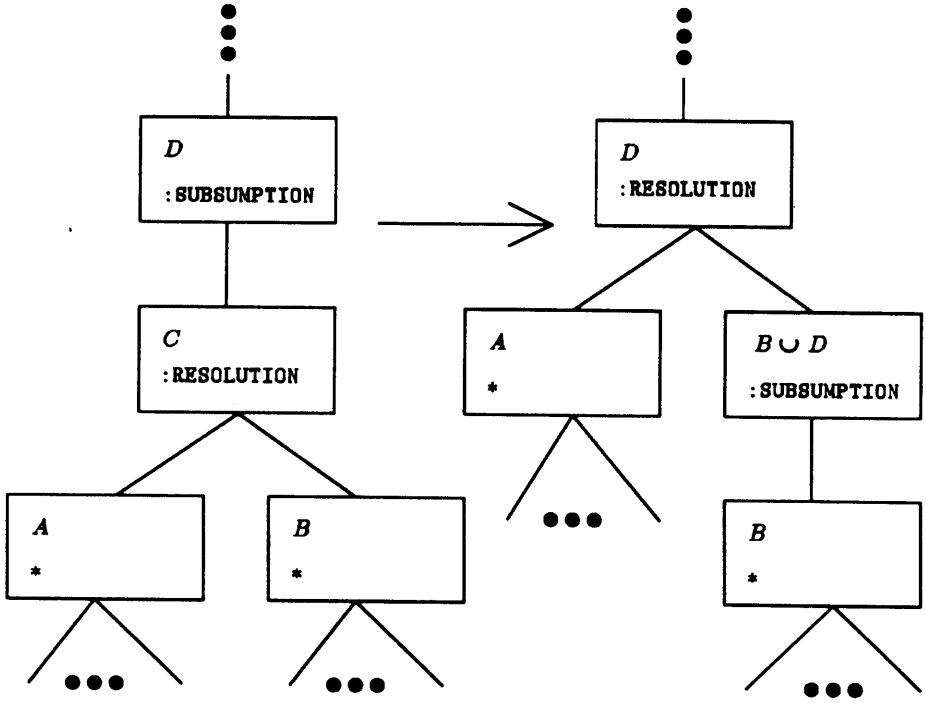
5.3 Weakened Relative Conditions

Consider the proof in Figure 5.2. The :FORWARD node is only assumed; how can we logically weaken its clause (and possibly others in the tree) and yet still be able to prove the desired conclusion at the root of the tree? This will turn out to be useful for deriving conditions used for finding and proving optimizations—weaker conditions allow more potential optimizations, because more objects will satisfy them. This section gives a technique for weakening conditions at leaves of proof trees; Chapter 9 explains how the technique is applied to program optimization.

One method for weakening is based on the following

Unioning Transformation: A proof having a subsumption

node, one of whose children is a resolution node, remains legal after the following local transformation, where starred nodes may be of any type:



By symmetry of the resolution rule, either child of the resolution node may be used, or both by using the transformation twice. The transformation is defined by the fact that the clause of the subsumption node on the right is the union of the root's clause and the clause of one of the resolution's children. Note that the root's clause and the two original children's clauses are unchanged.

We can apply this to an entire path from the root of a resolution proof to any particular leaf by first replacing the root with a trivial subsumption, one with parent and child clauses identical, and then applying the unioning transformation sequentially until the subsumption node is just above the bottom of the chosen path. The bottom of the path may then be replaced with any tree proving a subset clause of the subsumption node's clause.

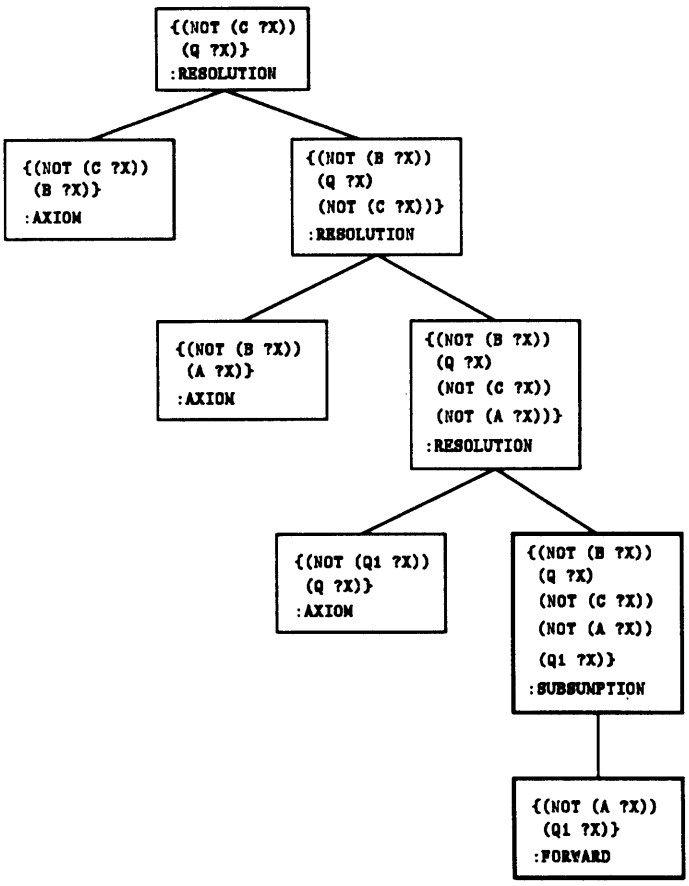


Figure 5.3: Applying the unioning transformation. This is the tree from Figure 5.2 after applying the unioning transformation from the root down to the :FORWARD leaf. Note that the tree remains legal (exercise for the reader). It is potentially desirable if we want to find a weaker condition to replace the :FORWARD leaf.

Figure 5.3 shows the result of applying the unioning transformation to the example of Figure 5.2. Clearly, we may replace the :FORWARD leaf by any node whose clause is a subset of the clause of the :SUBSUMPTION node. Note also that the :SUBSUMPTION node's clause is always a superset of the original leaf's clause, by construction. Thus, it is always logically entailed by (hence weaker-than-or-equal-to) the truth of the original leaf clause. Thus, it is no harder, and probably much easier, to prove the clause

$$\overline{A(x)} \vee \overline{B(x)} \vee \overline{C(x)} \vee Q(x) \vee Q_1(x)$$

than it is to prove

$$\overline{A(x)} \vee Q_1(x).$$

Note also that proving the weaker clause above is also strictly easier than just trying to prove one of the four clauses along the path from root to leaf in the original tree (Figure 5.2). For example, the clause

$$\overline{B(x)} \vee Q_1(x)$$

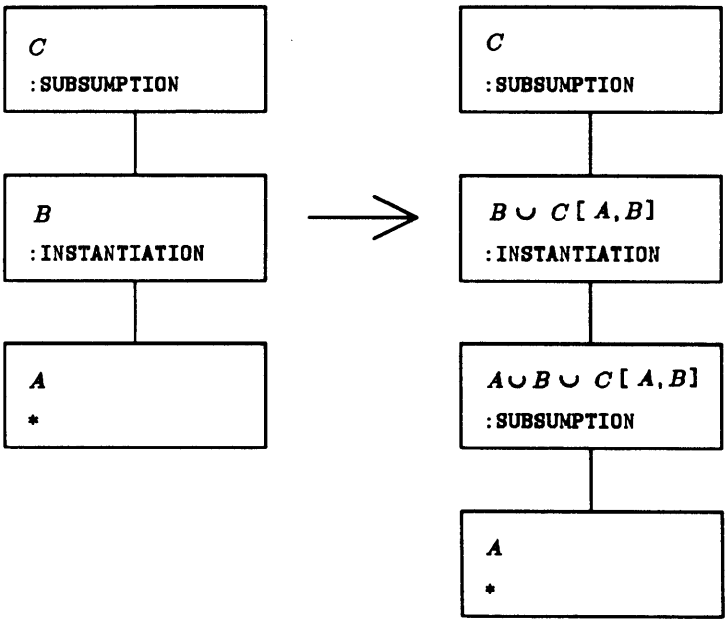
satisfies the weaker clause, but not any of the four pre-existing clauses.

I define the *weakened relative condition* of the leaf node n with respect to the proof-tree T, denoted $wrc(n, T)$, as the clause of the subsumption node immediately above n after applying the (extended, see below) unioning transformation on the path from the root to n.

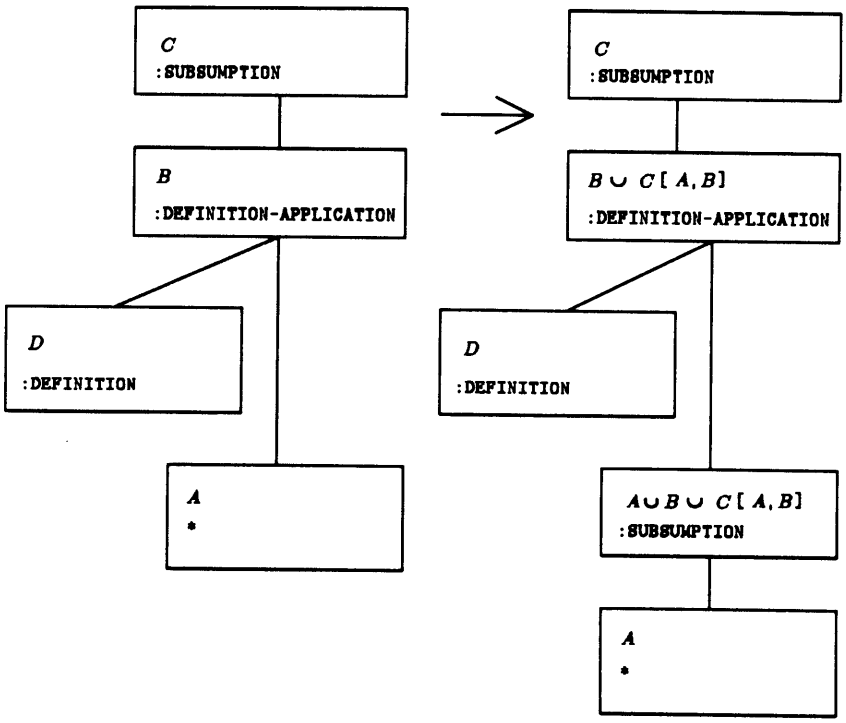
The example used only :RESOLUTION internal nodes; for the unioning transformation to apply to all proof trees, I must extend the unioning rules to handle the other types of internal proof nodes.

- :SUBSUMPTION *nodes:* A subsumption node whose child c is also a subsumption node may be replaced by a single subsumption node whose child is the child of c and whose clause is the same as the original node.

- :INSTANTIATION *nodes:* For this node type, we need some terminology. Define the *critical variables* of clause c_1 with respect to clause c_2 as all free variables appearing in c_1 that do not appear in c_2. Define the function $cterms(c, c_1, c_2)$ to be the set of terms in c containing some critical variable of c_1 with respect to c_2. Finally, for a clause c, define $c[c_1, c_2]$ to be $c - cterms(c, c_1, c_2)$. Given these definitions, the unioning rule for instantiation nodes is indicated in the following diagram:



- `:DEFINITION-APPLICATION` *nodes:* These nodes are handled by the following proof transformation, where the notation is as in the instantiation case:



Clearly, these rules suffice to propagate an initial (trivial) subsumption at the root all the way down to any chosen leaf, or group of leaves by simply repeating the process once for each group member. I call this process *parent-child clause unioning* (PCCU). If PCCU is applied to a given tree T with respect to a set of leaves S simultaneously, then the set of clauses obtained is denoted $wrc(S, T)$.

5.4 Proof-tree Restructuring

Suppose the previous example (Figure 5.2) had been structured instead as in Figure 5.4, the difference being that one pair of resolution steps is done in the opposite order. Whereas the first example was a linear sequence of resolutions, the root of the second has two nontrivial subtrees. Applying the extended unioning transformation to T_2 (denoting the `:FORWARD` node by n) we obtain

$wrc(n, T_2)$={`(Q ?X)`, `(NOT (C ?X))`, `(NOT (A ?X))`, `(Q1 ?X)`}

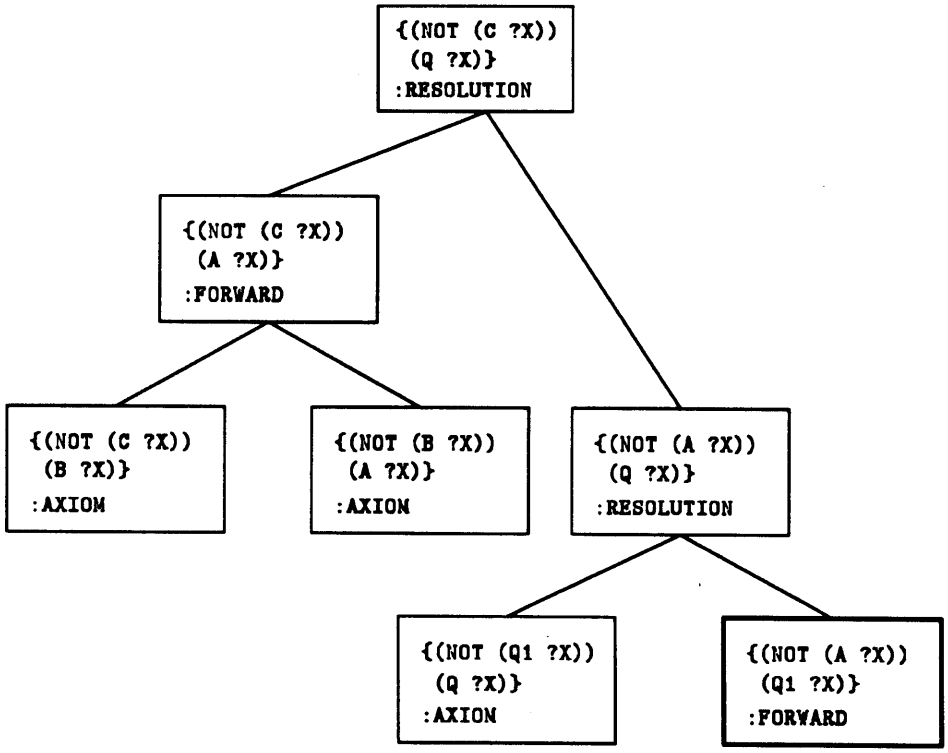


Figure 5.4: A different proof, T_2, of the same clause as in the previous example (Figure 5.2). Note that one pair of resolutions has been interchanged. The unioning transformation gives a smaller (fewer disjuncts) weakened relative condition.

which does not contain the term (NOT (B ?X)) that was present in $wrc(n, T_1)$.

The first important point to understand about this example is that *the two conditions are precisely equivalent assuming the conjunction of the other leaf nodes of the proof trees.* That is, if we keep the domain theory fixed, the two conditions are logically equivalent. This follows from first observing that $wrc(n, T_2)$ is a subset of (i.e. subsumes) $wrc(n, T_1)$, hence logically entails it; and second, observing that one of the other leaves (of both trees) is an axiom whose clause is {(NOT (B ?X)), (C ?X)} which can be resolved with $wrc(n, T_1)$ to get $wrc(n, T_2)$.

The second important point is that *the larger condition has more disjuncts, hence is probably more useful.* For example, it might be easier to prove things about the predicate B than directly about C. But the key advantage of extra disjuncts is that more disjuncts tends to increase the generality of the *operational* subset of the condition.

The explanation-based learning literature (Mitchell, Keller, & Kedar-Cabelli, 1986; DeJong & Mooney, 1986) defines a condition to be *operational* if it is stated in terms that are easily evaluated on a given data object; that is, if the condition is operationally useful for recognizing objects that satisfy it. Each problem-solver determines its own definition of "operational." I will define later what operationality means to my system, but part of that definition is the fact that some predicates must be effectively evaluable to be operational. Predicates that can't be evaluated are not operational.

With this in mind, it is easy to see that a condition with more disjuncts will usually have a more general operational subset than one with fewer disjuncts. In the example, if the predicate B is operational and C is not, then $wrc(n, T_1)$ has a more general (logically weaker) operational subset than does $wrc(n, T_2)$, because the former is satisfied when (B ?X) is false and the latter need not be.

IEBR uses PCCU to find an operational approximation to the weakest condition on a target that guarantees program correctness. A more general condition is satisfied by more sources, hence more optimizations will be allowed if the generalization process produces a more general answer. As the example illustrates, however, the process is sensitive to the form of the explanation tree. While some degree of this sensitivity will remain, I have implemented a way to *restructure proofs* that can improve the condition generated by reorganizing the proof tree. The restructuring is performed by iterating a terminating set of syntactic tree rewrite rules. Appendix D describes the al-

gorithm, RESTRUCTURE-PTREE(L, T), which returns a tree T_L that has been restructured by pushing the leaf nodes in the set L down deeper, hence with more intermediate clauses between them and the root.

Part II

The Optimization Algorithms

Chapter 6

Overview of the Algorithms

The system incorporates two optimization algorithms, IBR (for Invariant-Based Redistribution) and IEBR (for Invariant-and-Explanation-Based Redistribution), the latter being an augmentation of the former. Both are designed in accord with the generate-and-test paradigm, except that the testing is done in cascaded filtering stages, each successive filtering operation being more computationally expensive, but applied to less data than the previous. The two algorithms are depicted schematically in Figure 6.1.

The thick arrows represent conceptual flow of the collection of pairs remaining under consideration (not yet eliminated). Thin arrows show use of the different inputs: INV represents the optimization invariants; TEST-INPUTS are the test input data; PROGRAM is the program structure; and EXPLANATION is the proof. Note that both approaches have feedback paths (dotted) from the output to the generator. These arise because pairs are considered one at a time, and keeping or discarding a pair effects which pairs are subsequently generated.

The following three chapters give the main ideas of the two algorithms. Since the algorithms share the generation scheme, denoted by Generate in the diagram, and the syntactic pruning stage (*prune-syntactically*), these components are discussed together in Chapter 7. Note that "Generate" is not an explicit system function; rather, it is simply shorthand for the control structure implied by **Pair-Search** and its callees discussed in Chapter 7. The central search issues discussed there are the ways in which the system (1) avoids searching an exponential space of optimization candidates, and (2) orders the search, by estimated cost, to avoid the problem of insignificant

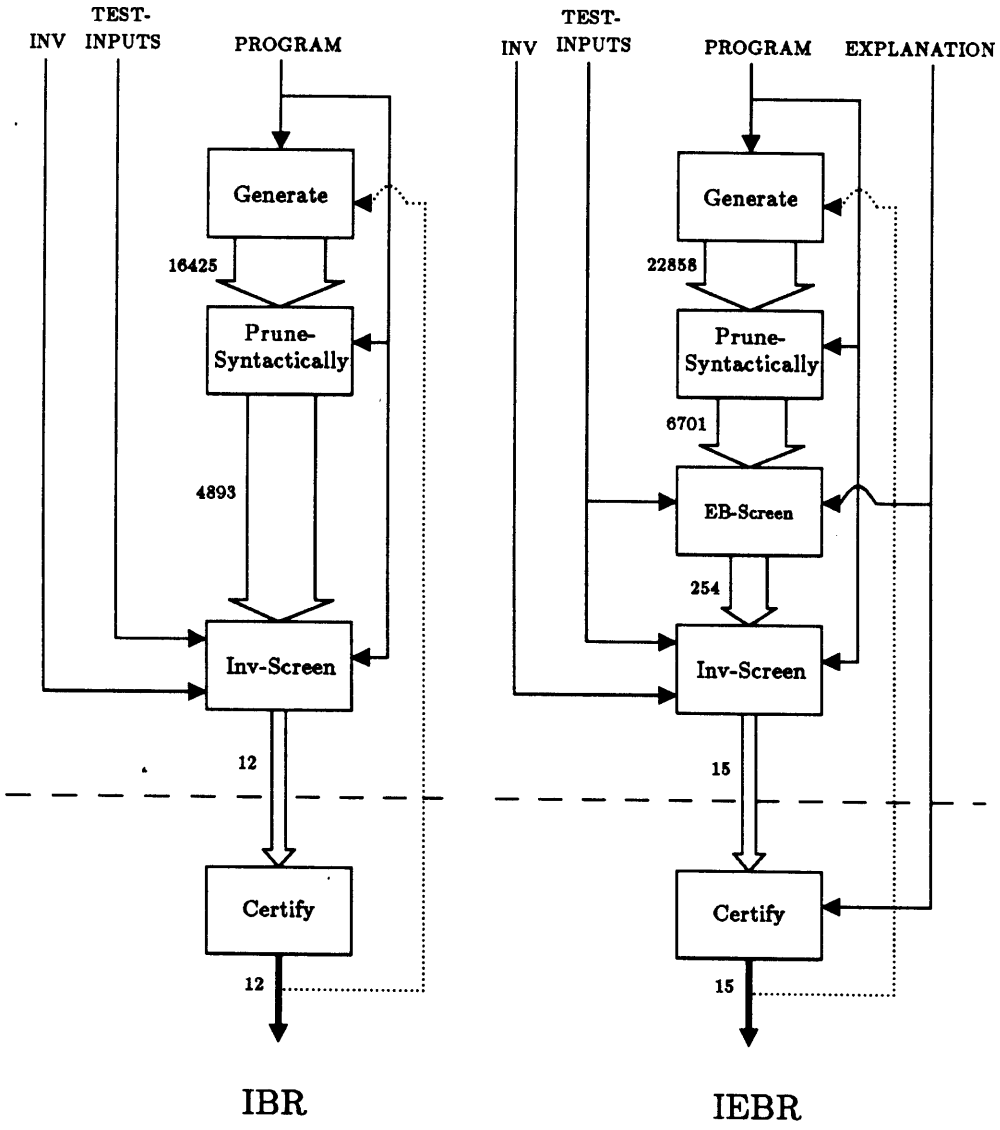


Figure 6.1: Conceptual diagram showing filtering stages in each of the two algorithms, IBR and IEBR. Arrow widths indicate relative numbers of candidate pairs not yet eliminated, though widths are not to scale. Numbers by the arrows are actual data (numbers of pairs) from the MERGE-SORT experiment. Note that there is a feedback path (dotted) from the output to the generator, because successful redistributions can cause structural elements to be eliminated from consideration.

optimizations interfering with more desirable ones.

Chapter 8 discusses the invariant-based screening technique embodied in the routine **Inv-Screen**. This is the heart of the IBR algorithm. Here, the central concern is the extra design information (optimization invariants) necessary to perform the screening.

Chapter 9 discusses the explanation-based screening technique of **EB-Screen**, which is unique to IEBR. This chapter motivates, defines, and proves properties of *target conditions*, the filter predicates used by **EB-Screen**. It also explains how these conditions are derived automatically from correctness explanations via a novel form of explanation-based generalization.

Throughout this part, I will use the following notation convention for naming system objects. Procedures whose names appear in **bold** face are system procedures with pseudo-code definitions given here. Functions whose names appear in *italic* face are system functions for which I have not given pseudo-code, but for which I have defined a specification, under the assumption that their implementations are routine. Names appearing in SMALL CAPITALS represent variables and formal parameters in pseudo-code[1]

Figure 6.2 shows a call graph indicating which procedures call which others. Caller is always above callee and connected by a tree branch. Note that *certify* is external to the system. Chapter 13 discusses the issues pertaining to certification.

Throughout, I assume implementations of various bookkeeping details and working datastructures to be used by the routines. Some of these datastructures are operated on by side effect as the search proceeds; I will indicate this in the pseudo-code. I will also assume a mechanism for retracting some of the modifying operations. Implementing these low-level operations efficiently is non-trivial but routine; therefore, I will not discuss them in detail, except where necessary for clarity.

[1] IBR and IEBR are exceptions to the small-capitals rule, since they denote the two optimization algorithms. No confusion can result, however.

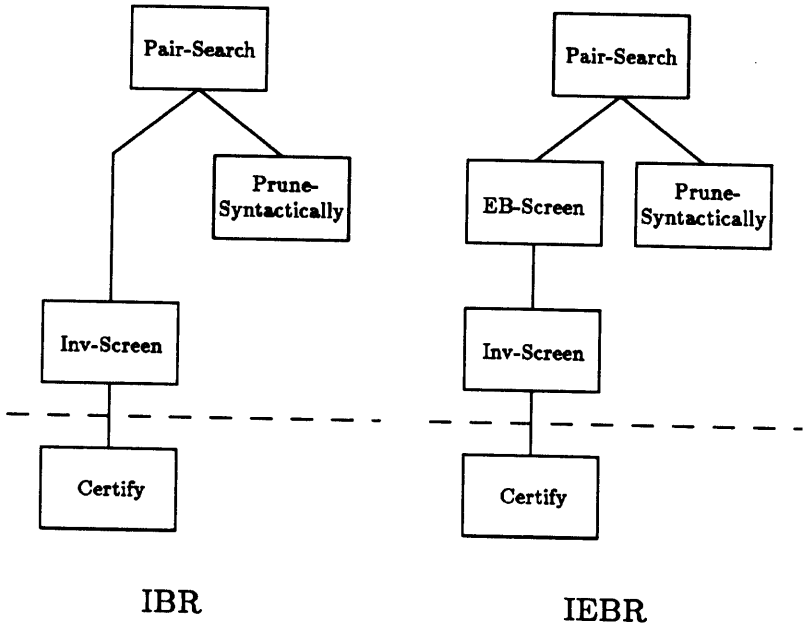


Figure 6.2: A call graph for the system, showing key subroutines.

Chapter 7

Candidate Generation

This chapter discusses the candidate generation process, including the syntactic pruning phase, used by both IBR and IEBR.

7.1 The Problem

Abstractly, the system's task is to find a set of redistributions that both maintains correctness and improves efficiency as much as possible. Because of several factors, such as the uncomputability of program efficiency, we cannot hope for an algorithm that finds the optimal set of redistributions. As do all program optimizers, we must settle merely for improving the program as much as possible within limitations imposed by practicality. In the worst case, we may even end up with a *less* efficient program.

A naive approach to search control would simply generate all possible sets of redistributions, asking the candidate screener about the correctness of each, and finally choosing the set that eliminates the most costly structure. The difficulty with this is the astronomical number of such sets. If we let t denote the number of targets in a given program and let s be the number of sources, then there are st source/target pairs. It doesn't make sense to propose a set of such pairs where two or more have the same target, so the total number of pair sets to consider is

$$\sum_{k=0}^{t} s^k \binom{t}{k}$$

which is much worse than exponential in t.[1] Since each port in a program, except program inputs and outputs, can be viewed as both a source and a target, the number of targets and the number of sources is roughly the same. Furthermore, the number of targets grows proportionately to the size of the input program, so even for programs of modest size, such a search would be impractical.

To avoid this complexity, the system considers pairs one at a time and never backtracks. The system's search control method generates source/target pairs, calling the candidate screener on each pair as it is generated, followed by a compile-time certifier[2] if desired. For each pair, the screener/certifier decides whether the redistribution maintains correctness of the current program. This reduces the worst case number of screening operations to the order of st. The incompleteness introduced by this approach appears not to be much of a hindrance in practice[3]; the time/power tradeoff is well worth it.

Note, however, that it is the responsibility of the search controller to provide the pairs in the right order, as once a redistribution is carried out it will not be retracted.[4] The screener/certifier makes no judgements regarding whether the redistribution actually improves the program's efficiency. Search ordering now becomes a central problem, because redistributions can mutually interfere; that is, it is possible for each of two redistributions to preserve correctness by itself, but for the combination of the two to violate correctness. Thus, the system must choose to do one first, thereby disallowing the other one; hence it is important to choose the one that saves the most time and space. I call this order-dependence among redistributions the *interference problem.* As an example of this, consider a program that copies the same input list twice before operating destructively on the result. Clearly, the system may eliminate either one of the copies via redistributions, but if it eliminates both copies the program would destroy the input list, violating its specification. If one of the copy operations is more expensive than the other, possibly because it is executed within a loop, then it is crucial for the

[1] This sum is greater than s^t, but less than $2^{st} = (2^s)^t$.

[2] A *compile-time certifier* is an external routine that checks whether a given pair is correctness preserving in all cases. See Chapter 13 for further discussion of this.

[3] "In practice" refers to the 26 examples run, of which a representative set is given in Appendix C. The examples are summarized in the Introduction.

[4] This is not completely true; see Section 7.2.

system to consider the more expensive one first. I will return to the issue of cost shortly.

Another aspect of the search ordering problem is that if the system is not careful, it can do a lot of work without eliminating any structure, i.e. without improving the program. This stems from the fact that *all* outputs of a box must be disconnected from all program targets before the box can be eliminated from the program. A great danger, therefore, is that the system can waste time finding and carrying out a number of redistributions that do not lead to actual improvement and that may in fact prevent other, more useful redistributions. I call this the *focus problem.*

In addition to search ordering, the system is faced with the fact that *st* pairs is still a lot of pairs to consider. As argued earlier, this is roughly quadratic in the program size. Moreover, it turns out that most of these—at least 50% and typically 80–90% in the examples run so far—are obviously illegal. One source of such pairs is *static type conflicts.* For example, any pair whose source is of type number and whose target is of type store is obviously bad. Illegal pairs also arise out of *causality conflicts*: any pair whose source is a causal successor of its target results in an illegal program, having a flow arc cycle. Fortunately, it is easy to rule out these obviously bad pairs.

7.2 The Solution Method

The system's approach to search control attempts to avoid all the problems mentioned in the previous section.

- As mentioned previously, the system avoids a combinatorial explosion by considering redistributions one at a time and never backtracking. Note that the system may retract a redistribution once it has been carried out—if the redistribution does not contribute to eliminating a box—but since it will never then reconsider adding the redistribution, this is not the type of backtracking that results in combinatorial search.

- The system avoids the focus problem by grouping pairs together that contribute to eliminating a particular box and considering one group at a time before going on to the next group. Successful members of a group corresponding to a box are only kept if there are enough to eliminate the box; that is, enough to completely disconnect its outputs.

- The system avoids the interference problem only partially. It considers boxes in order from most expensive to least expensive, hence if two pairs interfere, the one eliminating the more costly box is considered first. Of course, computing exact costs of programs is impossible, so the system uses a crude *qualitative cost estimation* technique. See Section 7.2.2 for the details of this cost function. This approach only partially solves the interference problem, both because the cost estimator is crude and because there exist programs where a more costly box may only be eliminated after a less costly box.

- The system uses simple syntactic criteria to eliminate obviously illegal pairs. See Section 7.2.3 for details.

7.2.1 Pseudo-code Description

This subsection gives a high-level pseudo-code description of the candidate generation algorithm. Conceptually, search control is implemented in the single top-level procedure, **Pair-Search**, which takes the program to be optimized and, via the input parameter SCREEN, a function to use for screening candidates. Currently, the only choices for this function are **Inv-Screen** and **EB-Screen**. These functions are discussed in subsequent chapters. **Pair-Search** is defined in terms of three other key subroutines, also defined below.

The code maintains a global working datastructure, referred to as the "program datastructure," corresponding to the program under optimization and operates on it by side effect. Auxiliary function names are written in italics; such functions have access to and may operate on the working datastructure. *cost*$(\cdot)$ is the cost function on program boxes defined in Section 7.2.2, and *prune-syntactically*$(\cdot,\cdot,\cdot)$ is the function that rules out obviously illegal sources for a given target (see Section 7.2.3). Names of procedures defined here are written in boldface; local variable names are written in small capitals. Recall that a recursion box is a box whose program type is the same as that of its parent program.

```
Procedure Pair-Search (PROGRAM, SCREEN) : set of source/target pairs
    Initialize global datastructures for PROGRAM
    RESULT-PAIRS ← {}
    BOXQ ← boxes(PROGRAM), sorted by decreasing cost(·)
    While BOXQ is not empty
        CURRENT ← head(BOXQ); remove CURRENT from BOXQ
        If physical?(CURRENT)
        Then BOX-RESULT ← Try-to-eliminate-box(CURRENT, SCREEN)
                If BOX-RESULT is not empty
                Then add BOX-RESULT to RESULT-PAIRS
                Else  If box-type(CURRENT) is not a primitive program
                      Then If CURRENT is a recursion box
                                Then add all non-recursion members of
                                     boxes(box-type(CURRENT)) into BOXQ
                                Else  add all boxes(box-type(CURRENT)) into BOXQ
                                Re-sort BOXQ by decreasing cost(·)
    Return RESULT-PAIRS.

Procedure Try-to-eliminate-box (BOX, SCREEN) : set of pairs
    RESULT-PAIRS ← {}
    OUTPUTS ← outputs(BOX)
    While OUTPUTS is not empty
        If there exists NEXT ∈ OUTPUTS such that
          (PAIRS ← Try-to-eliminate-box-output(NEXT, SCREEN))
          is not empty
        Then add PAIRS to RESULT-PAIRS
             and remove NEXT from OUTPUTS
        Else  retract all program datastructure changes caused by this call
              and return {}
    Return RESULT-PAIRS.
```

```
Procedure Try-to-eliminate-box-output (OUTPUT, SCREEN) : set of pairs
    RESULT-PAIRS ← {}
    TARGETS ← all targets connected to OUTPUT by a flow arc
    For each TARG in TARGETS
        NEW-SRC ← Find-substitute-source(TARG, box(OUTPUT), SCREEN)
        If fail?(NEW-SRC)
        Then retract all program datastructure changes caused by this call
                and return {}
        Else add the pair (NEW-SRC, TARG) to RESULT-PAIRS
                and carry out this redistribution on program datastructures
    Return RESULT-PAIRS.

Procedure Find-substitute-source (TARGET, BOX, SCREEN) : source or fail
    POSSIBLE-SOURCES ← prune-syntactically(all-sources(), TARGET, BOX)
    For each SRC in POSSIBLE-SOURCES
        If SCREEN(SRC, TARGET) then return SRC
    Return fail.
```

The rest of this section discusses the two key auxiliary functions, *cost*$(\cdot)$ and *prune-syntactically*$(\cdot,\cdot,\cdot)$.

7.2.2 Cost Estimation

Automatically determining the exact computational costs of a program is an impossible task in general. Even obtaining functional order measures is difficult (Le Matayer, 1988). For our purposes, however, such detailed calculations appear to be unnecessary for adequate optimization results. A crude qualitative measure suffices to determine which order to consider boxes for elimination. This subsection defines this measure.

The function *cost*$(\cdot)$ computes the *use-cost* of its input box. This is distinct from the *program-cost* of a given program. The program-cost of a program is a rough estimate of its run-time and (non-stack) space usage. Use-cost, on the other hand, includes a consideration of where the box is situated within the program. That is, it represents the cost to the top-level program of the box over all invocations. For example, if we have a box whose program type only requires constant time to execute but the the box is nested

within a loop with a linear number of iterations, the use-cost of the box is linear while the program-cost of its type is constant.

Costs (both kinds) take on values that are pairs *(time, space)* of non-negative integers where *time* represents the time cost and *space* represents the space cost. Increasing numbers indicate only an increase in nesting level within recursions; no account is taken of the relative sizes of the ranges of the recursions. A cost of 0 in either coordinate indicates no cost at all in that resource. A program constant box, for example, typically has cost $(0,0)$. A cost coordinate of 1 indicates a constant, but non-zero, cost; 2 indicates "linear" cost, 3 indicates "quadratic," and so forth. These are only intuitive guidelines, however.

For convenience, I define a cost incrementation operation, *cost1+*, as follows. Let *inc-nonzero* be the function that maps 0 to 0 and n to $n+1$ for $n > 0$. Then

$$\textit{cost1+}(a,b) \stackrel{\text{def}}{=} (\textit{inc-nonzero}(a), \textit{inc-nonzero}(b))$$

A box's cost is incremented when a box is used in an iteration, but iterating a zero-cost function does not increase its cost.

Program-cost

Computational primitive programs (see Section 3.4) must be assigned costs by the domain knowledge provider. Here are some examples:

- Constant programs (like `ONE`) are assigned the program-cost $(0,0)$.
- Accessing through a pointer (e.g., via `CAR`), which allocates no new memory, is assigned the cost $(1,0)$ indicating constant time and zero space.
- Allocating a memory cell is assigned $(1,1)$.
- Series operations that do not allocate memory are assigned cost $(2,0)$ because they typically represent iterations. Operations that do allocate memory cost $(2,2)$.

If a non-primitive program does not call itself, it will get a program-cost equal to the maximum of the program-costs of its boxes' types. Otherwise,

the recursive structure must be taken into account. More precisely, costs of non-primitive programs are determined as follows. First, program-costs are calculated recursively for all box-types within the program, avoiding loops caused by recursive programs, of course. Next, each box, b, in the program is assigned a context cost function, denoted $context_b(c)$, that maps a cost to a cost as follows. If b's condition does not conflict[5] with that of some physical recursion box[6] in the program, and if b is not itself a recursion box, then $context_b(c) = cost1+(c)$. In this case, I will say that *b lies in a recursive branch of the program.* Otherwise, $context_b(c) = c$. The program cost of the entire program is defined to be the maximum over all non-recursion boxes b in the program of $context_b(program\text{-}cost(box\text{-}type(b)))$.

As an example, the program-cost of `MY-REVERSE` (see Figure 1.2) is $(3, 3)$, because the box `$MY-REVERSE.CONC` has the same condition as the recursion box `$MY-REVERSE.REC` and also has program type `APPEND`, which in turn has a program-cost of $(2, 2)$. `APPEND` (see Figure 3.4) has a program-cost of $(2, 2)$ because the box `$APPEND.CPY` has program type `COPY-LIST` which has program-cost $(2, 2)$. Finally, `COPY-LIST` gets its program-cost from the series primitive `ZCOPY-CELL`.

Use-cost

The use-cost $cost(\cdot)$ of a box is intended to capture its contribution to the top-level program using it. It can be defined recursively using pathname notation as follows:

- $cost(\$P) = program\text{-}cost(P)$
- $cost(\$P.I1.I2.\ \ldots\ .Ik) = context_{\$P.I1}(cost(\$P1.I2.\ldots.Ik))$, where $P1 = box\text{-}type(\$P.I1)$.

Essentially, for every level of recursive nesting within which the box appears, its program-cost is incremented once (via $cost1+$). Note that $cost(\$P)$ is the program-cost of the program `P`.

Thus, for example, $cost(\$MY\text{-}REVERSE.CONC.CPY.ZCC)$ is $(3, 3)$, because it is nested within the `MY-REVERSE` recursion. $cost(\$MY\text{-}REVERSE.REC.REC)$ is $(3, 3)$

[5]Two conditions *conflict* if there is some test port appearing positively in one and negatively in the other. Conflicting conditions imply the two boxes are never both executed on the same invocation.

[6]Recall that a recursion box is one whose type is the same is its parent program.

as well, because even though it is nested inside a recursion, it is itself a recursion box.

Cost Comparison

Since costs are not numbers, they cannot be compared using the traditional $<$ relation. Instead, I have defined a total order on costs lexicographically as follows. I will overload the $<$ symbol to apply to this total order as well as numbers. $(a, b) < (a', b')$ if and only if either

- $a < a'$, or
- $a = a'$ and $b < b'$.

Thus, the costs are first compared on their time-cost coordinate; then, if the time costs are equal, their space-costs are compared. This scheme places primary emphasis on run-time and is based on the assumption that allocating space requires time proportional to the number of cells allocated. This model is appropriate, for example, on pointer architectures and on machines where allocating space implies initializing it as well, such as in the language C, as long as memory is allocated using the function `calloc`. The assumption implies that for all costs (a, b), $a \geq b$.

Shortcomings

This technique of cost estimation is rather crude; hence it fails to make some desirable distinctions. For example, the Fibonacci program (see Section C.1.2) gets a program-cost of $(3, 0)$, so will be treated the same as quadratic programs. It is, of course, exponential. This problem stems from the fact that it calls itself twice per invocation rather than once, a subtlety missed by the *cost*$(\cdot)$ function.

Another drawback is in the fact that *cost*$(\cdot)$ ignores relative lengths of iteration ranges. This can skew estimates considerably. For example, the **MERGE-SORT** program (see Section C.2.5) is assigned a program-cost of $(3, 3)$ ("quadratic") even though its actual complexity is $n \log n$, a complexity class without direct correspondence in the cost scheme. This could result in boxes of type **MERGE-SORT** being considered before more costly, quadratic boxes.

Even in the face of these and many other drawbacks, this approach to cost estimation and comparison has two key advantages: (1) it is simple to

compute, using a table to avoid repeated program-cost computations, and (2) it has performed well in guiding the search. A more accurate alternative, while desirable, must still be be easy to compute and must significantly outperform *cost*(·) as a search control function.

7.2.3 Syntactic Pruning

The function *prune-syntactically*(*source-list*, *target*, *box*) returns a sublist of *source-list* that contains only "syntactically plausible" alternative sources for *target*. This list is always finite. The *box* argument allows it to throw out any sources associated with *box*; this is always the box that the system is trying to eliminate. It would be counter-productive to consider sources within *box* if it is the one to be eliminated! The rest of this subsection explains the other criteria used to discard sources from *source-list*.

First, all sources appearing within any virtual structure (at any level) are discarded.

Next, all sources with static types incompatible with the static type of *target* are eliminated. This is tested by `STATIC-TYPE-CHECK?`, as discussed in Chapter 3.

Next, all sources that are causal successors of *target* are discarded, lest the corresponding redistribution create a flow arc cycle. This, of course, extends to all levels of the hierarchy.

Next, any sources (or targets[7]) that reside in branches of the program that were never executed for any test case are discarded. Clearly, no information is available, so screening such candidates would be a waste of time. To get these considered, the programmer must enter more test cases.

Finally, pruning is done based on *target*'s position in the recursion structure of the program. If *target* lies *within* a recursion box, then only sources from the body of the immediate parent call are kept. This reflects the fact that other possibilities are redundant, because they are accounted for by targets within the parent call. Also, sources within more than two levels of recursion of the same type are discarded, because we have to stop somewhere to get a finite list.

[7]Unreached targets are ruled out in **Try-to-eliminate-box-output**, causing the entire box output to fail to be eliminated, thus causing the entire box to fail to be eliminated. This is a refinement left out of the pseudocode for clarity.

7.3 Discussion

This approach to search control has worked well; on every example run so far (see Appendix C) the system completes in a reasonable[8] time, with adequate optimization results. A nice feature of this approach, though one I haven't used, is that basing the search on estimated cost provides a meaningful way of limiting the optimization search, in case programs take too long to optimize. That is, one can set a cost threshold that tells the system not to consider boxes whose cost estimates are at or below the threshold. Thus, we can avoid wasting a lot of time for relatively little return. I expect this feature will be crucial to applying the system to large programs.[9]

The system's approach is heuristic and can occasionally fail to examine boxes in the right order, because getting the most improvement requires eliminating a low-cost box before a high-cost box can be eliminated. Typically this will result in the system eliminating the low-cost box but not the high-cost box. This is a manifestation of the well known "local maximum" problem that afflicts all hill-climbing approaches. It is not yet clear how much of a problem this is, though it has not occurred in the 26 examples run so far. A possible solution (not as yet investigated) is to re-check a box when a target of one of its outputs is rendered virtual by an optimization that occurs after the box itself has been checked. This could succeed because that target may have been the only one for which a substitute source could not be found.

The most serious lack of generality in the implementation is that complicated recursive structures (such as coroutines) where there is indirect recursion cause the system to go into an infinite loop. This is because it only checks for *self*-recursion in considering whether to add a box to the processing queue. Also, the cost estimation functions check only for self-recursion as well. I expect that extending the system to handle more general recursion will not be too difficult.

Another small problem with the approach is the fact that it only examines

[8]By "reasonable" I mean in a time consistent with a low-order polynomial function. Keep in mind that actual times are inflated by the fact that the implementation has not been engineered for maximum efficiency.

[9]All examples were run with this cost threshold set to (0,0). I left the threshold test out of the pseudo-code description of **Pair-Search** for simplicity. It would appear as an additional condition to be checked before adding a box to the BOXQ.

boxes and sources down to a fixed level of recursive nesting. One can always construct examples requiring examination of deeper boxes; on the other hand, one can't allow the system to descend forever into a recursion. This problem has not yet arisen on my examples.

Chapter 8

Candidate Screening I: Inv-Screen

The **Inv-Screen** candidate screening procedure takes in a source/target pair for the program to be optimized, together with appropriately initialized datastructures constructed by **Pair-Search**, and decides whether the pair defines a "believable" redistribution. If the system is to be run with a compile-time certifier[1], then **Inv-Screen** calls it after first screening the candidate for consistency with the test inputs.

8.1 Pseudo-code Description

The following is a somewhat idealized description of the **Inv-Screen** screening procedure. In reality, some of the program datastructure alterations done here and in the caller (**Pair-Search**, et al.) would be shared instead of done twice and retracted in between. A compile-time certifier *certify*$(\cdot)$ would also take advantage of internals of this and the other routines for efficiency. These details are suppressed here.

To use **Inv-Screen**, the programmer must provide a comprehensive collection of effective *optimization invariants*. In the current system, each of these is a clause expressing some constraint between the inputs and outputs of the program. As stated earlier, each is *effective* if it can be evaluated

[1] See Chapter 13.

on ground instances by the system's clause evaluator. Each may be *relative* in that it may be expressed in terms of special pathname symbols that designate the outputs of the unoptimized program (these are the program output pathnames extended by the keyword `ORIGINAL`). Note that for the purposes of **Inv-Screen** alone, there is no reason for these invariants to be in clause form; they could simply be black-box Lisp subroutines provided by the programmer. I formulate them here as evaluable clauses for uniformity of mechanism with **EB-Screen** (see Chapter 9).

The programmer must also provide only a set of representative test *inputs*, rather than test *vectors*, which would include the outputs as well. The whole point of using optimization invariants is to allow the optimizer the freedom to change the exact output values for a given input if it makes the program more efficient.

Procedure **Inv-Screen** (SRC-TRG-PAIR) : Boolean
    [Assumes program datastructures are initialized appropriately
    for the current program to be optimized.]
    Carry out redistribution corresponding to SRC-TRG-PAIR
    For each test input TEST, of the program to be optimized:
        Evaluate the altered program on TEST producing new outputs
        For each optimization invariant, INV, of current program
            Evaluate INV on inputs and new and original outputs for TEST
            If INV is not satisfied
            Then retract all program changes made by this call
                and return FALSE
    Retract all program changes made by this call
    If a compile-time certifier, *certify*, is available
    Then return *certify*(SRC-TRG-PAIR)
    Else return TRUE.

8.2 Discussion

Inv-Screen is more powerful than **EB-Screen**, but suffers from several drawbacks of varying degrees of import. By far the most important difficulty is in the number of test input evaluations required. The other difficulties are relatively minor.

8.2.1 Optimization Power

Inv-Screen is conceptually simple, yet surprisingly powerful. The basic observation regarding its power is that it will not rule out any correct redistribution (i.e., it gives no false negatives), assuming the input invariants accurately reflect the desired postconditions. It can give false positives due to unfortunate test coincidences; this is a consequence of substituting test input reasoning for general theorem proving in the screening phase. If an omniscient *certify*$(\cdot)$ is available, however, then **Inv-Screen** will answer every query correctly, but generally much faster than simply calling *certify*$(\cdot)$, without screening, every time. Calling *certify*$(\cdot)$ on every pair would be drastically slower, because it would involve calling something like a theorem prover on every pair in the program. Since there are potentially hundreds, thousands, or more pairs in a program such an approach would be impractical.

The only redistributions missed will be those ruled out by inconsistency with an overly-conservative optimization invariant, and those which *certify*$(\cdot)$ is unable to prove correct. Missing a redistribution because of a too-conservative invariant can be viewed as a defect in the input rather than in the algorithm itself.

8.2.2 Test Input Evaluations

Most important of **Inv-Screen**'s shortcomings is that it requires altering the program and re-evaluating all test inputs once per candidate pair. In particular, many of these runs will be of the almost completely *unoptimized* program. Mitigating against this, however, is the hope that one can find a small set of small tests that are sufficiently representative for optimization, yet run the optimized ("production") program frequently on much larger cases. For example, `FIB`, in its unoptimized form, is exponential, so can only be run on small tests; but the optimized form is linear[2] and can be run practically for much larger inputs. Moreover, the single test input `4` suffices for optimization. Obviously, **Inv-Screen** is impractical to use on programs for which the only representative test sets are complex. This property should not be confused with complexity of the program itself: large and complex programs can have small representative tests. Moreover, even if no such set exists, it may still be possible to find small test sets that execute only a

[2]that is, the number of recursive calls is proportional to the input value

portion of the program, allowing the system to optimize the executed portion and ignore the unexecuted. I believe it is an open empirical question as to whether "real-world" programs have small enough representative test sets for **Inv-Screen** to be practical.

8.2.3 Certification Difficulties

The next most important shortcoming of **Inv-Screen** is its tendency to find conjectured optimizations that are difficult to prove correct (or incorrect). **Inv-Screen** passes these because it passes all redistributions whose resulting program outputs are consistent with the optimization invariants on all test inputs, regardless of the underlying correctness reasoning. This property is a double-edged sword in that on the one hand, one is impressed by clever optimizations; on the other hand, such optimizations are dangerous unless they can be certified.[3] Both intuitively and practically, redistributions that represent small perturbations on the correctness argument of the program are easier to prove correct than clever ones that are based on deep domain reasoning or complicated reasoning about the dynamic behavior of the program. The reason is that with a perturbation, "most" of the correctness proof of the program is simply carried over to the new program. Thus, it would be best if the certifier could be called only on the perturbation-type of optimization, as the other type is likely to consume lots of resources. Unfortunately, **Inv-Screen** has no way of distinguishing the two types and simply reports all of them. By contrast, **EB-Screen** usually does find only perturbation-type optimizations (see Chapter 9).

A compile-time certification scheme is more vulnerable to this difficulty than is a run-time scheme. Obviously, if the optimization is correct, then the run-time scheme (see Chapter 13) will never find a bug in it and thereby pays no cost due to it. If there is a bug, however, it will be found the same as any other optimization-induced bug.

8.2.4 Program Checking Difficulty

Another, less important, shortcoming of **Inv-Screen** is its reliance on *effective* optimization invariants. The issue of program checking—coding a

[3]See Chapter 13 for a complete discussion of certification schemes.

routine to check the correctness of the outputs of another program—is one that has received quite a bit of attention recently (Blum & Kannan, 1989). I have extended this notion to include *relative* program checking—where the checker is allowed access to the outputs of the original (unoptimized) program. It is both theoretically and practically easier to find relative checkers than non-relative ones. I will sometimes use the term "program checker" to mean a complete set of effective optimization invariants.

It is difficult enough to formalize specifications of complex programs; formalizing *effective* ones figures to be harder. In fact, there are computational problems that have easy programming solutions, but for which it is *impossible* to code an effective program checker for the problem. For example, the "compiler problem" of constructing a low-level machine code program equivalent to a given high-level source program has become routine—people write compilers all the time—yet computation theory shows that it is impossible to write a program that can tell for every (low-level program, high-level program) pair whether the high-level program computes the same function as the low-level one. This is true even given access to a known correct compiler for the same language; hence, even a relative program checker is impossible for the compiler problem. On the other hand, there are many examples of problems for which it is easy to find effective invariants. (All of the examples in Appendix C—except **APPEND**, which is simple enough to do absolutely—were done using relative invariants.) Moreover, it is usually much easier to find relative invariants than non-relative ones (i.e. postconditions); of course, it can reduce optimization power somewhat, by not allowing as much freedom as possible.

The uncheckability of some problems has not been an obstacle in practice. Appendix E, however, explores the relationships between computability, checkability, and relative checkability from the standpoint of computation theory.

Chapter 9

Candidate Screening II: EB-Screen

The **EB-Screen** candidate screening procedure satisfies the same input/output specification as **Inv-Screen**: it takes in a source/target pair for the program to be optimized, together with appropriately initialized datastructures constructed by **Pair-Search**, and returns whether or not the pair preserves correctness on all test cases, or whether the pair is correct in general if a certifier is used. If the system is to be run with a compile-time certifier, then **EB-Screen** calls it[1] after first screening the candidate for consistency with the test-cases. The key difference between **Inv-Screen** and **EB-Screen** is in how they perform this task.

9.1 Pseudo-code Description

As in the case of the pseudo-code for **Inv-Screen**, the following pseudo-code for **EB-Screen** is somewhat idealized and has many details suppressed for clarity. Again, the actual implementation is optimized so that expensive operations like program surgery are shared rather than recomputed.

[1] Actually, **EB-Screen** calls **Inv-Screen**, which calls it.

Procedure **EB-Screen**(SRC-TRG-PAIR) : Boolean
[Assumes program datastructures are initialized appropriately
for the current program to be optimized.]
TRG-CND ← *find-target-condition*(*target*(SRC-TRG-PAIR))
If TRG-CND evaluates to true in every test case trace in which
source and target are both initialized
Then return **Inv-Screen**(SRC-TRG-PAIR)
Else return false.

Basically, **EB-Screen** computes an operational (see below) CNF predicate of one argument, called a target condition, and evaluates it with pathname terms replaced by actual data values from each test case in succession, and with the argument replaced by the value of the source in the test case. If the target condition evaluates to true in every test case for which the source and target are both initialized, i.e. both appear in a part of the program that was actually executed on the test case, then **EB-Screen** calls **Inv-Screen** to verify that this optimization would not cause previous redistributions (those carried out in earlier iterations of **Pair-Search**) to become incorrect. Key questions, then, are

- What are target conditions and how are they computed?
- Why does **EB-Screen** need to call **Inv-Screen**?
- What is the relationship between target condition screening and certification? In particular, can such screening help restrict to routine optimizations only?

The following sections answer these questions.

9.2 Target Condition Theory I: Single Redistributions

A target condition is an effective predicate intended to capture what is required of the object connected to a target in order for the overall program to be correct. Ideally, the program will remain correct after connecting the target (via extended flow) to any object satisfying the target condition.

The system represents target conditions in conjunctive normal form—i.e. as a collection of clauses (see Chapter 4). Desirable attributes of a target condition are that it guarantee correctness, that it be effective, and that evaluating it not require re-execution of the tests. I will term the conjunction of the latter two properties *operationality* of the target condition, a term introduced by Mitchell, et al in the explanation-based learning literature (Mitchell, Keller, & Kedar-Cabelli, 1986).

There are two obvious candidates for target condition for a given target. First, one could set it equal to the top-level specification of the entire program, in effect saying that "any object may be connected to the target as long as the overall program remains correct." In effect, **Inv-Screen** employs this approach by using the optimization invariants as the target condition for *every* target in the program. The obvious difficulty is that we cannot evaluate the correctness of such a condition on given data unless we re-execute the altered program to actually compute the new outputs. Thus, this approach is not operational, because it requires re-executing the test-cases for every source/target pair.

The second obvious candidate for target condition is the predicate "equal to the value of the old source." That is, the target condition is true of any object if and only if the object is identical to the value of the source connected to the target in the original (unmodified) program. This is clearly operational; since programs may not have flow cycles, the value of the original source cannot depend on the value of the target and so would have the same value even after the modification.

Adopting "equal to the old source value" as the target condition for each target in the program, with "source" depending on choice of target, is viable and captures precisely all of the identical-value redistributions (see Section 1.4). Moreover, no proof inputs are required, so such an approach is at least as easy to use as IBR. To capture the other classes of redistribution phenomena, however, we must find more general (weaker) operational target conditions.

In the sections to follow, it will be useful to keep in mind that every target condition computed is entailed by "equality to the original source," and in fact will usually be strictly more general. More precisely, every clause of every target condition computed by the system will contain a term (disjunct) of the form (= *old-source ?target*), so this equality logically satisfies any target condition.

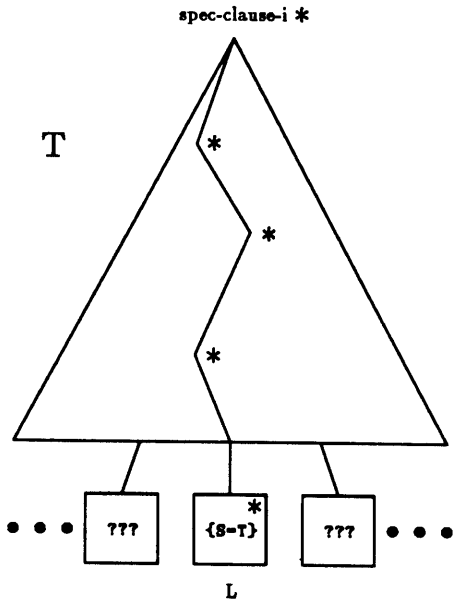


Figure 9.1: A "typical" proof tree, T, of one clause of a program specification. The leaf node labelled L has clause {S=T} and is the :PROGRAM-STRUCTURE node of subtype :FLOW corresponding to the flow arc in the original program. The nodes on the path from the root (a spec clause) to this leaf are marked with asterisks, indicating they are to participate in parent-child clause unioning.

9.2.1 Computing Target Conditions from Proofs

A redistribution (single flow rerouting) logically involves two steps: retraction of a flow equality and assertion of a new flow equality. I do not distinguish here between standard and extended flows, because they both amount to logical assertions of equality between a pair of values. I will motivate the target condition computation by showing how the program's correctness proof changes across a single redistribution operation, and by proving a theorem that shows that satisfaction of the target condition by the new source guarantees that the program remains correct.

Consider the proof structure of the original program. A schematic view of one proof tree is shown in Figure 9.1. This shows the connection of program structure to function: the flow arc (a structural entity) is captured in proofs through inclusion of a :PROGRAM-STRUCTURE node of subtype :FLOW (leaf node

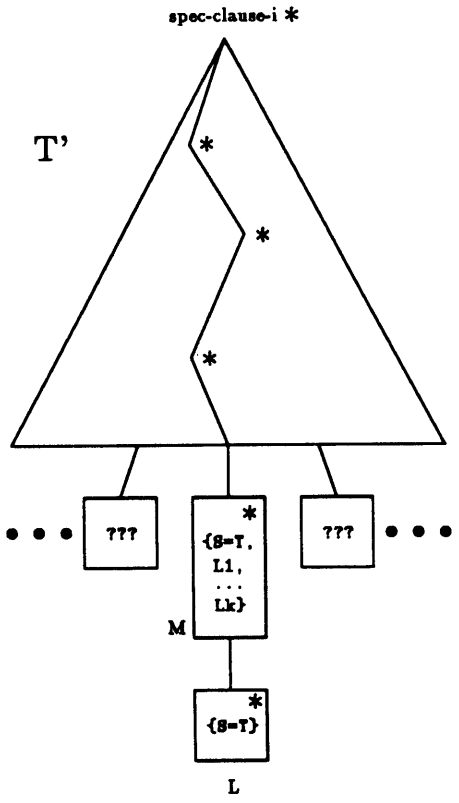


Figure 9.2: The tree T after tree unioning is applied. Note that all clauses along the marked path are altered (enlarged somewhat) and an extra subsumption node M (whose clause is $wrc(L, T)$) is installed just above L.

L in the figure). The clause of such a node is (as described in Chapter 5) basically an equality of the flow's source to its target, possibly including conditionalizing disjuncts.

For now, assume the program has a one-clause specification and that the flow equality is used only once in the proof. With multi-clause specifications, all proof trees are handled the same way, with the resulting target condition being simply the conjunction of the single-tree results. Moreover, each occurrence within the tree of the flow arc leaf node is handled separately as well, the results also being combined by conjunction.

To compute the target condition for the target t, the system applies

parent-child clause unioning (cf. Chapter 5) to the proof tree T along the path from the root to the flow arc leaf L, obtaining $wrc(L, T)$ as the clause of the (new) subsumption node M immediately above L. Thus, we obtain the altered proof structure shown in Figure 9.2. As explained in Chapter 5, the tree T' must remain a valid proof tree.

Now, retracting L and asserting a new flow equality in its place requires "patching" the correctness proof. We can do this by proving some clause subset of M's clause. This is depicted schematically in Figure 9.3. The identical-value special case of this would be to reprove that the old source is equal to the target by first proving that the new source is equal to the old source and then using the new flow equality of new source to target.

It should be clear that the truth of any subclause of M's clause guarantees the correctness of the overall program, because one can quickly and automatically construct a correctness proof by attaching a "patch" subproof in place of the original node L. I define the *target condition* for target t, $\mathrm{tc}_t(?target)$, to be the maximal operational subclause of M's clause, with every pathname term for t replaced by the variable *?target*.[2]

To find this subclause, the system simply discards any terms containing references to causal successors of the target; that is, any pathnames reachable from the target via a flow path. Since the clauses are disjunctions, we can throw out any disjuncts and still have a clause whose truth implies the truth of the original.

Finally, so that it may be viewed as a predicate of one input, the resulting condition is λ-abstracted by replacing all explicit references to the target's pathname by the variable *?target*. Note that there must be at least one, because the equality term has one. As stated earlier, this process is applied in parallel to all occurrences of the flow arc leaf in all proof trees supporting the specification and the results are logically conjoined—each must be proved true for the entire target condition to be proved.

This discussion essentially proves

> **Theorem 1:** Given a proved correct program, and given a source/target pair (s, t) for the program, if we change the program by retracting the flow arc ending at t and asserting a (possibly extended) flow arc corresponding to (s, t), not changing any of the

[2]Note that tc really depends on which particular proof is used. Since the system only allows one proof for a given program, I will leave this parameter implicit.

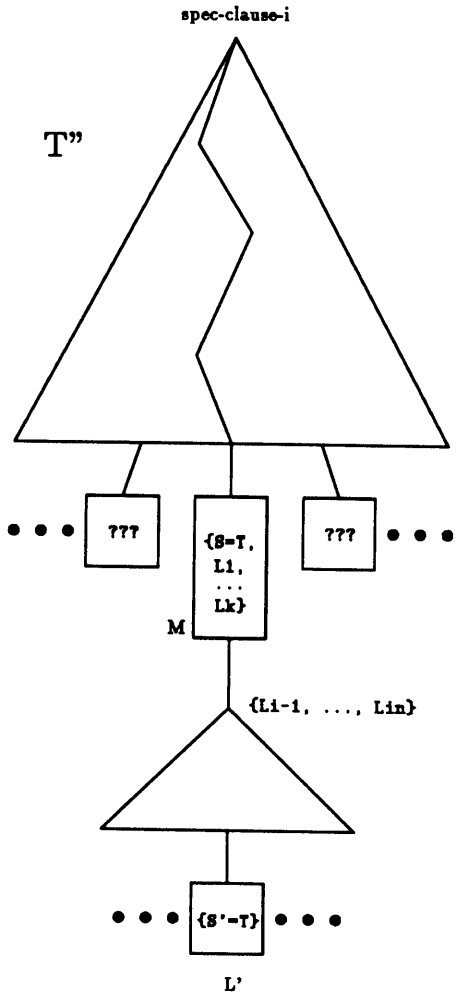


Figure 9.3: The proof after patching using the new flow equality (L'). Note that the clause at the root of the new subproof is constrained only to be a subset of the clause of M. Hence, it may or may not contain the equality $S = T$. Note that the subproof may contain any number of copies of the leaf L'.

other :PROGRAM-STRUCTURE nodes used in the proof, then the program still satisfies the specification proved by the proof if $\mathrm{tc}_t(s)$ is true for every program input.

Note that to calculate weakened relative conditions, the essential tool in calculating target conditions, the system need only walk the tree down the path from root to leaf collecting all the literals allowed by the tree unioning rules: all those not containing critical free variables as defined in Chapter 5. This saves the large amount of datastructure copying and manipulation that would be implied by actually carrying out the local tree manipulations implied in the definition of the unioning transformations. This is an important insight for efficient implementation.

9.2.2 Extensions to Handle Hierarchical Structure

The technique given above for computing a target condition applies only to the case when the target is a port in the top-level program directly. Ports within box implementations (at any level) will not appear in the proof of the top-level program, because only the boxes' instantiated specifications may be used in proving the top-level specification. Lower-level ports will appear in the correctness proofs of the specifications of the boxes' program types. This situation appears schematically in Figure 9.4. To compute the target condition for such a nested target requires extending the target condition computation to handle hierarchically organized proofs.

Conceptually, we can solve the problem by simply renaming the pathnames in the lower tree to their corresponding instances under the instantiation mapping[3] implicit in the use of the :PROGRAM-STRUCTURE node L_u, then attach the renamed tree in place of L_u, and use the approach given earlier on the constructed proof-tree. The result of this is shown in Figure 9.5.

As before, multiple occurrences of a leaf within a tree or multiple occurrences of different spec clauses are handled separately and simply combined by logical conjunction at the end. Some subsequent cleanup is performed as well, such as removing duplicate clauses.

[3]Every use of a :PROGRAM-STRUCTURE node of subtype :BOX requires instantiating the corresponding specification clause by renaming all pathnames in the specification of the box type to their global versions referred to the top-level program.

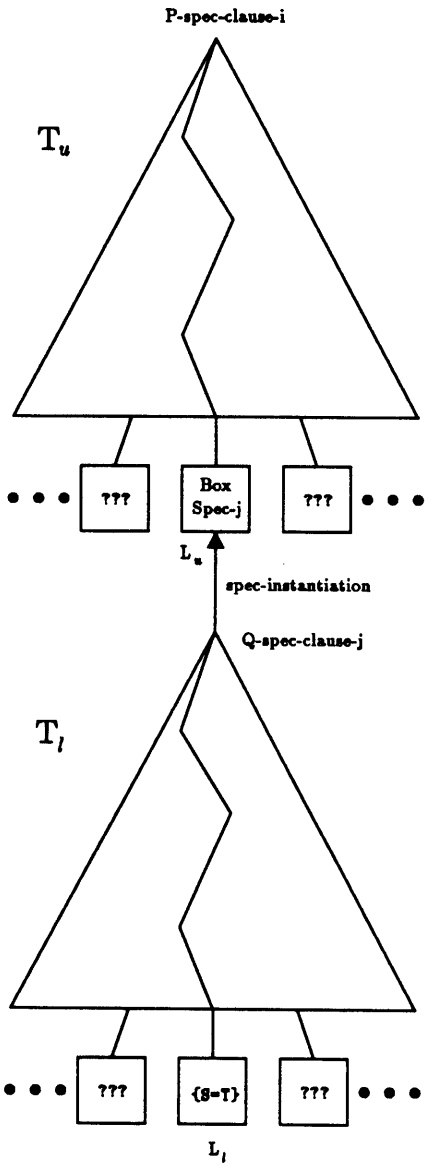


Figure 9.4: A schematic diagram of the hierarchical case. The upper tree T_u is a proof tree for a clause of the specification of top-level program P. It contains a leaf node L_u of type :PROGRAM-STRUCTURE and subtype :BOX which instantiates the spec of a box's type program Q. The lower tree is a proof of the corresponding clause of Q's specification, containing a flow arc leaf node labelled L_l.

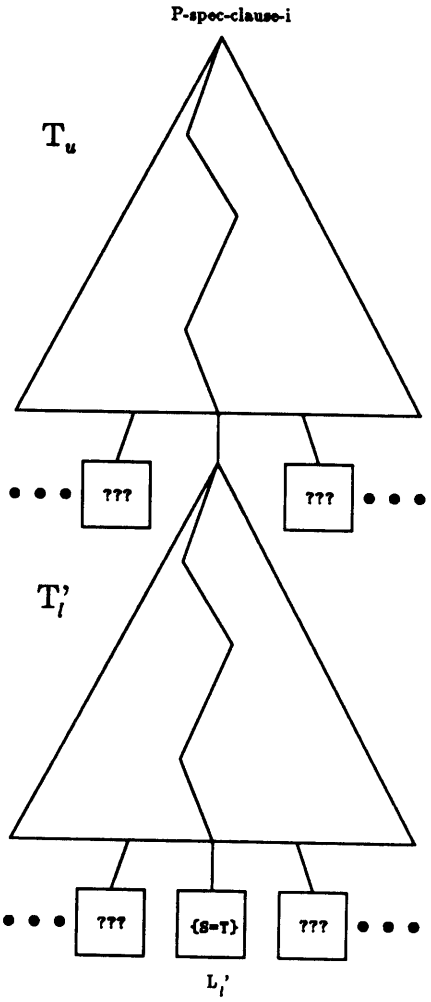


Figure 9.5: Conceptual solution to hierarchy problem. The instantiation mapping from the previous figure is used to map all the pathnames in the old T_l to the correct extended pathnames in T_l' and the renamed tree is installed in place of the :PROGRAM-STRUCTURE node L_u. The non-hierarchical target condition computation then produces the right answer. The actual implementation, however, does not construct the renamed tree and in fact performs no tree surgery whatsoever.

Implementing this approach naively would be inefficient, because new tree structure would have to be constructed for every target condition computation. Fortunately, it is again possible to avoid all tree renaming and surgery operations. By walking down the path starting from the overall root the implementation simply collects the additional (operational only) terms (disjuncts) along the way in accord with the tree unioning rules. Every time a `:PROGRAM-STRUCTURE` node of subtype `:BOX` is encountered, the system keeps track of the pathname mapping and uses it to translate pathnames in terms collected from lower trees to their correct global pathnames. The system gains even more efficiency from the observation that the term collection process need only be done once for any root-to-leaf path within a single tree, the results being stored in a table. Only the renaming must be done for each different usage.

9.2.3 Evaluating Target Conditions

Target conditions tend to be rather large, simply because they are derived from what tend to be large proofs and they have a fair bit of redundancy. It turns out that many of the terms in the clauses of a typical target condition do not refer to the target variable nor any free variables. These clearly need only be evaluated once per test case, rather than once per source/target pair. I have found that applying partial evaluation and caching to target conditions prior to performing search speeds things up immensely.

9.2.4 Quality of Proof Structure

As implied in Chapter 5, the form of the proof trees has a significant effect on the generality of the target conditions obtained from the target condition procedure.

The basic principle of program proof structuring is to *prove weak operational facts about a given target first and use only those facts in proving subsequent facts.* A fact is "operational" with respect to a target if it is stated only using terms having no pathnames in them naming causal successors of the target. The principle states that one should prove the weakest of such facts about a target that still suffice to prove the desired clause before going on with the proof.

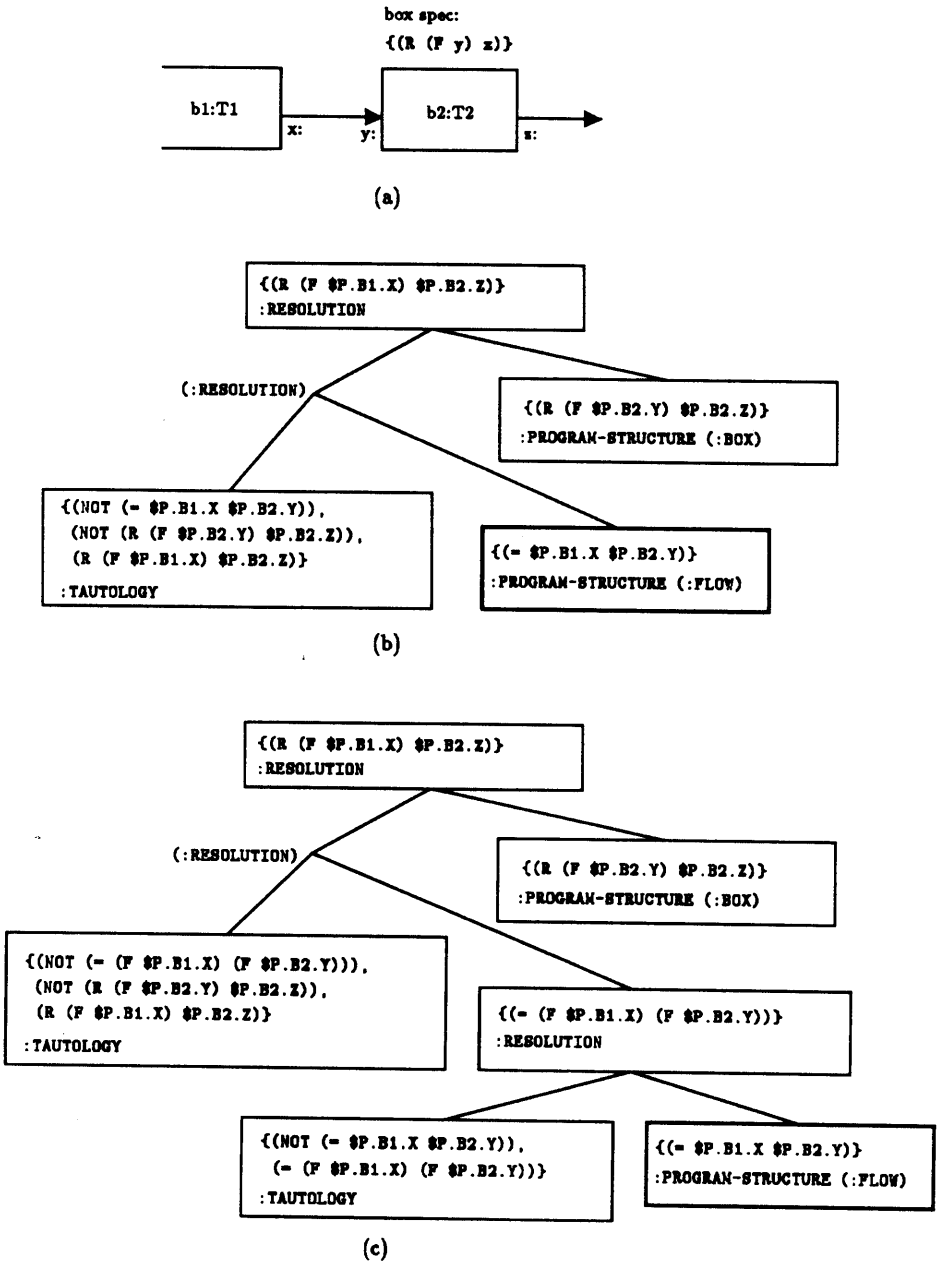


Figure 9.6: (a) A simple program fragment where box b2 enforces the constraint {(R (F y) z)}. We prove the fact that {(R (F $P.B1.X) $P.B2.Z)}. (b) shows a poorly structured proof, and (c) shows a well-structured proof.

Consider, for example, the simple program fragment shown in Figure 9.6 (a). The target condition derived for `$P.B2.Y` from the proof in (b) contains only one term (the original flow equality) because the only other term above the flow equality leaf node is non-operational with respect to the target, since it mentions the causal successor `$P.B2.Z`. On the other hand, the proof may be reformulated as in (c) so that the weaker (but operational) term `(= (F $P.B1.X) (F $P.B2.Y))` appears in the target condition. That this is more desirable can be seen if we instantiate `F` to the predicate *is-an-even-number?*. Clearly, a target condition requiring only an even number is much more general than one requiring equality to `$P.B1.X`.

Note that since the weakened conditions obtained from different occurrences of the same leaf node are conjoined (i.e. they must all be satisfied for the entire condition to be satisfied) the principle must be obeyed for every use of a flow arc proof leaf.

I have found that this principle is easy to follow, because it is compatible with a natural style of program verification: start from the input preconditions, successively propagating minimal facts forward through flows and boxes, until we prove the output postconditions. This propagation from inputs toward outputs tends to lead one to formulate intermediate lemmas in more operational terms.

9.2.5 Proof Restructuring

Proof restructuring (as discussed in Chapter 5) can compensate somewhat for proofs not structured in accord with the basic principle above. In particular, it is good at getting intermediate lemmas "moved above" chosen leaf nodes so that terms from the intermediate lemmas propagate down to the weakened relative condition. Two problems restrict its use, however, so the prover must still produce well-structured proofs. Proof restructuring removes some, but not all, of the burden.

The first restriction is that full proof restructuring, that which uses the resolution rule that can duplicate subtrees, can blow up the tree exponentially, hence may not be used indiscriminately. This restriction implies we cannot use optimal restructuring in every case.

The second restriction is that for target conditions to be correctness preserving it must be possible in principle to construct the unioned proof tree using them all simultaneously. This implies that we cannot restructure the

proof tree differently for each target (as we might be tempted to) because otherwise we might not be able to assemble the resulting target conditions into a proof. This restriction implies that the restructuring must occur only once, before optimization, and must treat all flow arc leaves "equally," i.e. it cannot be biased toward improving one target condition at the expense of others.

The system's approach to restructuring is to mark path from the root to a leaf of type `:PROGRAM-STRUCTURE`, allowing all and only those restructuring moves that involve an unmarked subtree. This allows proofs to use lemmas from a lemma library (whose proofs contain no `:PROGRAM-STRUCTURE` nodes) and still get the same benefit as if the proofs had been instantiated by hand directly into the proof. This effectively increases generalization power by using domain knowledge, in the form of intermediate proof structure from the lemma library, to improve the proof structure automatically. I have run examples where optimizations are found when restructuring is used, but missed when it is turned off.

I have not restricted the exponential restructuring rules, because they have not caused any blowup. It may, in a practical implementation, be necessary to implement such a restriction, however.

9.3 Target Condition Theory II: Multiple Redistributions

In principle, to calculate accurate (i.e., sound) target conditions for targets after the first redistribution has been carried out one must produce a patch proof, install it in the tree, and calculate it as described above. This has the major drawback of requiring that a theorem stating that source satisfies target be proved before the second and subsequent redistributions can be found.

Obviously, **Inv-Screen** alone avoids the need for patch proofs, but requires too many test-case evaluations. Fortunately, there is a way to avoid the patch proof requirement, yet still avoid almost all[4] test-case evaluations. To see how, I must develop a bit more theory.

Basically, we wish to see how the program's correctness proof can in

[4] *All* in some variants of **EB-Screen**

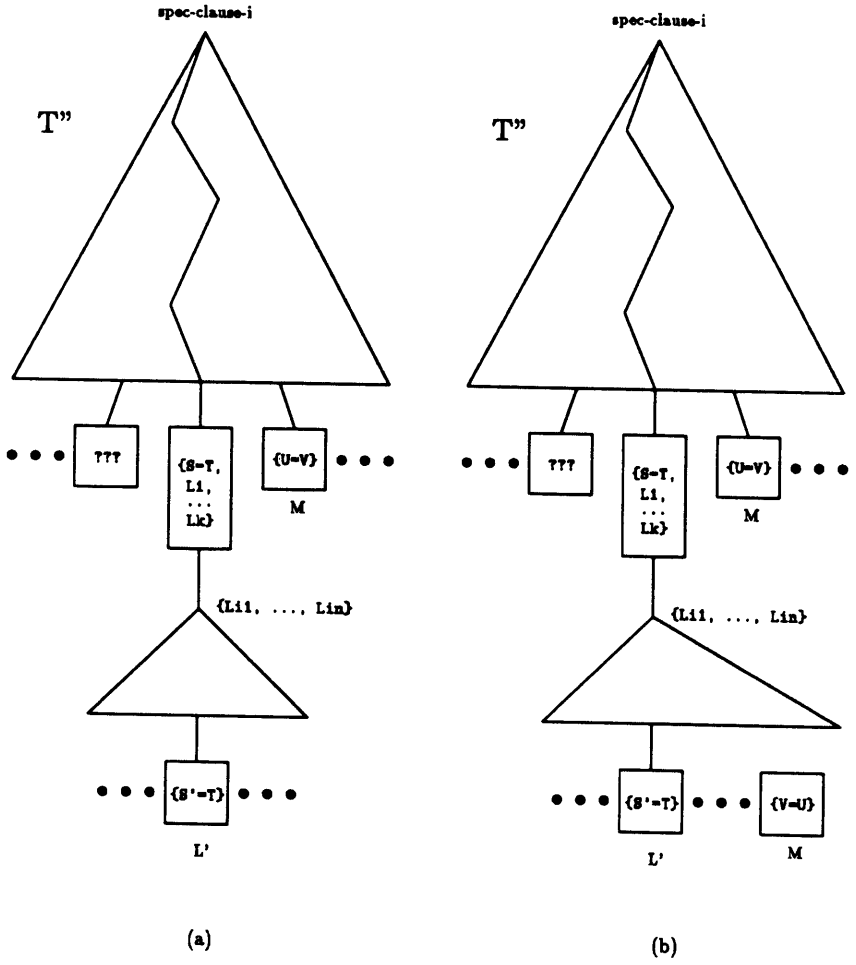


Figure 9.7: The two cases where references to the target U (leaf node M) may appear after a patch proof is put in for target T. (a) shows the case where the patch proof does not use M. (b) shows the case where it does.

principle change across a redistribution. Consider Figure 9.7. Suppose we wish to calculate a target condition for target U, in order to check pairs (V', U), after having decided to keep (carry out) the first pair (S', T). There are two possibilities:

- If the flow arc leaf node (M) for target U is not used in the patch subproof (Figure 9.7 (a)), then the target condition calculated for U from the patched proof is identical to that calculated for U from the original proof.

- If M is used in the patch proof (Figure 9.7 (b)), then the target condition calculated from the original proof *conjoined with the satisfaction of T's target condition by S'* suffices to maintain the correctness of the overall program. Note that even though additional references to M are added by the patch proof, they are all added below the root of the patch subproof, hence if the patch subproof's root clause is true in the program resulting from the second redistribution, then all we need to do to guarantee correctness is to make sure that the other occurrences of M are patched correctly.

I have just sketched a proof of

> **Theorem 2:** Given a proved correct program initially, and given a set of source/target pairs $\{(s_0, t_0), \ldots, (s_k, t_k)\}$ for the program, if we change the program by retracting each of the flow arcs ending at a t_i and assert a new (possibly extended) flow arc corresponding to each of the pairs, *not changing any of the other* `:PROGRAM-STRUCTURE` *nodes used in the proof*, then the program remains correct if for every pair (s_i, t_i), $\mathrm{tc}_{t_i}(s_i)$ is true, where all target conditions tc_{t_i} are computed with respect to the original proof structure.

To summarize, the insight is that *the overall proof will remain correct as long as all target conditions are satisfied in the fully modified program.* It is crucial here that no structural aspects of the program change other than the flow arcs involved in the redistributions. As long as we check that subsequent new sources satisfy the target conditions of subsequent targets and that previously carried out redistributions remain correct after subsequent redistributions are carried out, we can ignore any leaf occurrences below the

roots of the patch proofs. The key point is that the target conditions are calculated from the *original* proof, so that patch proofs are unnecessary to continuing the procedure. Note that carrying out a redistribution recursively may alter specifications of the recursion boxes; hence Theorems 1 and 2 do not apply to that case. I discuss recursion in the next section.

Theorem 2 has implications for screening strategies, but I shall put off discussion of them until after discussing recursive pairs.

9.4 Target Condition Theory III: Recursive Redistributions

To define precisely the concept of *recursive* as it applies to source/target pairs, I must first fix some terminology. A *call-ancestor program* of a program element named by a pathname `$P.i1. ... .ik.a`, $k \geq 0$, is either the program `P` or a call-ancestor program of the element `$P1.i2.....ik.a`, where `P1` is the program type of the box `$P.i1`. Let `Pj` be the program type of the box `$P.i1.....ij`. Note that `Pj` is a call-ancestor program of `$P.i1.....ik.a`, if $j \geq k$. Define $box_{\texttt{Pj}}$(`$P.i1.....ik.a`) to be the box named by `$P.i1.....ij`. That is, it is the box of the program `Pj` through which one must descend to get (eventually) to the named element.

I term a source/target pair *recursive* if and only if the source s and target t have a common call-ancestor program Q such that $box_Q(s)$ and $box_Q(t)$ both lie in a recursive branch of Q (see Section 7.2.2 for what it means to "lie in a recursive branch").

Intuitively, a recursive source/target pair actually corresponds to an infinite set of source/target pairs, one per level of recursion. For example, the pair (`$MY-REVERSE.REC.L-OUT`, `$MY-REVERSE.CONC.NC.C1`), corresponding to the left-hand curved flow arc in Figure 1.4 (Section 1.3), is recursive because source and target have the common call-ancestor program `MY-REVERSE`, and both lie within its recursive branch. This pair corresponds to the infinitely many pairs given by the following pair schema: (`$MY-REVERSE.REC`^{k}`.L-OUT`, `$MY-REVERSE.REC`$^{k-1}$`.CONC.NC.C1`). By contrast, the pair (`$APPEND.CPY.L-OUT`, `$APPEND.NC.C1`) is *not* a recursive pair (even though `NCONC` is a recursive program) because the only common call-ancestor program is `APPEND`, which is non-recursive.

Note that a recursive pair may or may not be treated as such: it requires additional effort to carry out such a pair correspondingly at all levels of recursion. Treating such a pair nonrecursively means only carrying it out at the top-level, that is, changing the structure of the top-level program but leaving the implementations of the recursion boxes unchanged. Treating a recursive pair non-recursively gives the pair exactly the same status and properties as non-recursive pairs; thus, the system can be set to treat all pairs non-recursively, and the theory discussed in the previous sections applies as is. However, it is usually beneficial (where possible) to carry out redistributions of recursive pairs at all levels, translating the savings of one pair to all invocations of the recursion. This section examines the relation between treating a pair recursively and its target condition.

9.4.1 The Bad News

Unfortunately, for recursive source/target pairs (treated recursively) proving that the source always satisfies the target condition does not guarantee the correctness of all of the infinitely many pairs corresponding to the pair; hence, it does not guarantee correctness of the program as a whole. The intuitive "reason" is that the actual target condition of each of the infinitely many targets is different, because each one is, after all, found in a different call context than every other. Moreover, each of the sources will generally have slightly different properties. The loophole in Theorems 1 and 2 lies in the clause "not changing any of the other `:PROGRAM-STRUCTURE` nodes used in the proof". By carrying out redistributions at all levels, we are also changing the implementations of the recursion boxes. This change is only guaranteed to preserve the weakened specification of the recursive program obtained during the hierarchical target condition computation. However, to maintain obvious correctness of the proof, the recursion boxes must satisfy their original specifications, which may be stronger than the weakened ones guaranteed by the target condition.

As an example, consider the program `SUM-ELTS` in Figure 9.8. This is a simple program that sums the first ten elements of an input array. It is implemented in a straightforward tail-recursive fashion. It is easy to see that `$SUM-ELTS.SUM-ELTS-REC.I` is always equal to `$SUM-ELTS.SUM-ELTS-REC.SUM`, because `SUM-ELTS` passes zero to both `SUM-ELTS-REC` inputs. But this means that the pair (`$SUM-ELTS.SUM-ELTS-REC.I`, `$SUM-ELTS.SUM-ELTS-REC.ADD.N1`)

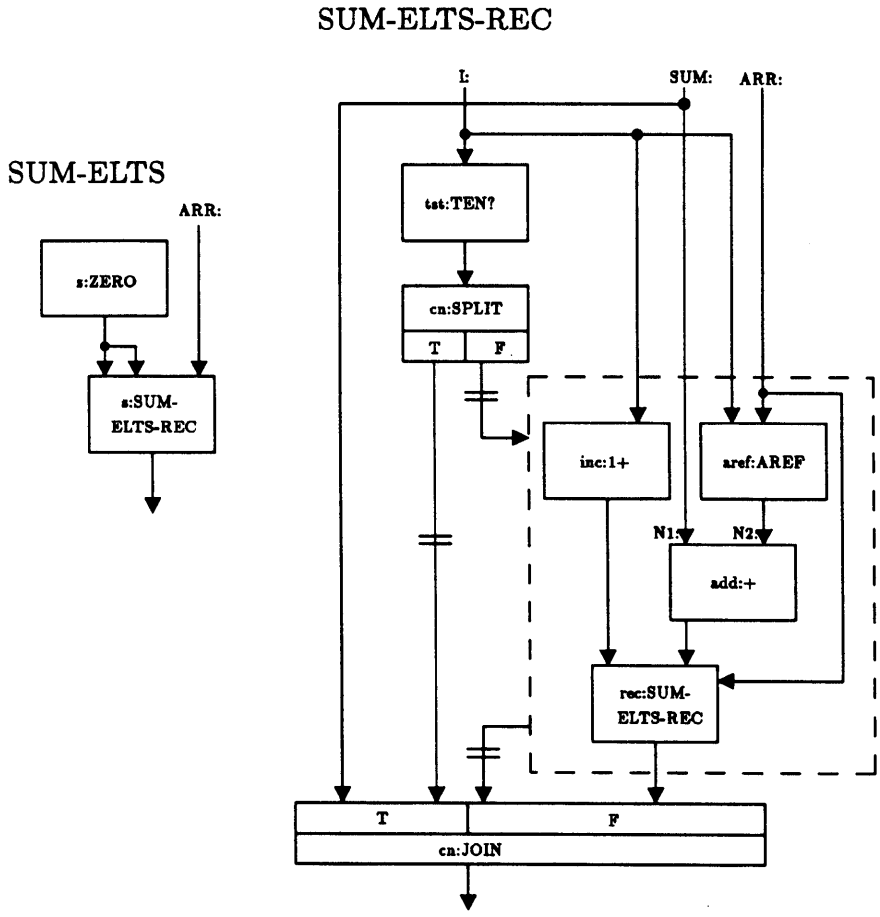


Figure 9.8: A tail-recursive program illustrating how the target condition fails to capture correctness at all levels of a recursion.

passes the target condition check. In fact, doing this redistribution at the first level is fine, but carrying it out recursively leads to an error: the program would always compute ARR[9] + 9 instead of the correct sum.

An important exception to this failure of target conditions is that a non-recursive pair within a program p, which can be proven correct by showing that the source satisfies the target condition, cannot become incorrect simply because p is called within a recursive program. This is easily seen because the correctness proof of the redistribution must be independent of how the nonrecursive program is used. This argument depends on the fact that correctness of a calling program may only depend on the proven specification of the called program and not on internal implementation details, because the system only guarantees preserving the program's specification.

The problem with treating pairs recursively, then, is that checking that a source satisfies a target condition for a recursive pair only gives information about the top-level member of the infinite class of redistributions corresponding to the pair.

9.5 Applications to Checking Redistributions

To use target conditions for screening, we must address the problem of recursive pairs as well as the problem of re-checking previous redistributions.

- *Recursive pairs.* Since the target condition fails to capture the correctness condition of recursive pairs, either another check must be done, such as the top-level check in **Inv-Screen**, or else the pairs must not be treated recursively.

- *Previous pairs.* Since target conditions of previous pairs won't necessarily be operational with respect to the most recent target U, we can't check them without re-executing test cases. The target condition for U itself, however, *is* operational, by definition, so we can check it *first*, discarding the pair if it fails to pass. For a successful pair there are three approaches:

  - *Update-and-top-level-check strategy.* do a top-level check, via **Inv-Screen**, after updating test case traces;

- *Update-and-re-check strategy*: update traces, then re-check each of the previous target conditions; or
- *No-re-execution strategy*: arrange only to consider subsequent pairs that cannot possibly change the correctness of previous pairs.

I have investigated these three approaches to adding the extra checking required to use target conditions without explicit patch proofs. Each of these has somewhat different power and applicability conditions. I discuss first the strategy implemented in the pseudo-code of Section 9.1. It is the most powerful, in that it handles redistributions that are to be carried out recursively. The next subsection discusses the two alternatives.

The *update-and-top-level-check strategy* assumes that re-executing test cases can be afforded once per *successful* redistribution. This is the simplest approach, but it assumes that in addition to the proof structure input, one has the effective top-level optimization invariants required by **Inv-Screen** as well. Basically, after a pair passes the target condition check, the system carries it out structurally, updates test case traces, and checks top-level invariants by calling **Inv-Screen**. Calling **Inv-Screen** guarantees that the altered program is correct on the test cases; it provides only indirect evidence that target conditions of previous pairs remain satisfied in the altered program.

It would be sufficient for correctness simply to call a compile-time certifier to check each recursive pair. However, that approach would result in many unsuccessful calls to the certifier, because it is rather often the case that a recursive pair will pass the target-condition test but fail the **Inv-Screen** check. For example (see Section C.2.5), the merge-sort optimization resulted in 254 pairs passing the target condition test, but 233 of them failed the subsequent **Inv-Screen** test. If the system instead had had to call a certifier unsuccessfully 233 times (in addition to the 15 successful times), it certainly would have taken far too long. Note that since the target condition fails to capture all information necessary to correctness of the redistribution, the certifier's job is also more difficult than with non-recursive redistributions.

Given this, one might wonder why the system should perform the target condition test on recursive pairs, given that they must be checked by **Inv-Screen** anyway. The answer here is again based on efficiency: the target condition test can be done without re-executing the test cases and yet prunes many potential candidates. Pairs failing the target condition check are not,

of course, checked by **Inv-Screen**. In the merge-sort example, 6701 pairs passed the syntactic pruning phase, virtually all of which were recursive pairs, yet only 254 passed the target condition test. Thus, about 96% of the pairs were pruned by the less expensive target condition test.

9.5.1 Alternative Checking Strategies

Here are two alternatives to calling **Inv-Screen** every time a pair passes the target condition test. I have implemented the first and run a few experiments using it. I have not implemented the second approach at all, for lack of time. These alternatives are important in that they have somewhat different applicability conditions, hence may be used under conditions in which the previous strategy is inapplicable.

The *update-and-recheck strategy* also assumes that re-executing test cases can be afforded once per successful redistribution To check the first redistribution, simply test the target condition as usual. Then, carry out the redistribution and update the test case traces by executing the altered program on the test case inputs. No further checking is required, since this is the first redistribution. For subsequent rounds, screen using the target condition check and if a pair passes, carry out the redistribution and update the test case traces. The target condition check guarantees that the current pair is correct with respect to the test cases. Updating the traces for the altered program guarantees that values recorded in the traces accurately reflect values in the altered program. Now, we simply need to recheck the target conditions of all previous pairs to make sure they remain satisfied in the altered program. The reasoning above based on the correctness proof alterations showed that it is enough to prove that all sources satisfy the corresponding target conditions in the altered program structure. By carrying out all redistributions and rechecking them, we guarantee that the program remains correct on the test cases. Here is pseudo-code implementing the update-and-recheck strategy.

Procedure **EB-Screen** (SRC-TRG-PAIR) : Boolean
[Assumes program datastructures are initialized appropriately
for the current program to be optimized.]
TRG-CND ← *find-target-condition*(*target*(SRC-TRG-PAIR))
If TRG-CND evaluates to true in every test case trace
Then If *update-traces-and-recheck-previous-pairs?()*
Then If *certify?* is available
Then return *certify?*(SRC-TRG-PAIR)
Else return true.
Else return false.

Note that, unlike the update-and-top-level-check strategy, this does not call **Inv-Screen**, hence it can be used even when effective top-level invariants are not available. It has drawbacks, however, such as not being usable when redistributions are to be carried out recursively. Note that we may still use this strategy for recursive programs, but only by sacrificing the power of having redistributions carried out recursively. Another disadvantage is that the number of checks per successful pair grows linearly with the number of successful pairs; for large programs this can slow things significantly relative to the single top-level check approach.

Note that it is possible to avoid many of these checks if one can first isolate those that could possibly have become false—if none of the values named in the target condition are causal successors of the target of the new redistribution, then it's truth cannot have changed. I did not implement this refinement, but expect that it would present no difficulty.

The *no-re-execution strategy* assumes that test case re-executions must be avoided at any cost. The approach is simply to avoid any pair satisfying one of

- Its source value or the values of any pathname terms in its target condition could have been changed by a previous redistribution; or
- Its incorporation could change either a source of a previous pair or some other value appearing in the target condition of a previous pair.

Obviously, if the program element values appearing in the target conditions of previous pairs don't change, the target conditions are still satisfied. Note

that one can decide (conservatively) whether one of these interactions exists based only on what pathnames appear in the target conditions and on the structure of the program. This strategy clearly sacrifices some optimization power, as there are often cases (in `MY-REVERSE`, for example) where early redistributions alter a source's value so that it satisfies a target condition. In `MY-REVERSE`, the first two redistributions caused the source of the third to always equal the target of the third. On the other hand, causally disjoint redistributions are common as well. Note that this approach is not applicable if redistributions are to be carried out recursively.

9.6 Discussion

All the strategies above depend on computing target conditions. Proof restructuring behaves linearly in the proof-tree sizes, unless the exponential resolution rule is unrestricted. In all experiments, the approach employed by the system described in Section 9.2.5 was not a significant consumer of time in relation to the rest of the computation.

The actual PCCU-based procedure for extracting target conditions is also linear in the size of the proof trees and the size of the output. Unfortunately, due to the inherent complexity of program correctness reasoning, target conditions can be annoyingly large and redundant, particularly when expressed in CNF. Again, however, the system's approach to target condition computation and evaluation, including partial evaluation in test cases, renders target conditions usable in practice. I expect that, with further research and engineering, target conditions will bear out as a practical tool for use even with complex programs. Heuristics can be adopted to hold their sizes down at the expense of generality, though I have not as yet needed to do so.

9.6.1 Comparison of Strategies

When it is usable, the update-and-top-level-check strategy is clearly better than the update-and-recheck strategy, both because it handles recursive pairs and because it tends to be significantly faster in that it does only the single, top-level check per successful pair instead of the many individual target condition checks. Of course, programs for which it is impractical to give optimization invariants are not handled by update-and-top-level-check, but

can be handled by update-and-recheck.

The update-and-top-level-check strategy is much more powerful than the no-re-execution strategy, both because it handles recursive pairs and because the latter must discard interacting sets of pairs. No-re-execution is faster, however, as it does not update test cases and does not do any checks other than the successive target condition checks. This may be the best strategy for highly complex or highly inefficient (non-recursive) programs, where running test cases more than once is impractical.

9.6.2 Wrong Answers

EB-Screen (all three strategies) can give both false positive answers and false negatives. As with **Inv-Screen**, false positives are simply due to the fact that I have substituted test case reasoning for general theorem proving; hence the system can be fooled by coincidences in the test cases.

False negatives are more interesting. There are two reasons **EB-Screen** can give a false negative answer: either a target condition exists, but it cannot be extracted from the given proof structure; or there is no proof of the program from which a general enough target condition can be extracted. I call this second type a "fundamental false negative."

Section 9.2.4 gives an example where the input proof structure (Figure 9.6 (b)) can fail to give a general target condition, yet where a better proof structure (Figure 9.6 (c)) exists. It is trivial to extend that example to an actual false negative redistribution. The key is the lack of generality in the target condition. I give below an example of the second type of false negative.

Fundamental False Negatives. Consider the program shown in Figure 9.9 (a). The primitive program `MOD2` returns 0 if the input is an even number, 1 if it is odd. The specification of the program `ID-0/1` defines the output to be 0 if the input is 0, 1 if the input is 1, and otherwise does not constrain the output. The overall program must satisfy the same specification as `MOD2`.

Fact: There is no target condition for `$P.B2.C` that allows all correct redistributions. To see this, assume that there is one, and call it τ. Any target condition for `$P.B2.C` must not depend on the choice of implementation of the box `$P.B2`, because only box specifications are allowed in proofs. Moreover, for any test case, the values of the pathname terms that may appear

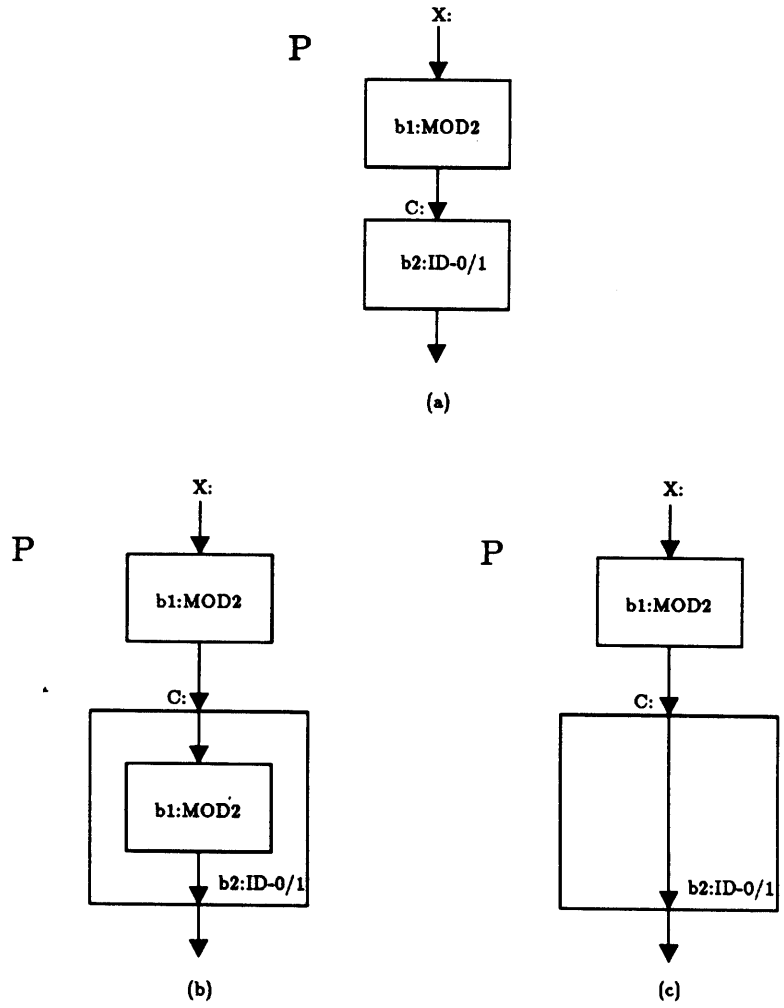


Figure 9.9: A fundamental false negative for IEBR. (a) For any correctness proof of `P`, the target condition for `$P.B2.C` must not depend on the implementation of the box `$P.B2`; it may only depend on the box specification. (b) An implementation of `$P.B2` in which the redistribution from `$P.X` to `$P.B2.C` is correct. (c) An implementation of `$P.B2` in which it is not.

in the target condition of $P.B2.C must be the same for any implementation of $P.B2, because the portion of the program containing pathnames operational with respect to the target $P.B2.C is not causally effected by the implementation choice. Therefore, any redistribution with target $P.B2.C passing **EB-Screen** must be true independently of the implementation of $P.B2.

But now consider the program in Figure 9.9 (b), where MOD2 is used to implement the ID-0/1 specification. Clearly, the redistribution with source $P.X and target $P.B2.C preserves overall correctness. Therefore, τ must be satisfied by the source $P.X in all test cases, because τ was assumed to find all correct redistributions.

Consider, on the other hand, the program in Figure 9.9 (b), where a single flow arc implements the identity function. Clearly, the redistribution is incorrect here, as the result would be just an identity program. But then there must be some input on which τ is not satisfied by $P.X. But since τ is the same for both programs, and evaluates the same way on all sources and all test cases, this is a contradiction. Therefore, no such τ exists. The redistribution ($P.X, $P.B2.C) is a fundamental false negative for IEBR. Note, however, that IBR would correctly give a positive answer for program (b) and a negative answer for (c).

Note that to prove the correctness of the fundamental false negative redistribution (program (b)), new details of the internal implementation of the $P.B2 box must be introduced into the correctness proof of P. Originally, P's proof used only the box's specification.

This example illustrates how IEBR restricts its output to pairs that (if true) have more routine proofs. The next subsection makes this more precise.

9.6.3 Routine Redistributions

Since a target condition may refer only to elements that are not causal successors of the target, we can conclude the following

> **Theorem 3:** Any correct, non-recursive redistribution passing the target condition test of **EB-Screen** must be provable independently of the implementations of any boxes that are causal successors of the target.

A consequence of this is that a certifier need not introduce structural axioms of box implementations for boxes succeeding the target. This can potentially reduce the theorem prover's search space greatly: assuming the target is on the average somewhere in the middle of the program, we might expect this to reduce the number of axioms a prover must consider by about half. Since the search space is usually exponential in the number of axioms, this can potentially reduce the size by taking the square root.

The redistributions passed by **EB-Screen** are, therefore, more routine than many passed by **Inv-Screen**. They represent only incremental changes to the program's correctness proof. As discussed earlier, the correctness of the altered program can be understood by understanding the original correctness proof and understanding each of the patch proofs used to justify the redistributions. Hopefully, each patch proof is simple compared to the entire proof. Moreover, Theorem 3 and the search space size argument above give us reason to believe that it will be much easier to prove redistributions passed by the target condition test than arbitrary redistributions.

A major benefit of restricting only to routine optimizations is that the compile-time certifier's job is easier. It will not be called unless there is good reason to believe both that the thing it is to prove is true and that the proof will be relatively easy compared with a proof of correctness of the entire program, and compared with correctness proofs of arbitrary redistributions. Of course, because of potential interactions between sequential redistributions, the certifier must make sure that previous patch reasoning is still valid in addition to proving the patch for the new redistribution.

Theorem 2 guarantees that this is the whole story for non-recursive programs. For recursive programs, however, more complicated reasoning may be required to justify recursive redistributions. Such redistributions, of course, are harder to justify than non-recursive ones, particularly because the target condition does not necessarily guarantee correctness. Thus, additional reasoning may be required, such as inductive proofs of useful properties. Nevertheless, I believe that target condition screening is a useful heuristic for ruling out extremely difficult recursive pairs, leaving only the more routine ones that pass the target condition check.

As an example, consider the optimizer's performance on the `LR1-REM+REVAPPEND` example (Section C.2.3). IBR eliminated the `$LR1-REM+REVAPPEND.REVAPP.REV.SNG.CNS.RPD` box via a recursive redistribution, while IEBR gave a fundamental false negative. The correctness argu-

ment requires reasoning about two separate cases: cells in the middle of the list and the last cell in the list. If we eliminate the above-named box, then instead of the CDRs of all cells being initialized by it, non-end cells get CDRs initialized by the $LR1-REM+REVAPPEND.REVAPP.REV.CONC.NC.RPD box, but the end cell gets initialized only by the $LR1-REM+REVAPPEND.REVAPP.APP.NC.RPD box. Thus, the form of the correctness proof changes from a uniform handling of the CDRs of the result list cells to an argument by cases. Moreover, the "saving" pointer operation performed in the APPEND box is not local to the redistribution, which occurs within the reverse box. Thus, only a global coincidence allows the redistribution. This shows how **EB-Screen** can help screen out difficult recursive redistributions as well as non-recursive ones.

9.6.4 Do We Need Recursion?

Even easy theorems about recursive programs are hard.[5] It is worthwhile asking the question of whether recursion is worth it.

It may at first seem that non-recursive programs are uninteresting or "toy"; however Waters (1979) has estimated that the great majority of all programs are written without recursion, using only simple loops and straight-line code. Loops, in turn, can be represented using series expressions, so these programs could all be written without explicit iteration of any kind. Thus, restricting to non-recursive programs may be interesting from a practical standpoint.

Moreover, if adopting more powerful representations, such as series expressions, leads to more powerful automated program development tools, it may be well worth adopting programming styles that use less recursion and more explicit approaches to iteration. Note that more complex recursive structures than tail recursion can be captured by generalizing series expressions from sequences to other mathematical objects such as trees (for tree recursion) and graphs (for more general types of recursion). I have not explored this past the conceptual stage, but it seems a promising line of research.

Two arguments favor the use of recursion, however. First, some redistributions exist only in the recursive form of a program, because they involve

[5]This oxymoron may need clarification: "easy" means "easy for humans," while "hard" means "difficult for automated theorem provers."

a recursion box (see Section C.3.1 for an example). Second, some problems are most naturally solved recursively (such as *Tower of Hanoi*), hence should be programmed that way. The point is that it may be easier to solve some problems using recursion, and it may also be easier to prove correctness of recursive solutions than of non-recursive ones.

The answer to the recursion question, then, is yes, we need recursion; however, we also need to understand the relationship between the recursive and non-recursive forms of a program better. We need to be able to automate the transformation of programs from one form to the other, along with their correctness proofs. This is an interesting area of future research.

Chapter 10

Proofs of Quasi-specifications

It is indisputable that formal specifications and proofs that programs satisfy them are difficult to produce. As discussed in Section 4.2, I address the specification difficulty by allowing relative optimization invariants, which are generally much easier to specify when allowed to refer to the outputs of the original program.

To make it easier to use IEBR—which requires a proof—I define a formal construct called a *quasi-specification* that allows one to automatically produce proofs capturing some, but not necessarily all, of the teleological information relevant to a program. Technically, a quasi-specification is a statement of program function which may include references to internal elements of the program in addition to inputs and outputs. It turns out to be rather easy to automatically compute proofs of certain classes of quasi-specifications relevant to programs. I discuss three such ways below, together with automatic procedures for producing proofs of quasi-specifications. Quasi-specifications and their proofs can capture enough routine information about a program to allow useful optimizations, yet still be produced quickly and completely automatically.

10.1 The Default Quasi-specification

The first automatic quasi-specification proof procedure, *default-proof*, is the weakest, in that it incorporates the smallest amount of information. It turns out that it is, *de facto*, exactly the specification that traditional compilers

preserve.

The basic idea is to define a default quasi-specification predicate for a program that is true precisely when all of the program's `:PROGRAM-STRUCTURE` leaves are true. In other words, it is postulated to follow from the conjunction of the program's structural axioms. This includes axioms corresponding to all flow arcs, all conditional outputs, and all box specification axioms. *Default-proof* takes in a program and puts out a single proof-tree containing one copy of each of the program's structural axioms and constructs (trivially) a proof of the default quasi-specification predicate based on a single assumed defining lemma.

This would at first appear to allow *no* changes to the program; however, there are several ways freedoms enter. In particular, it allows all identical-value redistributions, because all program structure axioms are expressed in terms of the values of ports, not in terms of actual presence of program elements. Moreover, box specifications may be sufficiently underdetermined to allow optimizations either within a box implementation, due to properties of its call context, or by communicating a value from within one box to within another. Note that while the top-level program's proof is constructed by *default-proof*, the proofs of its box's type programs may be constructed in a more informative way, either by hand or by one of the methods below. This allows, for example, considerable optimization freedom based on the richly detailed proofs from a software library.

Several examples were carried out using only *default-proof*; Appendix C has two: `POLY` and `FIB`.

10.2 Quasi-specifications for List Programs

It is common to write list-manipulating programs that should not modify any of their input arguments. They are, however, allowed to allocate new memory cells. To incorporate this information into proofs, I designed the procedure *lr1-proof*, which defines and proves for each program a quasi-specification predicate that conjoins the program's abstract list behavior with its side-effect-free nature.[1] I define this more precisely below.

[1] "LR1" (List Representation 1) designates a particular representation of abstract lists in terms of memory cells and stores. It is described more fully in Appendix C.

Default-proof does not capture this information because it requires that the cell at the source end of a flow arc be identical to the cell at the target end; this disallows a redistribution replacing the source by one representing an isomorphic list. Moreover, it requires that the store output of the program maintain exactly the same value across optimization, with no more nor fewer allocated cells and identical bindings; thus, optimizations leading to fewer allocated cells would not be allowed.

To use *lr1-proof*, the programmer must distinguish for each program (including all box type programs) certain specification clauses (via declarations) as *lr1-specs*. These are intended to be the ones capturing the abstract list properties. In my formalism, these can be distinguished by the fact that all occurrences of cell pathnames and store pathnames appear as arguments to the function `LR1` which converts them to abstract lists. An example of an lr1-spec's clause would be, for example,

```
{(= (LR1 $P.C-OUT $P.ENV-OUT) (TAIL (LR1 $P.C $P.ENV)))},
```

where `TAIL` is the abstract list function of taking the tail of a list. Non-lr1-specs include all lower-level information such as freedom from side-effects, freshness of output structure, etc.

The programmer must also declare which cell inputs and outputs are to represent abstract lists, since not all cells are necessarily used to represent lists.

The first property incorporated in the definition of the LR1 quasi-specification is the program's freedom from side-effects. In my formalism this is denoted by the predicate `APURE?`, a predicate of two store arguments. It is easy to construct a proof of the `APURE?`ity of the program (assuming each box modifying the store has an `APURE?` specification) because `APURE?` is transitive. Note that some boxes may be `APURE?` only when some preconditions are satisfied. Proving arbitrary preconditions is, of course, uncomputable; the system simply postulates them true in constructing the proof.[2]

Next, each flow arc involving a cell declared to represent an LR1 list[3] is

[2]This can, in principle, result in a target condition that is overly general, because some necessary teleological dependencies may be left out of the proof. On the other hand, the freedom from side-effects of most modules seldom depends on such preconditions. Thus, such cases should be rare. This phenomenon has not arisen in any example considered by the system.

[3]An "LR1 list" is an abstract list represented by a collection of memory cells and a

proved to enforce equality of abstract lists. That is, the source and target ends of the arc, paired with appropriate stores as determined from declarations, are proven to be equal as abstract lists. This may involve only equality reasoning, or it may involve using the APURE?ity specs of intervening boxes, but in any case *lr1-proof* constructs such a *list-equality subproof* for each flow representing a list. Note that these proofs may also involve postulating the truth of unknown preconditions.

Finally, *lr1-proof* uses *default-proof* to form a proved quasi-specification for all flow arc axioms for types other than list, together with all conditional axioms and all box lr1-specs. This has the effect of "freezing" the abstract list structure, but leaving some details of the side-effect behavior free to change.

lr1-proof then simply defines and proves a quasi-specification predicate representing the conjunction of APURE?ity, list-equalities, and the lr1-spec default specification.

Most of the examples run have been done using proofs produced by *lr1-proof*. Examples include MY-REVERSE and LR1-REM+REVAPPEND. It is remarkably successful (and fast) at capturing the low-level list representation information while still being fully automatic (except, of course, for declarations needed).

10.3 Quasi-specifications for Set Programs

The SR set representation is built on top of the LR1 list representation, so it captures even more high-level information. A set is represented by a list of its elements. Some set operations require that the list contain no duplicated entries (the no-duplicates invariant) for correctness, so all operations must maintain it, at least before optimization in context. Refer to Appendix A, particularly Figure A.1, for more information on the LR1 and SR representations.

Maintaining the no-duplicates invariant can be expensive in operations like SR-ADD that adds a member to a set. Prior to adding the new element, SR-ADD must check to be sure it is not already an element. This makes adding the element cost linear time. In some applications a set will not need to be represented with no-duplicates, since the operations requiring it aren't used on the set; thus, the optimizer should remove the check.

---

store in a certain way. Thus, the declaration contains more than the information that the cell represents an abstract list; it also declares how the list is represented.

To capture this, *sr-proof* builds upon *lr1-proof* but also maintains the no-duplicates invariant. Thus, as before, *sr-proof* constructs an APURE?ity proof. Next, corresponding to list-equality subproofs are set-equality subproofs for each cell flow arc representing a set. Next, each program must have declarations, provided by the programmer, stating one of three things:

- It requires that no-duplicates be true of its inputs for correctness; or
- It requires that no-duplicates be true of its inputs only to maintain no-duplicates in its outputs; or
- It doesn't require no-duplicates at all.[4]

Sr-proof then performs the analysis that decides whether each box's no-duplicates spec is to be incorporated into the final proof. Finally, all box sr-specs (as declared by the user) and necessary no-duplicates specs are conjoined into a default subproof.

These elements are combined by conjunction into a proof of the SR quasi-specification. Note that I have not implemented *sr-proof*, but have carried out the examples by hand simulation. These include SR-ELT?+UNION and SR-CHOOSE+REM in Appendix C.

10.4 Proved Quasi-specifications in General

In general, an automatic quasi-specification proof procedure might be written for each different representation (abstraction function). This effort could be amortized, however, over all the programs based on that representation. Moreover, I expect that further research on these techniques may systematize and unify them.

Note that there is no guarantee that the information captured by a quasi-specification proof procedure is compatible with the information in the optimization invariants used. This coherence must be enforced by the user. But since these techniques—quasi-specification proof procedures and relative optimization invariants—represent tradeoffs for usability, the user must accept more of the burden of correctness checking in return for having more of the

[4]Note that just because some operations on a data type require an invariant, others may be oblivious to it.

necessary inputs produced automatically. After all, using hand-produced proofs of absolute, effective specifications eliminates the coherence problem. In practice, it has not been difficult to enforce the necessary coherence, however.

Moreover, it is not clear what the relationship is between certification and the use of quasi-specification proofs, i.e., whether it makes the job any harder or easier. On one hand, it may be easier since the optimizations cannot capitalize on all the information in a program; rather, they must be involved with the representation invariants captured by the proof procedure and cannot involve the full abstract functionality. On the other hand, since the proof contains postulated axioms, which may substitute for necessary teleological dependencies, there may be hidden subtlety in proving safety. My experience with the examples indicates that it is no more difficult to check optimizations found when the input is supplied by a quasi-specification proof procedure than when the proof is supplied by hand. If these subtleties do arise, however, we can always restrict the modules to have *unconditional* APURE? specifications. More generally, all specifications used by automatic proof procedures might be required to be unconditional. In that case, no teleological dependencies would be omitted, so the target conditions generated would have exactly the same properties as any others.

Part III

Analysis and Discussion

Chapter 11

Comparison of Algorithms

This chapter summarizes and extends previous remarks comparing the two optimization algorithms. While the main point of this thesis is only to demonstrate the *feasibility* of automating the search for redistributions, it is nevertheless instructive to compare the two algorithms to each other.

11.1 Power

IBR is much more powerful than IEBR. More precisely, **Inv-Screen** gives no false negative answers, while **EB-Screen** can do so. In fact, **EB-Screen** is susceptible to two types of false negatives: those due to proof inadequacies and fundamental false negatives. (See Chapter 9 for examples of both.)

Both approaches are susceptible to false positives, simply due to the fact that the test cases cannot be guaranteed, *a priori*, to be free of coincidences. Every false positive given by **EB-Screen** will also be given by **Inv-Screen** (by construction); on the other hand, not every false positive of **Inv-Screen** will be given by **EB-Screen**. The reason is that **EB-Screen** does an additional check on top of that done by **Inv-Screen**; the target condition test can fail a redistribution while the top-level check passes it. I do not believe this phenomenon has occurred in any of the experiments, though it is difficult to know for sure—a pair failing the target condition test is not normally checked by the top-level invariant test, and IBR and IEBR do not always examine exactly the same sets of redistributions. There are, of course, several cases where a *correct* redistribution passes the top-level test, but not the

target condition test.

11.2 Time and Space Costs

The costs of running the two algorithms are different. The actual run-time and -space functions are too complex to derive, and the implementation has not been tuned for speed. Thus, it is not possible to draw hard conclusions about which approach will be less costly in a practical setting. On the other hand, there are some qualitative insights to be had based on the general design of the algorithms.

11.2.1 Run Time

Screening Only. Assume first that the two screening procedures give the same answers on each candidate pair for a given example (this happens, for example, in **MY-REVERSE** and **APPEND**). This ensures that the algorithms examine exactly the same set of pairs (in the same order), giving us a better idea of the relative costs of the screening procedures themselves.

In this case, IEBR seems to be significantly faster than IBR due to the need for the latter to carry out the redistribution and re-execute test cases for each candidate pair, while IEBR only does this once per *successful* pair. The number of successful pairs is usually several orders of magnitude lower than the number of candidates. We can derive crude run-time formulae for each of the approaches as follows.

The dominant term in IBR's time cost is proportional to the number of screened pairs (number of pairs not pruned syntactically) times the total aggregate size of all test case traces.[1] Thus,

$$\text{Time}_{\text{IBR}} \approx k_1 \times \text{number-screened-pairs} \times \text{sum-of-trace-sizes} + o_1$$

o_1 represents other terms that do not grow as fast with program size. Note that the pairs eliminated syntactically are dealt with in various aggregate ways, hence are not constructed individually, so the contribution of *prune-syntactically* is small by comparison.

[1] However, as optimization proceeds, the trace size will decrease (in IBR's case only) because virtual structure need not be evaluated for IBR. I ignore this effect here, though it is probably significant for some programs.

The dominant term in IEBR's time cost is proportional to the number of screened pairs times the average cost of evaluating a target condition in a test case, times the *number* of test cases (not the aggregate size). Thus,

$$\text{Time}_{\text{IEBR}} \approx k_2 \times \text{number-screened-pairs} \times \text{avg-tc-eval} \times \#\text{test-cases} + o_2$$

Given these two crude models, we can see that as the program gets larger (hence its traces get larger) IBR gets slower by comparison, assuming the target condition evaluations and number of test cases stay about the same. It is very difficult to apply these models, since the average target condition evaluation time depends on its size; its size, in turn, depends on how often the target appears in the proof, and that does not depend simply on the size of the program. It depends more on the connectivity and intended behavior. It also depends on how much generality we are willing to give up in the target conditions: if we restrict to "equal to the old source" as the only target conditions, then obviously evaluation doesn't grow at all with program size.

The "low order" terms can have quite large constants, also; for most of the programs tested, they had significant (but not overwhelming) effects. In practice, IEBR's low order terms tend to be larger than those of IBR; they include the costs of tree-restructuring, target condition derivation, and trace updating for successful pairs. IBR, on the other hand, has very little extra low-order baggage.

Overall. For most programs, the two screening procedures give different answers, so IEBR and IBR examine different pairs. IBR's greater power gives it a considerable natural advantage as follows. Any box eliminated by IEBR will also be eliminated by IBR (if it considers it), but not conversely. If IBR eliminates a box, it need not investigate its substructure, obviously. If, however, IEBR fails to eliminate it, IEBR will investigate it. Thus, the relative weakness of IEBR can result in it examining more candidates. The merge-sort example (Section C.2.5) illustrates this phenomenon: IEBR examined 370 boxes, while IBR examined 318.

In practice, this reduces the run-time advantage of IEBR, but IEBR usually still outperforms IBR. The typical advantage seems to be about a factor of 2 to 3 on the examples I've run. It cannot be overemphasized that these numbers are likely to change when more effort is put into careful engineering. It is still open as to whether for larger programs IEBR will maintain its run-time advantage.

Of course, for large programs, likely the most practical way of using the system would be to look only at the most expensive boxes; thus, a small run-time advantage for one algorithm is likely to be unimportant. Other considerations, such as the degree of confidence in the results are likely to dominate.

11.2.2 Space

Space costs are due mostly to storage of test case traces and maintenance of internal datastructures corresponding to the program. In the current implementation, IEBR must store all intermediate values for each test case trace, while IBR needs to maintain only whether each source and target was reached in the execution of the test case. Thus, significant space savings might be achieved by designing a clever way of storing this information without actually storing the trace values.

If the program surgery required to carry out redistributions is implemented using dynamic allocation and garbage collection (as in this system), then IBR suffers a much greater space cost due to program surgery over IEBR, since IBR does such operations once per candidate. These costs are converted into time costs by the garbage collector, since only one version of the program (together with the information necessary for chronological retraction of alterations) needs to exist at one time.

Other space costs include storing programs, optimization invariants, proofs, and lemmas in appropriate library structures. Storing proofs and lemmas is necessary only to IEBR, of course.

11.3 Routine Optimization

As discussed in Chapter 9, IEBR captures a notion of "routine" optimization that IBR fails to capture. The redistributions passed by IEBR are more routine because they can be proved from fewer structural axioms on the average. I believe this to be true, even for redistributions carried out recursively. This is a double edged sword.

- *Routine is good* if one wants each optimization to be certified by some automatic or semi-automatic means. Given the practical and theoretical limitations on automatic theorem provers, it is important both to

avoid calling the theorem prover as much as possible, and to avoid calling it when it will fail to find an answer. Deep or nonlocal optimizations tend to result in the theorem prover failing to find an answer and being cut off by some resource bound. This is obviously expensive.

- *Routine is bad* if one wants the highest degree of optimization and is willing either to accept the possibility of optimization-induced errors (using some run-time scheme to correct them), or to hand check each optimization.

From the standpoint of restricting to routine optimizations, the choice of which algorithm to use reflects more on how it is to fit in to a larger software development environment than on an absolute good/bad judgement.

Chapter 12

Supplying Design Information

This chapter discusses the issues surrounding the need to supply the system with design information beyond the program's source code. In particular, it addresses the questions of what characteristics the system's inputs should have, and how the system and user can cope with the practicality issues involved.

12.1 Optimization Invariants

Optimization invariants should be cheap to evaluate, at least for the test inputs. Since it is the certifier's job to guarantee safety, we do not require absolute correctness of the invariants. Probabilistic or even only approximate checking would probably suffice as long as it did not allow too many unsafe candidates to pass the screening. This observation significantly enlarges the set of programs to which the approach is applicable, since probabilistic algorithms can be faster than the best known determistic ones. Blum and Kannan (1989) give probabilistic checkers for several problems.

The well known difficulty of formalizing specifications is mitigated by the use of relative optimization invariants, which allow access to the outputs of the original, unoptimized program. As argued in Chapter 4, there are both practical and theoretical grounds for believing that such invariants are easier to find and formalize. Recall, however, that as a last resort one can sacrifice some power to use one of the variants of IEBR that does not require optimization invariants, if even relative invariants cannot be found.

12.2 Explanations (Proofs)

Explanations represented as resolution proofs need to be structured according to the guiding principle of operationality discussed in Section 9.2.4.

In a rich development environment, most proofs are free (in an amortized sense). This is because one constructs programs mostly from reusable library modules, each of which has its proofs done once and for all. The only thing to be supplied for a given user-program is the top-level proof, but quasi-specification proof techniques (as discussed in Chapter 10) can fill the gap here. Adopting an iterative development strategy, one might use *default-proof* on the first round, and progressively incorporate more detail into the proofs on succeeding rounds as the program stabilizes. Finally, if the program were of lasting value, it would be put into the library with a richly detailed proof.

Note that even if we cannot supply an adequate proof input, we can always use IEBR by letting "equal to the original source" be the target condition for every program target. This sacrifices some power, but nevertheless is strictly more powerful than the well known technique of common subexpression elimination.

12.3 Representative Test Inputs

Test inputs should be "representative." This can be defined differently for different uses of test inputs. For the purposes of automatic redistribution, representative means "lacking coincidences." I have no precise description of this as yet, except that there are some properties that are *not* important. I believe the appropriate definition of "representative" for automatic redistribution is close to that used for generating test suites in a "glass box" testing methodology. A "black box" testing methodology is where test engineers design tests without knowledge of any implementation details; this approach seems less likely to produce useful test inputs for optimization screening. In glass box test generation, by contrast, the test designer creates tests specifically for the given implementation.

It is not crucial that the test cases reflect the probability distribution of the intended use of the program; for example, just because the program is to be used mostly on large inputs, the optimizer doesn't need large inputs, unless an efficiency bottleneck is only exercised by large inputs. For example,

MY-REVERSE was optimized using only a single short list, while one might expect that the "typical" list would be large. Note that even if it is prohibitively costly to run the program on the necessary test set, one can (sacrificing some optimization power) use a variant of IEBR that never re-evaluates tests.

It is not crucial that every branch of the code be exercised by the test cases since the system discards candidates where no test case exercised either the source or the target. Since the system can tell when a part of the program has not been executed by any test case, it can avoid trying to optimize it; hence, failing to cover parts of the code won't lead to erroneous optimizations. Moreover, the system could (though I haven't implemented this yet) keep track of how many test cases exercise a given source or target and use this as a heuristic estimate of confidence in a given redistribution. That is, if the source or target is only initialized in one out of ten test cases, any redistributions using it could be flagged as more suspect than others involving well-tested sources. Aside from this, however, the test case problem is difficult. It is no worse than the problem faced by current software engineers who use testing to verify correctness, however.

12.4 Program Structure

With regard to the form of the structural representations of reusable components, the key point is that programs should be represented in a "functionally exploded" form. As much as possible, boxes should implement (or take part in implementing) only one "property" of a program. This allows redistributions to eliminate a box when the single property is not required in context. Having a box implement multiple properties would obviously mean that the context would have to be insensitive to *all* the properties before the box could be eliminated.

This criterion is only intuitive at this point; it is best explained by example. Consider the most well-known implementation of list APPEND:

```
(DEFUN APPEND (L1 L2)
  "Concatenate two lists"
  (IF (NULL L1)
      L2
      (CONS (CAR L1)
            (APPEND (CDR L1) L2))))
```

This source code translates into a recursive program that is definitely inferior—for optimization purposes—to the more exploded structure shown in Figure 1.3 (page 24). The problem is that the subfunctions of (1) copying the input list, (2) finding the last cons-cell of the to-be-altered list, (3) and setting the CDR of the last cons cell are all distributed among the CONS, CAR, CDR, and recursion boxes. Thus, it is very difficult for the optimizer to eliminate any of these from the recursive form of the program. By contrast, all of these are separate and explicit in the preferred structural representation of Figure 1.3. Figure 1.4 (page 25) shows an optimization that eliminates just the copying subfunction; Figure 1.5 (page 26) shows an optimization that eliminates the last-cons subfunction. If I had chosen the recursive structure the effect of the first optimization could have been achieved in a more complicated way requiring more search time; the second optimization could not have been achieved at all.

Note that if context-dependent optimizations are *not* possible, it at first appears that we are stuck with an inefficient implementation: both the copy box and the last-cons box perform essentially the same iteration over the list, where the recursive implementation only iterates once over the list. In this case, however, a different (context-independent) optimization fuses the two iterations. This optimization is illustrated in Figure 3.6 (page 67). Thus, even when APPEND must perform the copy, the functionally-exploded implementation leads to an optimized implementation that is at least as efficient as the recursive version. This is an important observation; if a module can't be automatically optimized from the functionally exploded form to be at least as efficient as other forms; it may not be worthwhile keeping that version in the library.

Other examples of this phenomenon are illustrated in Appendix C. They include data invariant suspension, where a single data invariant is enforced by a single box; copy elimination, as in the example above; and identical-value redistributions.

A question raised by this research concerns the fact that "functionally exploded" does not uniquely define a best form in some cases. There are programs where one form is better for certain uses, another form is better for other uses, and there is no "join" form which can be optimized into both forms. The examples I've come across center around the two ways of representing iteration: via tail-recursion and via series objects. The LR1-REM program is a case in point. Used within the SR-ELT?+UNION example (Sec-

tion C.3.1), the key optimization eliminates the recursion box of `LR1-REM` in its tail-recursive form. In the `LR1-REM+APPEND` example (Section C.2.4), however, the key optimization is a loop fusion of the iteration within the `LR1-REM` box. Of course, the recursion box is not present in the series implementation, so it can't be eliminated, and the series object is not present in the recursive implementation, so it can't be shared. The question raised by this and other such examples is, should multiple structural representations be stored in the library? Or should we try to develop methods for automatically converting between different structural representations? Note that in the case of recursion/series, the *structural* translation is straightforward, but the translation of the explanatory information is not. Automatic conversion methods would appear to have further advantages in optimization power, since they could be executed several times *during* the optimization process, possibly allowing more optimizations than could be obtained simply from choosing one or the other at the beginning.

Chapter 13

Certification

Inasmuch as the system's output represents a set of conjectures, rather than certainties, safe (bug-free) use of the system requires some form of *certification*, or checking that a given redistribution preserves correctness of the program with respect to the top-level specification. There are two places in the development life-cycle that one can install certification: compile-time is while the optimizer is operating, but before the optimized program is run on any new examples; and run-time is during actual use of the optimized program on new examples.

13.1 Compile-time Certification

Since neither IEBR nor IBR requires proof structure to be produced during their operation, the compile-time certifier needs only to decide whether the proposed optimization is safe; it need not produce a proof. Thus, rather than requiring a theorem *prover*, the certifier needs only a theorem *checker*. The difference between checking and proving is significant, because one must believe in the correctness of the theorem checker to believe its answers, whereas one can quickly and independently check the (positive) answers of a theorem prover. The major advantage of requiring only checking for certification is that the system is then insensitive to the particular reasoning used by the system, so long as it is sound; no constraint is placed on the style of proof. This allows more latitude in the design of the reasoner.

Compile-time schemes vary in how much human interaction they require.

As remarked earlier, IEBR, in finding only more routine optimizations, will presumably be more practical to use with a theorem checker than IBR.

- *Fully Automatic.* A fully automatic approach would employ some form of limited automated reasoning system to try to formally verify the correctness of each redistribution. The theorem checker *must* be limited in some way, because there is no guarantee that either a proof or disproof exists for a given redistribution. Thus, an unlimited theorem checker might run forever if given an untrue theorem to check. Moreover, there can be theorems that could be found but only in an impractical length of time. A conservative approach would be to discard a redistribution whenever it can't be certified within the given limits.

- *Interactive Theorem Proving.* A more practical approach, given the current state of the art in theorem checkers, would be to use a human-guided theorem checker. This would both increase the number of theorems checked and maintain the correctness guarantee of the resulting program. The main drawback of this approach, as with all interactive approaches, is that it requires the programmer to think about low-level details.

- *Human Oracle.* Finally, a practical near-term approach is for the system to simply ask the user whether to accept or deny each redistribution, without employing formal proof methods at all. While this may be asking for trouble, it still appears to be a significant step up from current programming practice: the search is systematically organized to examine all the possibilties in a reasonable order, based on likelihood of payoff. Most non-expert programmers do poorly at optimizing their programs simply because they do not know how to organize the search. Frequently, a great deal of effort is put into saving milliseconds out of a much longer run, because the programmer doesn't understand the real bottlenecks. Note that since today's programmers typically think about low-level details anyway, having them do it here is no extra burden.

13.2 Run-time Certification

The notion of *efficient program checking* introduced by Blum and Kannan (1989) provides another means of coping that does not rely on automated theorem proving, yet is still fully automatic. Intuitively, a program checker is a distinct program (assumed correct) that can check the outputs of any program claiming to solve the given problem Note that a set of optimization invariants stated absolutely, without using the `.ORIGINAL` construct, is a program checker; though, for certification purposes I do not restrict program checkers to be in the effective-clause formalism.

My proposal is to check the optimized program's outputs every time it is run, using an *efficient* program checker as defined below; if it is found to be incorrect, then signal the user and offer to re-optimize the program using the inputs for the faulty run as an additional test-case. Of course, the re-optimization can ignore all source/target pairs that were shown incorrect in the previous run. This saves most of the time of re-optimization. The result of the re-optimization will be a program that is more often correct than the original optimized version.

This approach is practical as long as the checking is efficient: the run-times of the optimized program and the checker together must be significantly less than that of the original program alone. Blum and Kannan (1989) develop a theory of efficient program checking related to this idea, but do not apply it to program optimization; hence, their definition of an efficient program checker, as one whose run time is little-omicron of the program's run time, is different than mine.

Note that an efficient checker cannot be *relative*, because it would be forced to run the unoptimized program to get those outputs every time it checked a run of the optimized program. Such a strategy obviously violates the efficiency condition. Note, however, that the *optimizer* can still use a relative optimization invariant during optimization; it is only forbidden at program use time.

It is clear (see Appendix E) that it will not always be possible to find a checker, much less an efficient one, so this approach to certification is not applicable to all programs.

A Hybrid Scheme. It may be useful to combine compile-time and run-time approaches by using a human oracle to check the redistributions as they

are produced, but still do run-time checking.

Note that using run-time checking with the other compile-time schemes is pointless, since proving some of the redistributions correct doesn't reduce the time required to check the outputs at run-time.

An approach that won't work. It might at first appear that with IEBR one can simply add a run-time check to the program that evaluates the target condition every time the production code is run to make sure that the new source is still okay. This is exactly the same type of thing as array bounds checking or run-time type checking. Unfortunately, it won't be practical because target conditions are frequently stated in terms of virtual structural elements which won't be present at production run-time.

Chapter 14

Literature Review

This chapter places this research in relation to previous approaches to the problem of improving the performance of programs. It also discusses relations with other branches of Artificial Intelligence and Software Engineering research.

14.1 Program Improvement and Redesign

Traditional Compiler Techniques. One way to view redistribution of intermediate results is as an attempt at generalizing many traditional compiler techniques to apply to arbitrarily high-level abstractions. Techniques like common subexpression elimination, copy elimination, and loop fusion (Aho, Sethi, & Ullman, 1986) are fundamentally limited by the level of their source languages. That is, they can only exploit the semantics of the data types that are primitive to the language, because they cannot capture and fully exploit the semantics of user-defined types. This stems from having no access to the true specifications of the programs or the domain theory that connects the structure to the specification. For example, a FORTRAN optimizer can only exploit the algebraic laws of integers, arrays, and other low-level types, knowing nothing of the laws of higher-level types such as sets, mappings, graphs, etc.

I include in the category of traditional compiler techniques the operations of type inference and automatic datastructure choice and aggregation performed by the SETL compiler (Freudenberger, Schwarz, & Sharir, 1983).

SETL is a much higher-level language than most, hence the optimizations its compiler performs have greater impact on the performance of programs. Nevertheless, these optimizations are still limited to the semantics of language primitives. The optimizer demonstrates a great deal of ingenuity in determining when certain optimizations regarding sets and mappings may be performed, but the language cannot capture any extra semantic information about higher-level, user-defined abstractions. While it might infer that a particular copy operation on a set is unnecessary, it will not be able to infer the analogous fact about a copy operation on a user-defined type. Since no language will ever predefine anywhere near all of the useful programming abstractions, traditional approaches to optimization will never be free of the "source language tarpit."

Low-level Program Transformation Systems. The program transformation school (Partsch & Steinbruggen, 1983; Cheatham, 1984; Darlington, 1981; Reddy, 1991) takes the view that optimization should take place as a process of *program transformations,* usually at the source code level. Each transformation must provably preserve program correctness. Consequently, each has a set of applicability conditions which must be verified. An as yet unattained goal of the research is that these conditions be checked automatically so that program correctness is guaranteed no matter how the human influences the process.

Fully automatic approaches to choosing the sequence of transformations are not practical ways of producing efficient code, both because the search space is too large and because powerful optimizations require powerful theorem checking capabilities not generally available. Consequently, most transformational approaches are semi-automatic in that a human must guide the selection process. Also, the human must sometimes assist in verifying applicability conditions. This line of research cannot be termed a success as yet, because the process of (a human) guiding the transformations is difficult and tedious. Each transformation is relatively low-level, so many are required to carry out any particular optimization. Optimizing a large program from its clear, but inefficient specification requires too many small-grain steps to be feasible. There is, however, ongoing research into structuring the transformation process; see Fickas (1985) for one approach and Meertens (1986) for many papers on this subject.

A particular branch of this field (Wile, 1981; Scherlis, 1981) investigates

specializing data type implementations to their contexts. While these approaches discuss some of the transformations possible, they again do not discuss the issue of automating the search. In particular, there is no discussion of explanatory structure or the use of correctness information in guiding the search.

It is possible that the search control ideas developed in this research (focus on eliminating a box at a time, in order of estimated cost) may be applicable to program transformation technologies; I have not looked into this connection as yet.

High-level Program Transformation Systems. KIDS (Smith, 1991) is an interactive program transformation system incorporating many powerful transformation tools, such as algorithm designers, an inference system, a finite differencing subsystem, and a partial evaluator. This is qualitatively different from the other program transformation approaches in that the human guidance is in terms of much higher-level operations. This ameliorates the search problems faced by the lower-level transformation systems.

In fact, a redistribution module would, I believe, fit in well in the KIDS environment as another automatic transformation step available, as it can do things the other steps can't. For example, Smith (1991) mentions several shortcomings in the final program output by KIDS for the k-queens problem: it performs unnecessary list member checks and unnecessary copy operations. I have given examples here of how my system can get rid of these things. Furthermore, the KIDS environment could probably be easily adapted to maintain the teleological information needed by the redistribution system, since all the steps are automated.

Similar in philosophy to KIDS is the Programmer's Apprentice (Rich & Waters, 1990). It too contains various types of experts (a designer, a program recognizer, and a requirements assistant) and a general inference system (CAKE). It is in principle capable of taking much higher-level and less precise initial problem statements than KIDS, but is less automated than KIDS. I believe that a redistribution subsystem would fit in to the PA for much the same reasons as it would with KIDS; it provides necessary capabilities not already available and the environment naturally provides most of the extra teleological information as a by-product of the design and analysis processes.

Finite Differencing. Finite differencing (Paige & Koenig, 1982) is a method for improving programs by replacing repeated all-at-once computations with more efficient incremental versions. The implementation in RAPTS (Paige, 1983) is semi-automatic in that a user must decide which instances of differencing to apply and whether an instance is desirable. The system is given a sizable base of specific "differentiation" rules, each of which applies to some pattern of operations expressed in SETL. For example, one such rule says that the expression $\#S$, size of the set S, can be maintained incrementally by (1) initially calculating the size of S, (2) for every addition to S adding 1 to it, and (3) for every deletion subtracting 1.

Though finite differencing is an elegant idea, expressing this idea in terms of a large rule base of highly specific instances has significant problems. The biggest is that to exploit the idea of differencing to its fullest, the system would require new differencing rules for any new abstraction. This is not simply the "standard expert system complaint," however. Typical expert systems solve relatively fixed problems, where the expertise changes only slowly. By contrast, every new user program can potentially have new configurations of function calls that require new differencing rules. Therefore, in order for the rule-based implementation of finite differencing to be considered a complete theory, it must also account for how the rules are (automatically) derived from the definitions of the abstractions. The analog to this problem in my system is the difficulty of supplying appropriate design information as input to the optimizer; by contrast, I *have* given an account of how much of the needed extra knowledge can be automatically supplied (cf. Chapter 10). Other finite differencing problems include difficulties in deciding which rules to apply and whether a given rule will improve the efficiency of the program.

With regard to redistribution, I believe that much of the finite differencing *behavior* can be seen as an application of the redistribution principle. A principled approach to redistribution would, therefore, supply a partial answer to the problem mentioned above. Finite differencing could then be seen as an emergent behavior rooted in deeper principles. Of course, rules are not bad *per se*. Transformation rules compiled automatically from experience can be useful for speeding up a system in which they could otherwise be derived in a principled but slow way.

Memoizing. Mostow & Cohen (1985) have investigated automating the well-known technique of *memoizing*. This is the idea of a subroutine main-

taining a cache of its output values for those inputs for which it has already computed an answer. If the subroutine is ever called more than once on the same input, the answer is looked up in the cache the second and succeeding times, rather than being recomputed. Mostow and Cohen explored the addition of caches to Interlisp functions, with an eye to building a fully automatic tool. Unfortunately, it appears that this problem is too difficult, because side effects and large datastructures make the technique difficult to justify in many cases. I believe the chief problem in this approach is, again, that the memoizer has neither knowledge of nor control over the design process, because, like a compiler, it takes in only the Interlisp source code. It must always assume the worst possible cases of usage for any given subroutine, cases which could possibly be ruled out if extra information relating to purpose and correctness were known. Memoizing can potentially be used to implement equal-value redistributions, but I have not explored this technique.

Tupling. Pettorossi (1984) has defined *tupling* to be the combination of two initially separate functions into a single, vector-valued function in order that their implementations may share partial results. He proposes it as a program transformation, but does not discuss automation of the search. The essence of the technique is interesting in comparison with my approach to redistribution. My system avoids the need for tupling, by viewing the program as "virtually flattened," so that all intermediate results are available to be shared. This is limited only by execution ordering of the boxes and by recursion constraints. The system keeps track of the module boundaries, but is free to add new input and output ports to boxes in order to achieve sharing. I believe tupling can be viewed as a transformation that could help implement redistribution in a general program transformation system.

Automatic Programming Approaches to Efficiency. Automatic programming systems usually implement a top-down refinement approach to program design, always "staying within" abstraction boundaries. This contrasts with my system which is primarily concerned with breaking these boundaries to gain efficiency. Thus, my system is complementary to these systems. The most interesting point of comparison is in the method of deriving and using cost information to guide the search.

Kant's LIBRA system (Kant, 1983) is designed to be a search control ex-

pert which guides the stepwise refinement process of the PSI (Green, 1976) synthesis system. LIBRA guides the search by performing an incremental symbolic analysis of the efficiency of the evolving design, obtaining "optimistic" and "achievable" performance estimates. The refinement is then controlled using branch-and-bound. LIBRA spends most of its time performing algebraic manipulation to simplify *quantitative* cost estimates of the overall program. McCartney's MEDUSA system (McCartney, 1987) designs efficient algorithms in the domain of computational geometry problems. McCartney's approach augments the stepwise refinement paradigm of Kant's approach with domain knowledge of general problem decomposition techniques. MEDUSA uses analytic knowledge in a similar way to LIBRA, i.e. to provide a cost function for use with branch-and-bound search.

I have avoided the complexity of maintaining a quantitative cost estimate of the entire program at each step by deriving *qualititative* cost estimates only of portions (boxes) of the program.

Both LIBRA and MEDUSA would have trouble producing the optimized implementations achievable with redistribution, simply because they operate within abstraction boundaries. It would be interesting to integrate my system with these systems. Moreover, since they perform stepwise refinement, much of the teleological information, including the specifications and module correctness proofs, would be a natural by-product of the software library and the initial design process and could then be used by my system.

Replanning and Designing Extensible Software. Linden (1989) proposes using much the same additional information—specifications and teleological information—to support the evolution and redesign, for a changing specification, of software systems. While not directly relevant to program optimization, this work nevertheless points out other valuable uses of this extra information. This is yet another reason to move toward integrated software development environments that can support the type of automated documentation required for both approaches.

14.2 Explanation-based Generalization

In this research, target conditions are derived from proofs via a novel technique related to explanation-based generalization (DeJong & Mooney, 1986).

Though there are many approaches to EBG, they all take in an explanation of why a given concrete example belongs to a concept and extract from it a generalized, operational definition of concept membership. The technique used is basically the operation of turning constants to variables and back-propagating constraints through operators.

My system contrasts with this first in the fact that the input is an explained *program* while the "concepts" to be learned are conditions on portions of the program. Thus, the explanation is not of why a single concrete example satisfies a single concept, but rather why many different structural elements work together to satisfy a goal. The system then "learns" an approximation to the role of each structural element in the correctness of the design. Each of these roles is a target condition. Thus, each single explanation encodes many concepts.

Another contrast is in the technique of generalizing: my system uses *parent-child clause unioning* (PCCU) in resolution proof trees which is different from standard approaches to EBG. Standard approaches generalize both the concept at the root of the tree and the conditions at the leaves of the tree through a form of variabilization. PCCU does not attempt to generalize the clause at the root of the proof tree, which EBG does to get a proof of concept membership in terms of a free variable, because the proofs it operates on are not direct proofs of concept membership. Rather, the proofs establish global program properties and the PCCU technique infers implicit "concept membership conditions" (target conditions) from these proofs. As such, variabilization is neither appropriate nor desirable. On the other hand, given a particular proof tree, the PCCU technique extracts more general leaf conditions than do standard EBG techniques, and is able to generalize selected proof leaves rather than simply all of the leaves.

Though there is work on using EBG to acquire programming *methods* from experts (Shavlik, 1988, among others), I have not seen any that uses EBG to derive facts about elements of a given program in order to perform optimization.

Chapter 15

Conclusions

The purpose of this research has been to investigate ways in which function sharing—a universal principle of design—may be automatically introduced into software designs. This approach is particularly applicable to the problem of eliminating redundant or unnecessary operations introduced as a natural result of reusing general software modules in specific designs.

The class of optimizations considered in this work, redistribution of intermediate results, restricts the introduction of function sharing to those cases where the adaptation problem is easy. Nevertheless, the range of phenomena covered is broad, including examples of copy elimination, generalized loop fusion, data invariant suspension, and identical-value redistribution (generalized common subexpression elimination).

Key steps in achieving full automation of this process include exploiting extra design information beyond program structure, deriving operational target conditions for screening candidate optimizations, and using representative test inputs to quickly eliminate almost all candidate optimizations. External to this thesis, but nevertheless important to the eventual success of the overall research effort, are the steps required to automate certification of candidate optimizations, whether at compile-time or run-time.

I designed and implemented two different algorithms, IBR and IEBR, to find candidate optimizations (redistributions). IBR is more powerful but appears to be slower, because it re-evaluates all tests for each candidate optimization. IEBR, by contrast, avoids the need to re-evaluate tests by constructing operational target conditions and checking them using the original test trace data. Importantly, IEBR restricts to a class of routine optimizations

which are significantly easier to automatically certify than the arbitrarily clever optimizations found by IBR.

This research contributes both to software engineering and to artificial intelligence, serving both as an initial feasibility demonstration of a novel software engineering technique and as a case study in automating a part of the design process. The success of the implemented system has demonstrated the feasibility of automating the introduction of limited forms of function sharing into programs. The question of whether the approach will ultimately be useful to software engineers cannot, as yet, be answered conclusively. I believe, however, that the potential benefits and likelihood of success justify the further research required to answer it. The secondary question of which of the two algorithms, IBR or IEBR, will ultimately be more useful cannot be answered based on this research; to answer it, we must await a concentrated re-engineering effort. I believe something like an order of magnitude or more in run-time performance can be achieved through re-engineering each algorithm.

15.1 Contributions

- Generate and test approaches to design optimization have traditionally been limited by the need to prove arbitrarily difficult theorems to certify the safety of design changes. This research has taken a first step toward alleviating this problem through the application of machine learning ideas to deriving operational target conditions which supply theorems that are easier to prove than arbitrary safety conditions.

- Function sharing has long been acknowledged as an important principle of design. This work has demonstrated that restricting solutions to the adaptation problem can constrain the search space of possible function sharing optimizations to manageable proportions. I believe this idea carries over to the study of design automation in other engineering domains, though the range of phenomena captured in each domain may be more or less useful.

- The system is based on a new methodology for structuring the program optimization task: first, it finds plausible and useful candidate optimizations, then a certifier checks the candidates for safety. I have

automated the first problem, while separating out the second problem. This screen/certify methodology is more attractive than standard approaches to program optimization, because it allows more powerful hybrid approaches to certification, including (but not limited to) technology based on correctness-preserving transformations, limited theorem proving, and program checking. All current automatic approaches to optimization, by contrast, are fundamentally (and inextensibly) limited in power to a single approach to certification.

- Current program optimizers are stuck in what I call the "source language tarpit." The only specification freedoms available to them are those that follow from the specifications of the language primitives. Such optimizers know nothing about the intended use of the program, so must assume the worst cases of possible usage. By contrast, my system can exploit specification freedoms present at any level of the user-defined abstraction hierarchy. Supplying optimization invariants to the system enables a large, *qualitative* improvement in optimizer performance.

- When considering a large space of interacting optimizations, it is crucial to structure the search by adopting some cost-based heuristic. By ordering the search in this way, the system avoids low-payoff program changes interfering with subsequent high-payoff changes. Even the simple qualitative approach used here is surprisingly effective at this.

- There is a symbiosis between the reuse methodology and powerful optimization tools: each benefits from the presence of the other.

  - The availability of a more powerful optimizer makes reuse of components more practical. This is true for the same reason that standard optimization techniques make the use of high-level languages more practical. The corresponding benefits are of the same kind, but the key difference between standard techniques and my system is that the "language"—the modules in the software library—is variable and arbitrarily high-level, while the optimizer's power is not limited to a fixed set of primitives. The high level of the library modules allows initial programs to be written more quickly and clearly than in a standard programming language.

– A reuse methodology can greatly improve the practicality of IEBR and IBR. This follows from the simple observation that the costs of producing the additional required information can be amortized over the many uses of the library modules. This leads to a two-stage model of the history of a module: it starts out as a user program written in terms of library modules. Its optimization invariants, explanations, and test inputs are produced as quickly and simply as possible using (possibly) quasi-specification proof generation and relative invariants. Then, experiments are performed on the module, debugging and redesigning it. Once the module's design stabilizes and it is deemed potentially useful for other applications, the programmer invests time and effort in improving the invariants and proofs to a standard good enough to enter into the library. The enrichment of the invariants and proofs will mean that subsequent uses of the module will be more highly optimized than the prototype was.

  Note also that if this system is embedded in a larger, machine-mediated design environment, it should be possible to obtain more complete specification and proof information than might be gotten from quasi-specification proof generators. This is because during the initial design there might be other tools that acquire and augment such information. An example would be the Requirements Apprentice (Reubenstein, 1990) which acquires evolving informal requirements.

15.2 Limitations

Limitations of this research exist at many levels of description:

- As an approach to facilitating reuse of software modules, function sharing is limited in that it cannot capture all possible optimizations. Other techniques, such as partial evaluation and finite differencing, will easily capture some optimizations that function sharing will not get at all. Function sharing, however, both captures new optimizations and unifies classes already performed by other methods.

- Maximal function sharing is not well-suited to significantly parallel computations. Forcing all processors to wait for one to compute a shared value may be less efficient than having several different processors compute the value as needed. Redistributions tend to sequentialize the program.

- Redistribution of intermediate results is more limited than function sharing in general in that it introduces no new code into the design, hence useful optimizations will be missed even if only simple additions are required. I expect that this research will serve as a starting point for exploring function sharing with nontrivial adaptation.

- Each of the tradeoffs made for practicality makes the system less powerful. For example, the system can give answers (conjectures) that introduce bugs into the program if the tests have unfortunate coincidences in them—and it is difficult to find good test inputs. Also, the system can miss the best optimizations if eliminating an inexpensive box is required to enable eliminating a costly box. IEBR can miss optimizations if a quasi-specification proof is used. IEBR also restricts attention only to more routine optimizations, trading off power for help with certification.

- Redistribution of intermediate results is sensitive to idiosyncracies of the input program's structure. That is, small changes to a program's structure can have significant effects on optimization results. I believe prospects are good, however, for extending adaptation techniques without harming the basic practicality of the present approach.

15.3 Future Work and Potential Applications

There are two primary directions in which this research can be pursued, each of which holds promise for significant theoretical and practical advances. The two differ in the type of certification technology.

In pursuit of the long-term goal of a completely automatic compile-time optimizer, I believe this work has laid a foundation for solving the certification problem by devising a way of restricting certification only to candidates that

are feasible to prove. To further pursue this goal, the following problems must be addressed:

- The certification of optimizations is a restricted theorem-proving task, where a reasoner might capitalize on such task properties as incrementality and the use of target conditions. Tailoring a theorem prover to exploit these constraints can potentially improve its performance on this task. Furthermore, usable ways of limiting the reasoner must be explored as well. The work reported in (Hall, 1990) may provide a first step along these lines.

- Redistribution optimizations within recursive programs are often thwarted by the base case needing a different redistribution than that needed by the recursive case, such as in the case of `FIB-DESIRED` (see Section C.1.2). It should be relatively simple to extend the theory to handle this simple instance of the adaptation problem. Other cases of non-trivial adaptation should be explored as well, though I don't anticipate a fully automatic solution to the problem of introducing arbitrary function sharing.

- As discussed previously, a fuller understanding of representation issues would improve the system's performance. In addition to studying possible automatic conversion between series and recursive forms of programs, powerful series-like representations should be developed as alternatives to various other (non-linear) forms of recursion, such as tree recursion. Undoubtedly, these would raise semantic issues similar to those pertaining to series (see Appendix F), as well as the issue of how and when to change among the different forms of a program.

- Resolution proof trees and clausal form are particularly simple and difficult to use. Future research should address incorporating more usable proof formalisms. Of course, the design must take into account the needs of the target condition procedure; i.e., the PCCU procedure must be suitably altered.

The less ambitious goal of a system based entirely on run-time (or interactive) certification is not dependent on automatic theorem proving as an enabling technology. Aside from that, however, all of the above issues still

arise. Added to them is the fact that the test case coincidence problem is more crucial, so further research must go into schemes for finding and improving test suites. One possible approach to the testing problem is to locate candidate optimizations whose empirical base, the set of test inputs that exercise that portion of the program, is small, and try to "perturb" those tests somehow to see if they contain an unfortunate coincidence.

In addition to these two research agendas, I believe that the PCCU procedure is interesting in its own right. Further work should clarify its relation to other explanation-based generalization techniques. In particular, does the extra power added by PCCU improve learning performance in some system?

Chapter 16

References

[Aho, A.V.; Sethi, R.; & Ullman, J.D. 1986.] *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley.

[Blum, M. & Kannan, S. 1989.] Designing Programs that Check Their Work. In Proceedings of the 21st Symposium on the Theory of Computation, 86–97. Association for Computing Machinery.

[Boehm, B.W. 1981.] *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall.

[Boyer, R.S., & Moore, J.S. 1988.] *A Computational Logic Handbook.* San Diego, CA: Academic Press.

[Brahms, J. 1880.] *Tragic Overture, Op. 81, bars 229–234.* Excerpted from the *Edition Eulenberg* score, Ernst Eulenberg Ltd, London.

[Cheatham, T.E. 1984.] Reusability Through Program Transformation. *IEEE Transactions on Software Engineering*, SE-19(5):589–595.

[Darlington, J. 1981.] An Experimental Program Transformation and Synthesis System. *Artificial Intelligence* 16:1–46.

[DeJong, G., & Mooney, R. 1986.] Explanation-based Learning: An Alternative View. *Machine Learning* 1:145–176.

[Downey, P.J., Sethi, R., & Tarjan, R.E. 1980.] Variations on the Common Subexpression problem. *J. Association for Computing Machinery, 27(4)*, 758–771.

[Feather, M.S., & London, P.E. 1982.] Implementing Specification Freedoms. *Science of Computer Programming* 2:91–131.

[Fickas, S.F. 1985.] Automating the Transformational Development of Software. *IEEE Transactions on Software Engineering* SE-11(11):1268–1277.

[Freudenberger, S.M., Schwartz, J.T., & Sharir, M. 1983.] Experience with the SETL Optimizer. *ACM Transactions on Programming Languages and Systems,* 5(1):26–45.

[Green, C.C. 1976.] The Design of the PSI Program Synthesis System. In Proceedings of the Second International Conference on Software Engineering, 4–18. Long Beach, CA:Computer Society, IEEE Inc.

[Hall, R.J. 1990.] Parameterizing a Propositional Reasoner: An Empirical Study. *Journal of Automated Reasoning*, **6**, 1990. The Netherlands: Kluwer Academic Publishers.

[Hall, R.J. 1991.] Program Improvement by Automatic Redistribution of Intermediate Results: An Overview. To appear in *Automating Software Design*, M. Lowry & R. McCartney eds. Menlo Park, CA: AAAI Press.

[Hopcroft, J.E., & Ullman, J. D. 1979.] *Introduction to Automata Theory, Languages, and Computation.* Reading, MA: Addison-Wesley Publishers.

[Kant, E. 1983.] On the Efficient Synthesis of Efficient Programs. *Artificial Intelligence* 20:253–306.

[Le Metayer, D. 1988.] ACE: An Automatic Complexity Evaluator. *ACM Trans. on Programming Languages and Systems, 10(2),* 248–266.

[Linden, T. 1989.] Representing Software Designs as Partially Developed Plans. In *Automating Software Design*, eds. M. Lowry & R. McCartney, Palo Alto, CA: AAAI Press.

[Mason, I. 1986.] *The Semantics of Destructive Lisp.* Chicago, IL: University of Chicago Press.

[McCartney, R.D. 1987.] Synthesizing Algorithms with Performance Constraints. In Proceedings of the Sixth National Conference on Artificial Intelligence, 149–154. Los Altos, CA: American Association for Artificial Intelligence (Morgan-Kaufmann).

[Meertens, L.G.L.T. ed. 1986.] *Program Specification and Transformation: Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation.* Amsterdam: North-Holland.

[Mitchell, T.M., Keller, R.M., & Kedar-Cabelli, S.T. 1986.] Explanation-based generalization: a unifying view. *Machine Learning, 1*, 47–80. Amsterdam: Kluwer Academic Publishers.

[Mostow, J., & Cohen, D. 1985.] Automating Program Speedup by Deciding What to Cache. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, 165–172. Menlo Park, CA: International Joint Conferences on Artificial Intelligence.

[Paige, R. 1983.] Transformational Programming—Applications to Algorithms and Systems. In Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages, 73–87. New York, NY: ACM Press.

[Paige, R. & Koenig, S. 1982.] Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems* 4(3):402–454.

[Partsch, H., & Steinbruggen, T. 1983.] Program Transformation Systems. *ACM Computing Surveys* 15(3):199–236.

[Pettorossi, A. 1984.] A Powerful Strategy for Deriving Efficient Programs by Transformation. In Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, pp 273–281. New York, NY: ACM Press.

[Reddy, U. 1991.] Design Principles for an Interactive Program Derivation System. In *Automating Software Design*, eds. M. Lowry & R. McCartney. Palo Alto, CA: AAAI Press.

[Reubenstein, H.B. 1990] *Automated Acquisition of Evolving Informal Descriptions,* Technical Report, AI-TR-1205. M.I.T. Artificial Intelligence Laboratory.

[Rich, C. & Waters, R. 1990.] *The Programmer's Apprentice.* New York, NY: ACM Press.

[Scherlis, W.L. 1981.] Program Improvement by Internal Specialization. In Proceedings of the Eighth ACM Symposium on Principles of Programming Languages, 41–49. New York, NY: ACM Press.

[Shavlik, J. 1988.] An Approach to Acquiring Algorithms by Observing Expert Behavior. In Proceedings of the AAAI-88 Workshop on Automating Software Design. Palo Alto, CA: Kestrel Institute.

[Smith, D.R. 1991.] KIDS—A Knowledge-based Software Development System. In *Automating Software Design*, eds. M. Lowry & R. McCartney. Palo Alto, CA: AAAI Press.

[Steele, G. 1990.] *Common Lisp: the Language, 2nd Edition.* Digital Press, Digital Equipment Corporation.

[Stoy, J. 1977.] *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* Cambridge, MA: MIT Press.

[Ulrich, K. 1988.] *Computation and Pre-parametric Design,* Technical Report, AI-TR-1043. M.I.T. Artificial Intelligence Laboratory.

[Waters, R.C. 1979.] A Method for Analyzing Loop Programs. *IEEE Trans. on Software Engineering, SE-5(3),* 237–247.

[Wile, D. 1981.] Type Transformations. *IEEE Transactions on Software Engineering*, SE-7(1):32–39.

Part IV

Appendices

Appendix A

Experimental Domain Knowledge

This appendix defines the domain knowledge used for experiments, and the next appendix is a glossary of the function and program names defined as part of the domain knowledge.

I hand coded the domain knowledge for the system. Note that the system can be applied to other machine models and sets of primitives. I made these particular choices, because they model a simplified subset of destructive Lisp, a domain with which I am familiar. A comprehensive formalization (semantics) of destructive Lisp is available in (Mason, 1986); to apply my system to a "real" Lisp environment, one should probably start from something like Mason's formalism.

A.1 Primitive Data Types

The primitive, atomic data types in my system are all mutually disjoint. Here are brief descriptions of each. Note that because of the recursive nature of Lisp's types, the descriptions will corefer and will also refer to the compound types (defined subsequently).

- `N`: *Nonnegative integers.*
- `Bool`: *Booleans.* `true` and `false`.

- T: *Tokens.* Similar to Lisp symbols, these have names and nothing else. NIL is the most useful member of this type.

- NIL: the subtype of T whose only element is the token NIL. This is an overloading of the name NIL, which denotes both a type and an element of type T.

- U: *Undefined* or "error" object. This type contains the distinguished constant, _u_. This type is disjoint from every other predefined type and is not included in type O.

- C: *Mutable memory cells.* Each memory cell has a positive number of fields, numbered $0, 1, \ldots, k-1$, with k denoted by CSIZE(c). Each cell c can be bound (in a store) to a tuple of values of length CSIZE(c), where the values can be any element of type O (see below), the union of all data-like types. There are infinitely many memory cells of each CSIZE.

- S: *Stores.* Each store maps finitely many memory cells to objects of type P, which are tuples (of compatible size) of objects of type O. The rest of the cells are mapped to type U.

- Z: *Series.* (Not necessarily finite) sequences of values of type O.

- P: Finite sequences (tuples) of objects of type O.

For convenience, I have defined the following compound (union) types:

- A: (atoms) union of N and Bool and T.

- O: (data-like objects) union of A and C.

Other types I have defined include L, the type of all finite abstract lists of type-O objects, and Set, the type of all finite sets of type-O objects.[1]

[1]The type Set should not be confused with the computational primitive function SET. Context and typography should adequately disambiguate the two.

A.2 Computational Primitives

Computational primitives are the leaves of the program structure hierarchy. They are subroutines whose implementations are built in by the domain knowledge provider and thereby define the virtual machine architecture on which the user's programs are built.

First, each of the types `N`, `Bool`, `T`, `U`, and `C` has an associated computational type recognizer whose name is formed by prepending `CHI-`. Each of these has an assigned cost of $(1,0)$.

Next, there are several computational constants defined: `ZERO`, `ONE`, `TWO`, etc., are the standard type `N` constants. `TRUE` and `FALSE` are the `Bool` constants. `NIL` is a type `T` constant used in Lisp (and here) to denote the empty list. Contrary to Lisp, I will not use it to denote boolean falsity. `_u_` is the error constant, of type `U`. All constants are assigned cost $(0,0)$.[2]

Memory cell manipulation is performed by the three computational primitives `SEL`, `SET`, and `NEW`. `NEW` allocates a memory cell, `SET` sets a field of a memory cell, and `SEL` accesses a field. See their glossary entries for further detail.

Primitive equality is available in the computational function `EQ?`. This tells whether any two elements of type `O` are identical. It has cost $(1,0)$.

Finally, some standard integer (type `N`) functions are provided as well. `1+` increments, `-1` decrements, `+` adds, `-` subtracts, `*` multiplies, `DIV2` divides by 2 (integer division), and `<` decides the less-than relation. Each of these has cost $(1,0)$.

Series objects are described briefly in Section 3.2.2. Descriptions of series primitives are available in the glossary (Appendix B). Unless otherwise specified, each series primitive has cost $(2,0)$.

A.3 Noncomputational Functions

To state and prove facts about programs written in terms of the computational primitives discussed above, it is necessary to introduce logical functions (including predicates). Each of these is endowed with a primitive term evaluator (as distinct from a computational primitive program; see Section 4.1.3).

[2]Note that constants can be the sources of redistributions, so must be treated as explicit boxes.

Many logical functions (such as 1+, and SEL) have corresponding computational primitives that implement them. Term evaluators for such functions are identical to their corresponding primitive programs, and I will use one name to denote both, since no confusion can arise.

Most basic of the noncomputational functions are characteristic predicates for types without computational predicates: CHI-S, CHI-Z, and CHI-P. Note that the union types A and O have type predicates that can be coded in terms of given primitives.

Equality (=) is not entirely covered by the EQ? primitive, so there is a term evaluator for judging equality of such things as stores and series.

Next, we have functions for reasoning about memory cells and stores. V takes a cell c and a store s and returns either _u_ if c is unallocated in s, or the tuple (P element) to which c is bound. A? takes either a P or a U and returns true if and only if the input is a P. This predicate is used for telling whether a cell is allocated in a given store, via the cliche (A? (V c s)). See also NTHPTR and PTRPOS in the glossary.

A.4 Universal Instantiators

Recall (Section 4.1.3) that the system evaluates universally quantified clauses by having "instantiation experts" recognize the form of the (implicitly bounded) quantification and return lists of data values to substitute for the variables. Here are the univeral instantiation experts given to the system.

- *Allocated-cells.* This notices a term in a clause of the form (NOT (A? (V ?C s))) and instantiates the variable ?C once for each cell allocated in the situation to which s evaluates. If more than one term of the clause matches the pattern, the one whose corresponding situation has the fewest allocated cells is chosen.

- *List-members.* This notices a term in the clause of the form (NOT (MEMBER? ?X l)) and instantiates ?X once for each member of the list value of l.

A.5 Abstract Data Type Implementations

I have performed experiments on two simple data type implementations. LR1 lists[3] are representations of abstract lists in terms of memory-cell and store pairs. SR sets are representations of finite sets in terms of abstract lists. Each of these is defined by a pair of functions. Note that neither abstraction function can be a computational primitive simply because their ranges are not computational types. The relations among these types is shown in Figure A.1.

A.5.1 LR1 Lists

Mathematically, the abstract type here is `L`, finite lists of objects of type `O`. Note that this is not the same as the notion of list in Lisp, because Lisp's lists may have lists as elements. The crucial difference is that LR1 lists are compared for equality by using **`EQ?`** on corresponding elements. Lisp lists are compared recursively by **`EQUAL`**.

LR1 lists get their name (List Representation 1) from the fact that they are an approach to representing abstract lists (LR2 is another approach not reported here). Thus, they are defined by two functions.

`LR1?` is a characteristic predicate taking a cell c (or `NIL`) and a store s and returning true if and only if the pair legally represents an abstract list according to the encoding of LR1 lists. Operationally and more precisely, this is exactly equivalent to `(NOT (CHI-U (PTRPOS (ONE) (NIL)` c s`)))`; i.e., a pair represents an LR1 list precisely when it is `NIL`-terminated via the `(ONE)` field. Thus, infinite length lists are not represented by LR1 lists.

`LR1` (no question mark) is the abstraction function that maps a cell c (or `NIL`) and a store s into the abstract list represented by the pair. (`LR1` is only defined when `LR1?`(c, s) holds.) `LR1` is recursively defined as follows:

- `LR1(NIL,` s) is the empty list for any s.
- A cell c whose `ZERO` field is bound to x in s and whose `ONE` field is bound to y in s (and such that `LR1?`(c, s) is true) is mapped to the list whose first element is x and whose tail is `LR1`(y, s).

[3]My use of "LR1" is completely unrelated to LR(k) parsers. It stands for List Representation number 1.

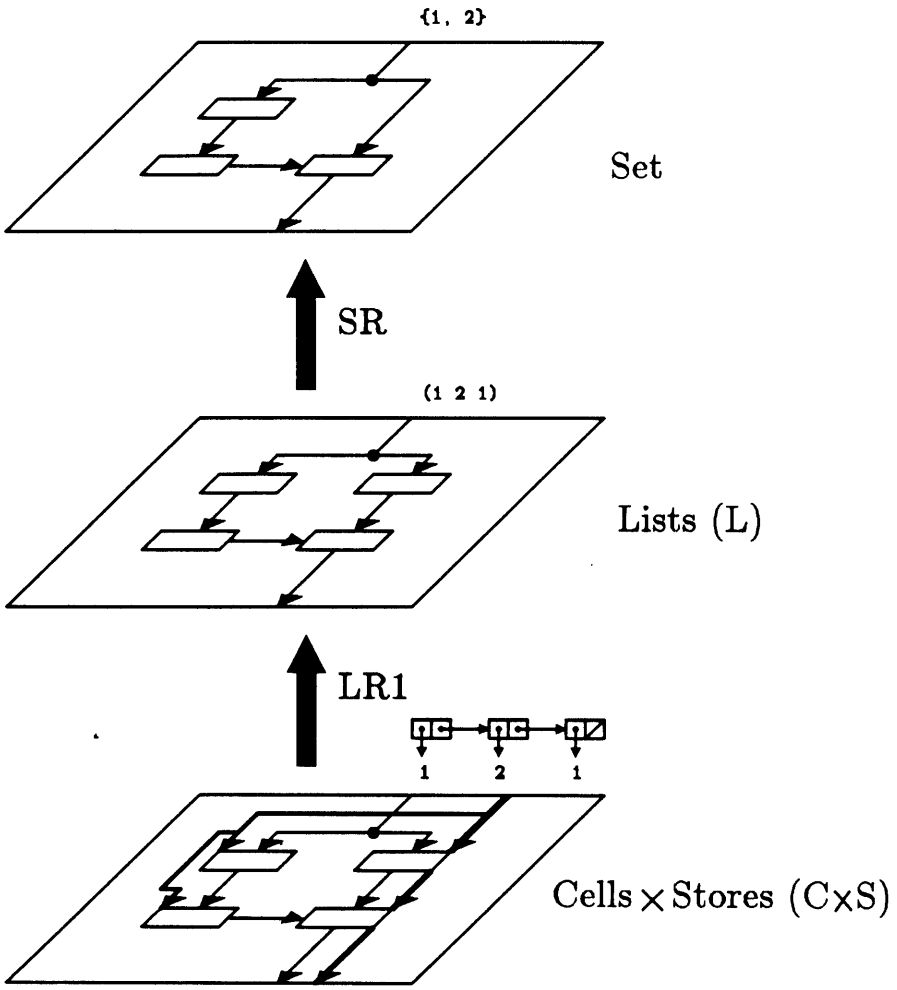


Figure A.1: A diagram illustrating the relationship among the abstract data type implementations studied. Even though the abstraction functions operate on data objects, there is a sense in which they operate on programs as well: any program on cells and stores having lr1-specs (see Chapter 10), corresponds canonically to a program on lists where each box has only its lr1-specs and none others. Similar remarks hold for the sr transformation.

A.5.2 SR Sets

The abstraction function SR maps an abstract list into an abstract finite set via the usual forgetful functor.[4] The characteristic predicate SR? recognizes exactly those lists mappable to a finite set, those with finitely many elements. (Thus, LR1?(c, s) implies SR?(LR1(c, s)).) Note that SR and SR? are oblivious to whether the list has duplicate entries; the no-duplicates invariant on lists is a concept used in implementing set operations efficiently, but is not required for mapping to a set. This is a key observation that allows suspension of the invariant.

[4]... that is, by "forgetting" the ordering information and the number of occurrences of each element.

Appendix B

Glossary

This appendix lists alphabetically all names of functions, programs, computational primitives, etc., used in the experimental system as well as in the examples in the rest of the document. Each name is accompanied by a brief description and cross-reference if necessary. Immediately following the name (in boldface), is a descriptive term distinguishing its type. "Primitive" describes the name of a computational primitive program, a leaf of the program structure hierarchy. "Function" describes a mathematical function defined in the logic. All primitives correspond with functions, so "primitive" implies "function." For multi-output primitives, multiple functions are defined corresponding to the output port names given in the signature. "Program" describes a program name in the structure hierarchy. "Program" alone implies non-primitive, that is, it is a program that has a dataflow diagram definition in terms of other programs and primitives. Each program, primitive, and function has an associated signature of the form

$$(i_1 : t_{i_1}, i_2 : t_{i_2}, \ldots, i_k : t_{i_k}) \rightarrow (o_1 : t_{o_1}, o_2 : t_{o_2}, \ldots, o_k : t_{o_k})$$

that indicates the port names and types of its inputs and outputs.

- **-**: primitive. **(N1: N, N2: N)** $\rightarrow$ **(-: N** $\cup$ **U)**.
  Subtracts two numbers. _u_ if difference less than zero.

- **-1**: primitive. **(N: N)** $\rightarrow$ **(-1: (N** $\cup$ **U))**.
  Subtracts one, except when input is zero, then returns _u_.

- `+`: primitive. `(N1: N, N2: N)` → `(+: N)`.
  Adds two numbers.

- `1+`: primitive. `(N: N)` → `(1+: N)`.
  Subtracts one, except when input is zero, then returns _u_.

- `*`: primitive. `(N1: N, N2: N)` → `(*: N)`.
  Multiplies two numbers.

- `#MSBIFEQ`: synonym for `MSBIFEQ`.

- `=`: function. `(X1: <universe>, X2: <universe>)` → `(=: Bool)`.
  Standard equality relation on the entire universe. Available in logic on entire universe. Computational primitive available on `O` is called `EQ?`.

- `A?`: function. `(P: (P ∪ U))` → `(a?: Bool)`.
  Takes either a `P` or a `U` and returns true if and only if the input is a `P`. This predicate is used for telling whether a cell is allocated in a given store, via the cliche `(A? (V c s))`.

- `APPEND`: synonym for `LR1-CONCAT-C+D`.

- `APURE?`: function. `(S1: S, S2: S)` → `(apure?: Bool)`.
  True iff every cell defined in `S1` is defined in `S2` and also is mapped to the same tuple. Note that `S2` may have more cells allocated, however.

- `CAR`: synonym for `LR1-HDL`.

- `CATENATE`: primitive. `(Z1: Z, Z2: Z)` → `(catenate: Z)`.
  Concatenates two input series, returning a series.

- `CDR`: synonym for `LR1-TLL`.

- `CHI-`*t*: primitive(s). `(O: O)` → `(CHI-`*t*`: Bool)`.
  Type recognizer for some primitive type. Allowed values of *t* are `N`, `Bool`, `T`, `C`, `U`. These are a subset of the following type predicates.

- `CHI-`*t*: function(s). `(O: O)` → `(CHI-`*t*`: Bool)`.
  Type recognizer for some type. Allowed values of *t* are anything. Most are non-computational, but appear in the logic. Examples include `CHI-S`, `CHI-Z`, `CHI-P`.

- CHOOSE: primitive. (ZBOOL: Z, Z: Z) → (choose: Z).
  Takes in a series of Bools and a series of other objects and returns a series of those objects of the second input in the same positions as TRUE in the first.

- COLLECT: synonym for LR1-COLLECT.

- COLLECT-AND?: primitive. (Z: Z) → (collect-and?: Bool).
  Returns logical and of bools in input series. (At the implementation level, it terminates the iteration as soon as the output is determined, possibly before the entire input is seen.)

- COLLECT-LAST primitive. (Z: Z) → (collect-last: O).
  Returns the last element of its input series.

- COLLECT-OR?: primitive. (Z: Z) → (collect-or?: Bool).
  Returns logical or of bools in input series. (At the implementation level, it terminates the iteration as soon as the output is determined.)

- COLLECT-SETCDRS: synonym for COLLECT-SETNEXT.

- COLLECT-SETNEXT: primitive.
  (Z: Z, S: S) → (collect-setnext-c: (CU NIL), collect-setnext-s: S).
  Takes a series of memory cells and sets each ONE field to point to the next cell in line, with the last pointing to NIL. This has the effect of assembling a list out of the series of cells. (Section 3.2.2 called this COLLECT-SETCDRS.)

- COLLECT-SUM: primitive. (Z: Z) → (collect-sum: N).
  Returns the sum of the input series' (type N) elements.

- CONS: program. (A: O, D: O; ENV: S) → (CONS-C: C, CONS-S: S).
  Puts out a fresh size-2 memory cell whose 0 field ("CAR" or "DATA" field) is bound to A and whose 1 field ("CDR" or "NEXT" field) is bound to D. APURE?. Structure is shown within diagram (below) for LR1-PPL (page 205).

- COPY-LIST: synonym for LR1-COPY.

- **COTRUNCATE**: primitive.
  (Z1: Z, Z2: Z) → (cotruncate-z1, cotruncate-z2: Z).
  Takes two series and returns the same two series, except the longer of the two is truncated to the same length as the shorter.

- **CSIZE**: primitive. **(C: C) → (csize: N)**.
  Gives number of fields in a memory cell.

- **DATA**: program. **(C: C, ENV: S) → (data: O)**.
  Selects the zero field (**CAR**) of the memory cell. Structure shown (below) in diagram for **LR1-HDL** (page 203).

- **EQ?**: primitive. **(O1: O, O2: O) → (eq?: Bool)**.
  Tells whether two elements of type **O** are identical.

- **FIB, FIB-DESIRED**: programs. **(N: N) → (fib: N)**.
  Two different implementations of the Fibonacci function. Structure appears in Section C.1.2.

- **FSERIES**: primitive. **(O: O) → (fseries: Z)**.
  This takes in a single type **O** object and makes a length-one series out of it. This has cost $(1,0)$.

- **LAST**: synonym for **LAST-CONS**.

- **LAST-CONS**: program. **(C: (C ∪ NIL), S: S) → (last-cons: (C ∪ NIL))**.
  Returns the unique memory cell accessible from input cell by **ONE** fields whose **ONE** field is itself **NIL**. Same as Lisp's **LAST**. Returns **NIL** if input is **NIL**.

- **LIST**: shorthand for structural cliche of calling **LR1-PPL** with its second argument **LR1-EL**. Makes a length-one LR1 list out of its only argument. Used in **MY-REVERSE** definition for familiarity.

- **LR1**: function. **(C: (C ∪ NIL), S: S)**
  → (lr1: L). This function maps a cell (or **NIL**) and a store into the abstract list it represents. (**NIL**, s) represents the empty list for any s. A cell c whose **ZERO** field is bound to x in s and whose **ONE** field is bound to y in s and who satisfies **LR1?**(c, s) is mapped to the list whose first element is x and whose tail is **LR1**(y, s).

- LR1?: function. (C: (C ∪ NIL), S: S) → (LR1?: Bool).
  This is a predicate taking a cell c (or NIL) and a store s and returning true if and only if the pair legally represents an abstract list according to the encoding of LR1 lists. Operationally and more precisely, this is exactly equivalent to (NOT (CHI-U (PTRPOS (ONE) (NIL) c s))); i.e., a pair represents an LR1 list precisely when it is NIL-terminated via the ONE field. Thus, infinite length lists are not represented by LR1 lists.

- LR1-COLLECT: program.
  (Z: Z, S: S) → (collect-ac: (C ∪ NIL), collect-s: S).
  Returns fresh list whose elements are those of the input series.

- LR1-COLLECT-DATA: program.
  (Z: Z, S: S) → (lr1-collect-data-ac: (C ∪ NIL), collect-data-s: S).
  Returns fresh list whose elements are the DATA fields (CAR fields) of cells in the input series. Implemented in terms of ZCOPY-CELL as

  $$\text{COLLECT-SETNEXT}(\text{ZCOPY-CELL}(\text{SCAN-NEXTS}(z)))$$

  with store flow hooked up appropriately.

- LR1-CONCAT-C+D: program.
  (ACL1: (C ∪ NIL), ACL2: (C ∪ NIL), S: S) →
  (lr1-concat-ac: (C ∪ NIL), lr1-concat-s: S).
  Exactly Lisp APPEND viewed as a function on abstract lists. Does not destroy either input argument. Implementation given as that of APPEND in Introduction. Structure appears in Section 1.3.

- LR1-COPY: program.
  (C: (C ∪ NIL), S: S) → (lr1-copy-ac: (C ∪ NIL), lr1-copy-s: S).
  Makes a structurally fresh copy of the input LR1 list, without modifying any existing structure. Same as Lisp's COPY-LIST. Structure is shown in Figure 3.4 under the synonym COPY-LIST.

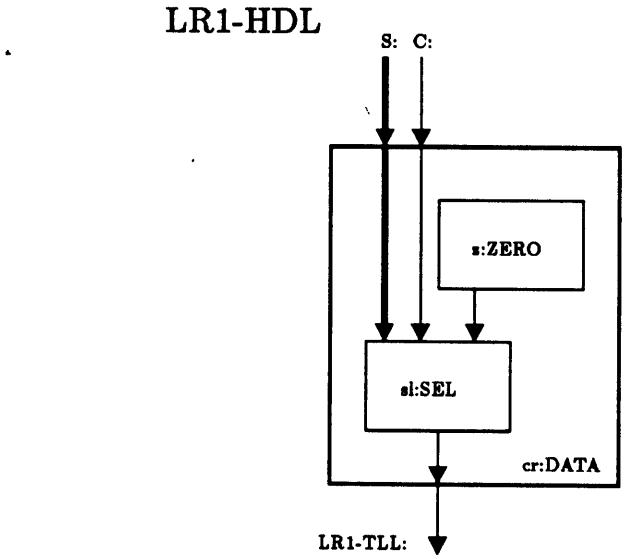
- LR1-EMPTY?: program. (AC: 0) → (lr1-empty?: Bool).
  Returns true iff input is NIL. Satisfies LR1 spec of recognizing the empty list.

- LR1-EL: program. () → (lr1-el: NIL).
  Constant empty LR1 list. This is an "LR1" wrapper around the constant NIL function to provide lr1-specs.

- LR1-EQUAL?: program.
  (L1: (C ∪ NIL), L2: (C ∪ NIL); ENV: S) → (lr1-equal?: BOOL).
  Determines whether L1 is isomorphic (element-wise identity) to L2. Structure appears in Section C.2.6.

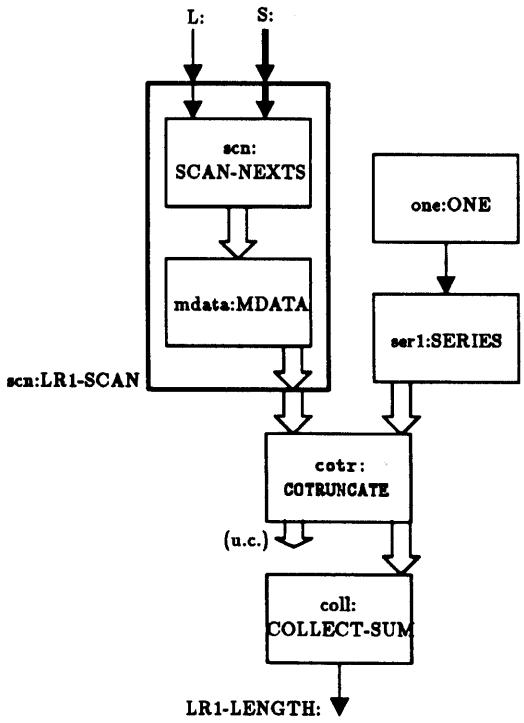
- LR1-FIRSTN: program.
  (L: (C ∪ NIL), N: N, ENV: S) → (l-out: (C ∪ NIL), env-out: S).
  Puts out a fresh list consisting of the first N elements of the input list. APURE?. Structure appears in Section C.2.5, within the diagram for LR1-SPLIT.

- LR1-HDL: program. (C: C, S: S) → (lr1-hdl: O).
  Returns ZERO field contents of c in s. Has LR1 list spec of performing *hdl* function—head left, or leftmost element—on the abstracted list input. Similar to Lisp CAR primitive when viewing CAR as a function on abstract lists. Here is the structure:



- LR1-LENGTH: program. (L: (C ∪ NIL), ENV: S) → (lr1-length: N).
  Gives length of LR1 list L. Here is the structure:

LR1-LENGTH



- `LR1-MEMBER?`: program.
  `(X: 0; L: (C ∪ NIL); ENV: S) → (lr1-member?: BOOL)`.
  Determines whether `X` is identical to some element of the `LR1` list `L`. Program cost is $(2, 0)$.

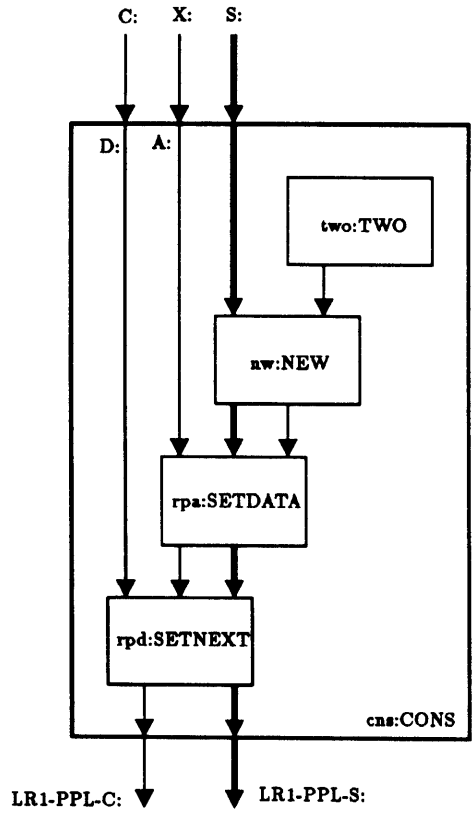
- `LR1-MSORT`: program.
  `(L: (C ∪ NIL), ENV: S) → (lr1-msort-ac: (C ∪ NIL), lr1-msort-env: S)`.
  Sorts the input list of numbers (undefined if list contains nonnumbers) using the MERGE-SORT algorithm. `LR1-MSORT-REC` is a recursive helper for the top-level `LR1-MSORT` program. `APURE?`. Structure appears in Section C.2.5.

- `LR1-NTH`: program. `(L: (C ∪ NIL), N: N, ENV: S) → (lr1-nth: 0)`.
  Puts out the `N`th element of the list `L`. First element (`LR1-HDL`) is the 0th, etc. Structure is shown in Section C.1.1.

- LR1-NTHTLL: program.
  (L: (C ∪ NIL), N: N, ENV: S) → (l-out: (C ∪ NIL)).
  Puts out a fresh list consisting of all but the first N elements of the input list. Structure appears in Section C.2.5, within the diagram for LR1-SPLIT.

- LR1-PPL: program.
  (X: O, C: (C ∪ NIL), S: S) → (lr1-ppl-c: C, lr1-ppl-s: S).
  PrePend-Left; that is, make an LR1 list whose tail is the abstraction of c and whose head is x. Exactly like viewing Lisp's CONS as a function on abstract lists. Here is its program structure:

LR1-PPL



- LR1-REM: program.
  (O: O, C: (C∪NIL), ENV: S) → (lr1-rem-ac: O, lr1-rem-env: S).

Removes O from the input list and returns a fresh output list without modifying any pre-existing structure. Series form structure appears in Section C.2.4; recursive form structure appears in Section C.3.1.

- LR1-REM+APPEND: program.
  (ACX: O; ACL1: (C ∪ NIL), ACL2: (C ∪ NIL); ENV: S) →
  (lr1-rem+append-ac: (C ∪ NIL), lr1-rem+append-env: S).
  Structure appears in Section C.2.4. Removes ACX from the list ACL1 and appends the list ACL2.

- LR1-REM+REVAPPEND: program.
  (ACX: O, ACL1: (C ∪ NIL), ACL2: (C ∪ NIL), ENV: S) →
  (lr1-rem+revappend-ac: (C ∪ NIL), lr1-rem+revappend-env: S).
  Structure appears in Section C.2.3. Removes ACX from the list ACL1, reverses the result, and appends the list ACL2.

- LR1-REV: program.
  (L: (C∪NIL), ENV: S) → (lr1-rev-ac: (C∪NIL), lr1-rev-env: S).
  Reverses its input list. Structure appears in Section 1.3 in different nomenclature as the MY-REVERSE program.

- LR1-SCAN: program. (C: (C ∪ NIL), S: S) → (scan: Z).
  Returns elements of LR1 list as elements of a series. Structure is shown (above) in the diagram for LR1-LENGTH.

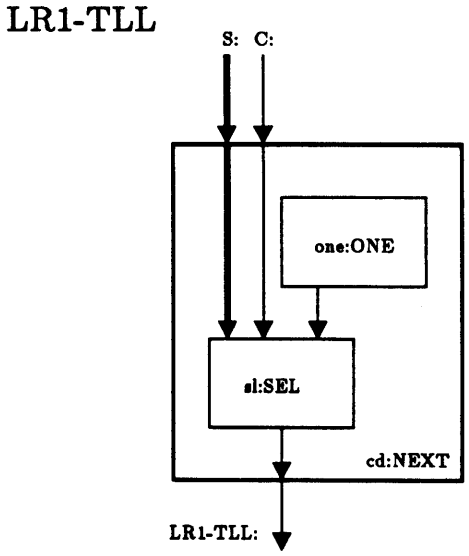
- LR1-SPLIT: program.
  (L: (C ∪ NIL), N: N, ENV: S) →
  (lft: (C ∪ NIL), rgt: (C ∪ NIL), split-env: S).
  Puts out two lists: lft contains the first N (in order), and rgt contains the rest. APURE?. Structure appears in Section C.2.5.

- LR1-SUBSTITUTE: program.
  (OLD: O, NEW: O; L: (C ∪ NIL), ENV: S) →
  (lr1-substitute-ac: (C ∪ NIL), lr1-substitute-env: S).
  Subsitutes NEW for OLD in L. APURE?. Structure appears in Section C.2.6.

- LR1-SUBSTITUTE+EQUAL?: program.
  (OLD: O, NEW: O; L1: (C ∪ NIL), L2: (C ∪ NIL); ENV: S) →
  (lr1-substitute+equal?: BOOL, env-out: S).

Substitutes NEW for OLD in L1, then determines whether the result is isomorphic (element-wise identity) to L2. APURE?. Structure appears in Section C.2.6.

- LR1-TLL: program. (C: C, S: S) → (lr1-tll: (C ∪ NIL)).
  Returns ONE field contents of c in s. Has LR1 list spec of performing *tll* function—tail left, or list without leftmost element—on the abstracted list input. Similar to Lisp CDR primitive when viewing CDR as a function on abstract lists. Here is the structure:



- MARKER: primitive. () → (marker: Z).
  Puts out a length-one series whose element is a marker token distinct from all type T tokens in use by the user. (This is related to the Lisp function GENSYM, but puts out a length-one series.) This has cost $(0, 0)$.

- MDATA: primitive. (Z: Z, S: S) → (mdata: Z).
  Takes in a series and a store s and maps SEL($\cdot$, ZERO, s) over the series (presumably of memory cells).

- MEQ?: primitive. (Z1, Z2: Z) → (meq?: Z).
  Maps EQ? over its two input series in step.

- **MERGE-SORTED-LISTS**: program.
  (L1, L2: (C ∪ NIL), ENV: S) → (L-OUT: (C ∪ NIL), ENV-OUT: S).
  Structure appears in Section C.2.5. Takes in two sorted lists and constructs the sorted merge of them. **APURE?**.

- **MNOT?**: primitive. **(ZBOOL: Z) → (mnot?: Z).**
  Maps logical negation over a series of **Bools**.

- **MSBIFEQ**: primitive. **(NEW: O, OLD: O; Z: Z) → (msbifeq: Z).**
  Takes a series, a test object, and a substitution object and copies the input series except that occurrences of the test object are replaced by the substitution object.

- **MSNGLTN**: primitive.
  (Z: Z, S: S) → (msngltn-z: Z, msngltn-s: S).
  Takes in a series and a store and returns a series of cells and a new store, with each of the cells newly allocated and with **ZERO** field bound to the corresponding element of the input series and with **ONE** field bound to **NIL**. Note that this has cost $(2, 2)$.

- **MY-EXPT**: program. **(BASE: N, POWER: N) → (my-expt: N).**
  Structure appears in Section C.1.1. Exponentiates a number to a power.

- **MY-REVERSE**: program. Synonym for **LR1-REV**.

- **NEW**: primitive. **(N: N, S: S) → (new-c: C, new-s: S).**
  Output **NEW-C**(n, s) is a memory cell of **CSIZE** n that was unallocated in s (i.e., it was bound to _u_). Output **NEW-S**(n, s) is a store in which (a) all cells except **NEW-C**(n, s) have the same binding as in s, and (b) **NEW-C**$(n.s)$ is no longer bound to _u_. (Its binding is undefined, in that applying **SEL** to it is undefined—but not _u_, i.e. not erroneous.) This is the primitive memory allocation function, and is assigned a cost of $(1, 1)$.

- **NEXT**: program. **(C: C, ENV: S) → (next: O).**
  Selects the one field (**CDR**) of the memory cell. Structure shown (above) in the diagram for **LR1-TLL**.

- NTHPTR: function. (N: N, F: N; C: C; S: S) → (nthptr: (O ∪ U)). Returns the nth iterate of SEL(f, c, s), the result of following the f field pointer n times. If it runs out of pointers before the nth, it returns _u_. Note that NTHPTR$(0, f, c, s) = c$.

- NULL: synonym for LR1-EMPTY?.

- ONE: primitive. () → (one: N). Number one constant.

- POLY: program (X: N, COEFFS: (C ∪ NIL), S: S) → (poly: N). Structure appears in Section C.1.1. Evaluates a polynomial (specified by an input list of coefficients in increasing order of exponent) on a number.

- POLY-REC: program. (I, LENGTH, X, SUM: N; COEFFS: (C∪NIL); S: S) → (poly-rec: N). Structure appears in Section C.1.1. POLY-REC is a recursive helper to POLY.

- PTRPOS: function. (F: N, V: O, C: C, S: S) → (ptrpos: (N ∪ U)). Returns the least nonnegative integer p such that NTHPTR$(p, f, c, s) = v$, if it exists. Otherwise, returns _u_. For example, the usual way to state that a list, connected through the ONE field, is NIL-terminated is by (NOT (CHI-U (PTRPOS (ONE) (NIL) c s))).

- RPLACD: synonym for SETNEXT.

- SCAN: synonym for LR1-SCAN.

- SCAN-NEXTS: primitive. (C: (C ∪ NIL), S: S) → (scan-nexts: Z). (This is called "scan-sublists" in (Steele, 1990)). and SCANCDRS in Section 3.2.2.) Makes a series out of the cons cells making up the input list. If the input is NIL then output is the empty series.

- SCANCDRS: synonym for SCAN-NEXTS.

- SEL: primitive. (N: N, C: C, S: S) → (sel: (O ∪ U)). Returns either the object bound to field n of c in s or else _u_ if either c is unallocated in s or is of CSIZE smaller than $n + 1$. This is assigned a cost of $(1, 0)$.

- **SERIES**: primitive. (O: O) → (series: Z).
  This takes in a single type O object and makes an infinitely repeating series out of it. This has cost $(1,0)$.

- **SET**: primitive. (N: N, C: C, O: O, S: S) → (set: (S ∪ U)).
  If c is allocated in s and is of size at least $n+1$ then it returns a store with every field of every allocated cell bound the same as in s except field n of c, which is now bound to o. All unallocated cells are untouched. This is assigned a cost of $(1,0)$.

- **SETNEXT**: program. (C: C, O: O, S: S) → (set: (S ∪ U)).
  Exactly SET with first argument bound to ONE, a.k.a. the "next" field. Same as Lisp's RPLACD.

- **SR**: function. (L: L) → (sr: Set).
  Abstraction function mapping a finite list into a finite set.

- **SR?**: function. (L: L) → (sr?: Bool).
  Predicate recognizing lists mappable to a finite set.

- **SR-ADD**: program.
  (X: O, S: (C ∪ NIL), ENV: S) → (sr-add-s: (C ∪ NIL), env-out: S).
  Adds the element X to the SR set S. APURE?. Requires no-duplicates to maintain no-duplicates. Structure appears in Section C.3.1, within the diagram for SR-UNION.

- **SR-CHOOSE**: program.
  (S: (C ∪ NIL), ENV: S) → (sr-choose-elt: O; env-out: S).
  Arbitrarily chooses some element of the input SR set and returns it. Does not require no-duplicates. Structure appears in Section C.3.1, within diagram for SR-CHOOSE+REM.

- **SR-CHOOSE+REM**: program.
  (S: (C ∪ NIL), ENV: S) →
  (sr-choose+rem-elt: O; sr-choose+rem-s: (C ∪ NIL); env-out: S).
  Arbitrarily chooses some element of the input SR set and returns it together with a representation of the set with the element removed. APURE?. Requires no-duplicates to maintain no-duplicates. Structure appears in Section C.3.1.

- SR-ELT?+UNION: program.
  (S1: (C ∪ NIL), S2: (C ∪ NIL); X: O; ENV: S) →
  (sr-elt+union?: BOOL, env-out: S).
  Determines whether X is an element of the union of the sets represented by the SR-sets S1 and S2. APURE?. Does not require no-duplicates. Structure appears in Section C.3.1.

- SR-REM: program.
  (S: (C ∪ NIL), X: O, ENV: S) → (sr-rem-s: (C ∪ NIL); env-out: S).
  Removes X from the SR set S and returns the result. APURE?. Requires no-duplicates to maintain no-duplicates. Structure appears in Section C.3.1, within the diagram for SR-CHOOSE+REM.

- SR-UNION: program.
  (S1: (C ∪ NIL), S2: (C ∪ NIL); ENV: S) →
  (sr-union: (C ∪ NIL), env-out: S).
  Constructs the union of the sets represented by the SR-sets S1 and S2. APURE?. Requires no-duplicates to maintain no-duplicates. Structure appears in Section C.3.1.

- TWO: primitive. () → (two: N). Number two constant.

- V: function. (C: C, S: S) → (V: (P ∪ U)).
  Returns either _u_ if c is unallocated in s, or the tuple (P element) to which c is bound.

- ZCOPY-CELL: program.
  (Z: Z, S: S) → (zcopy-cell-z: Z, zcopy-cell-s: S).
  Takes a series of memory cells and a store and returns a series of fresh memory cells, together with a new store. Each of the result cells has CAR field equal to the CAR field of the corresponding input cell and each of whose CDR field is uninitialized.

- ZERO: primitive. () → (zero: N). Number zero constant.

- ZERO?: program. (O: O) → (zero?: Bool).
  Returns true iff input is identical to the constant (ZERO).

- ZN<=K: primitive. (N: N) → (zn<=k: Z).
  Output is a series of the integers $0, 1, \ldots, k$. Cost is $(2, 0)$.

- **ZN<K**: primitive. **(N: N)** $\rightarrow$ **(zn<k: Z)**.
  Output is a series of the integers $0, 1, \ldots, k-1$. Cost is $(2, 0)$.

Appendix C

Selected Examples

This appendix demonstrates the system's performance on a collection of interesting examples. Each example is discussed and interesting features are highlighted. Note that some of the nomenclature for list-manipulating programs is different here than in the body of the thesis. This is done to maintain consistency with the uniform naming scheme I adopted for the experiments. Synonyms are mentioned where appropriate. See Appendix A for a discussion of the experimental domain knowledge and Appendix B for a cross-referenced glossary of program and function names.

For each program, I will summarize each of the following aspects

- *Program structure.*
- *Optimization invariants and proof generation method.* Recall that optimization invariants are simply collections of clauses. I have used four proof generation procedures in these examples: by hand; *default-proof*, which basically recapitulates the top-level structure axioms; *lr1-proof*, which incompletely proves the `LR1` properties and merely recapitulates the abstract list structure; and *sr-proof*, which incompletely proves `LR1` and `SR` properties while recapitulating abstract set structure.
- *Test case inputs.*
- *Optimizations.* I will give the boxes eliminated directly by **Try-to-eliminate-box**, possibly indicating *collateral* boxes eliminated as well. (The elimination of one box can cause the elimination of others if the

first box is the sole user of the value produced by the others. I term these others "collateral.") I will also summarize the source/target pairs.

- *Statistics.* I will report system statistics in a table (see for example, page 222). "Boxes examined" gives the number of times **Pair-Search** called **Try-to-eliminate-box**. "Boxes eliminated" gives the number of such calls returning a nonempty set of pairs. (This does not include collateral boxes.) "Total src/trg pairs" is the number of pairs examined after eliminating those sources and targets too deep in the recursion. "Syntactically pruned" is the number of such pairs eliminated by *prune-syntactically.* "No. target conditions" gives the number of target conditions computed by IEBR, hence the number of initialized targets considered. The IBR column for this row will, of course, always be empty. "Sum of trace sizes" is the sum of the numbers of entries in the program traces for the test cases. This is, of course, the same for both IEBR and IBR. This gives a relative idea of how expensive a single call to the **Inv-Screen** candidate screener is, as it must recalculate (whether it stores the values or not) a value for each such entry. "No. of trace updates" gives the number of times each candidate screener recalculated a test case trace. Thus, for example, we expect the run-time of IBR to grow on the order of the product of this field and the previous. "Candidate screening time" is the total real time (in seconds) spent in the candidate screening routine, while "Total time" is the total real time (in seconds) spent altogether. We expect the differences between these rows to be roughly the same across columns, since the rest of the time is spent in common code (this is only roughly true, since the two algorithms usually consider different sets of boxes). Real time values vary widely due to environmental nondeterminism, anyway.

- *Notes.* These are simply what I believe interesting or distinguishing about the example.

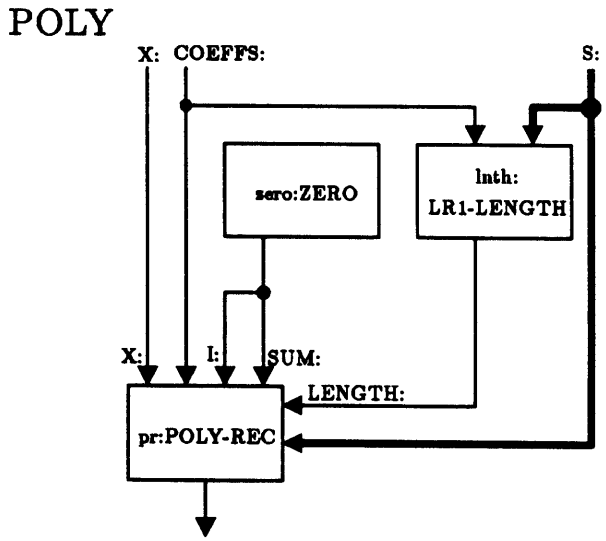
C.1 Example Programs On Numbers

These examples are of interest primarily as examples of identical-value redistribution. POLY is a good example of how a natural and self-evidently correct, but inefficient, implementation can be optimized to gain back the efficiency

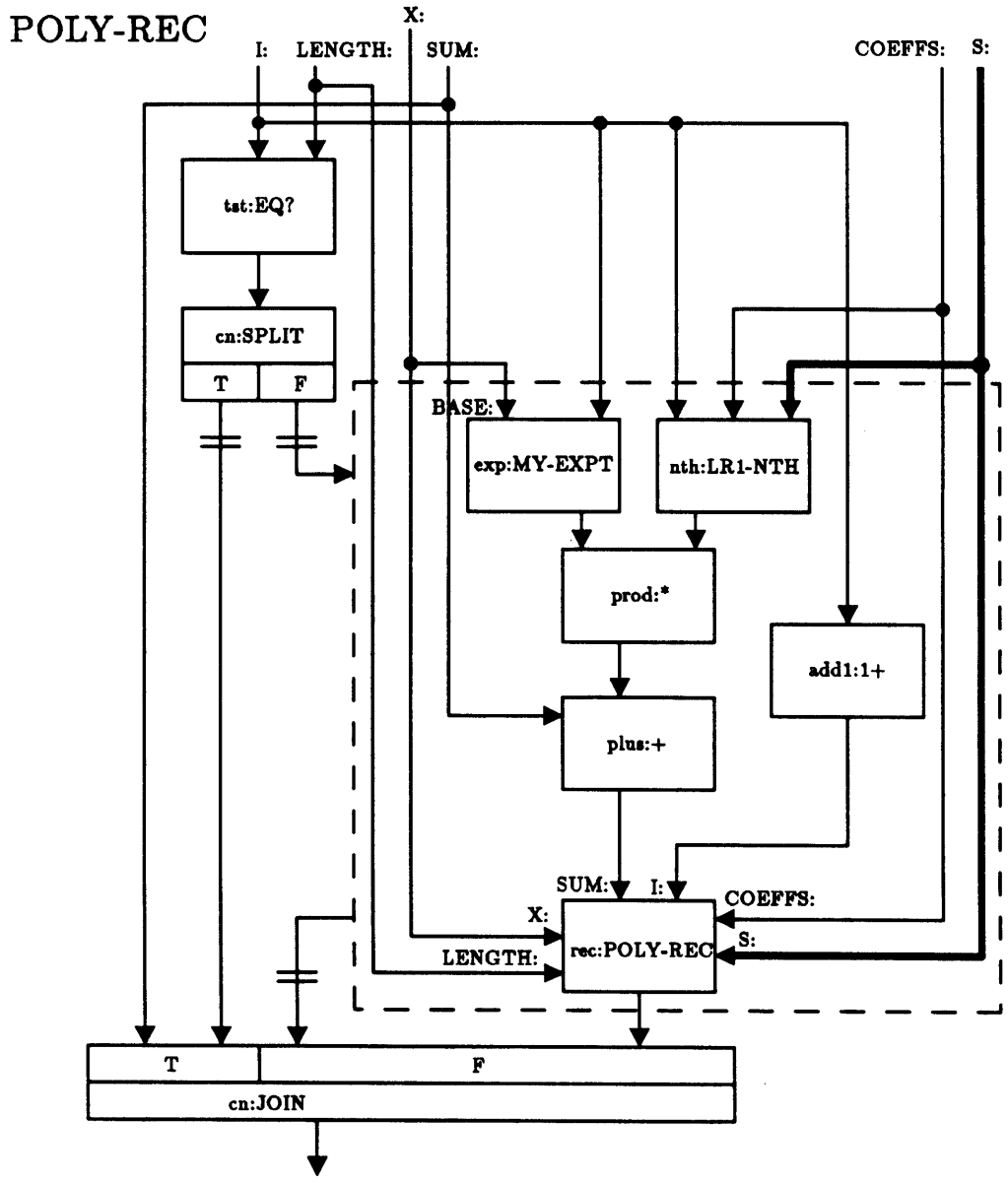
sacrificed to clarity. FIB holds the record for the largest run-time improvement: the original is exponential while the optimized result is linear.[1] FIB also illustrates a shortcoming of the technique: sensitivity to detailed form of the input program. In particular, the system fails to optimize the most natural implementation of the Fibonacci function, because single, sequential redistributions can't quite express the necessary structure changes. A slight change to the input structure, however, yields an optimizable form.

C.1.1 POLY

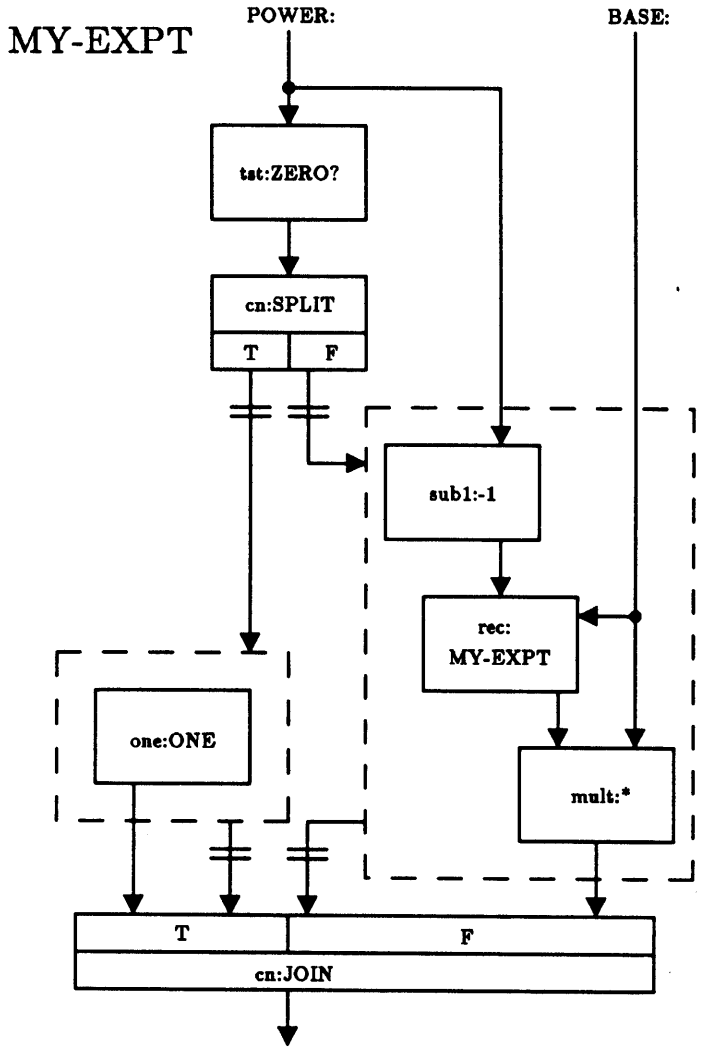
Program Structure. The system's model of the POLY program discussed in the introduction is shown below followed by its key subroutines. Some program names have been changed to conform with my naming conventions. Structurally, the main difference between the figures and the code shown in the introduction is that the loop in POLY is re-expressed as a tail-recursive subroutine.



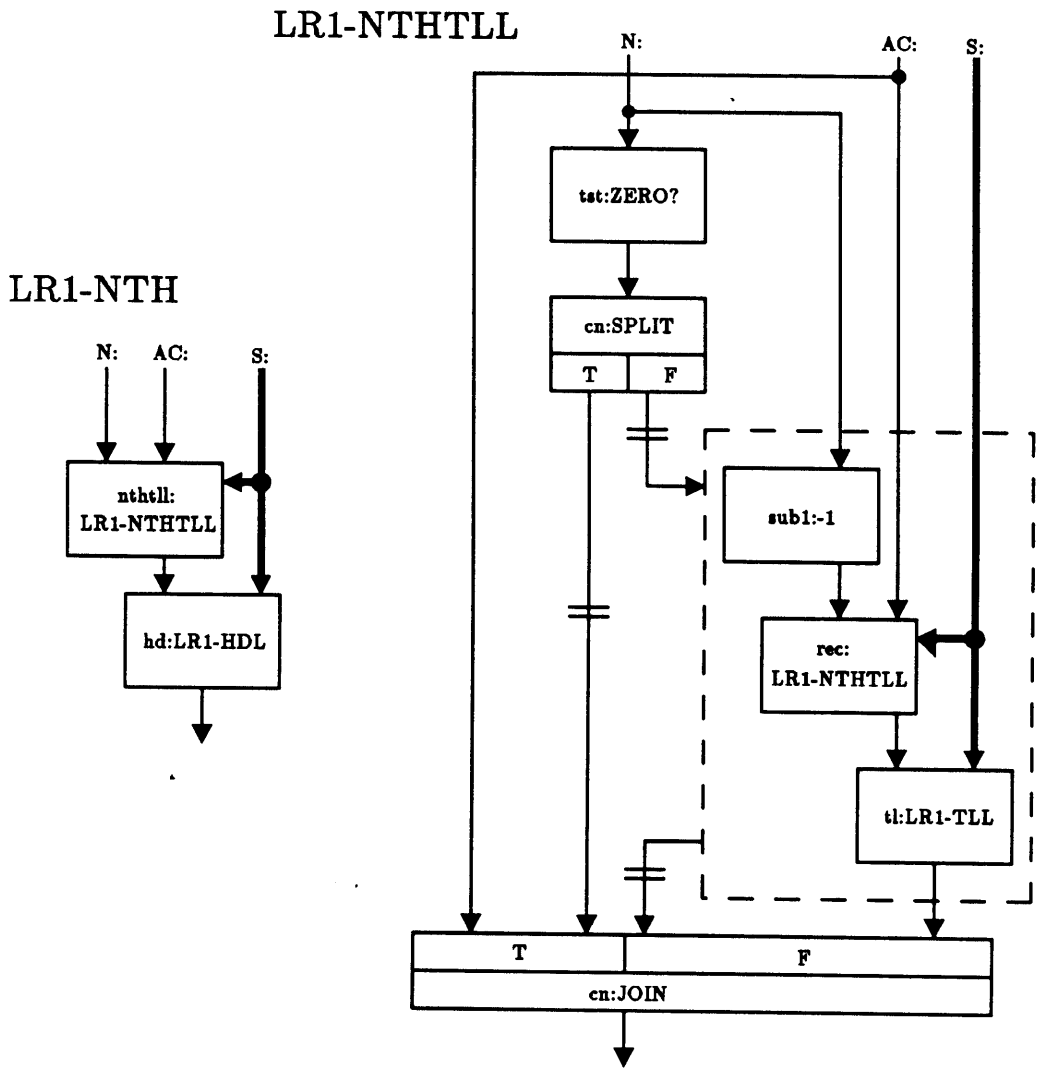
[1] One must be careful when giving the complexity of computing Fibonacci numbers. Regardless of how one computes it, the *size* of the output is exponential in the *size* of the input. The more interesting measure, however, is how the run-time grows as a function of the input (not its size). Throughout, I will be referring to run-time as a function of input, not size.



`MY-EXPT` raises a base to a power. This is the same as Lisp `EXPT`, except only applicable to nonnegative integers.



LR1-NTH and LR1-NTHTLL correspond with the Lisp functions NTH and NTHCDR, respectively. These versions are used by the POLY example. Note the recursive structure of LR1-NTHTLL; the second optimization wouldn't exist if LR1-NTHTLL were formulated in terms of series expressions.



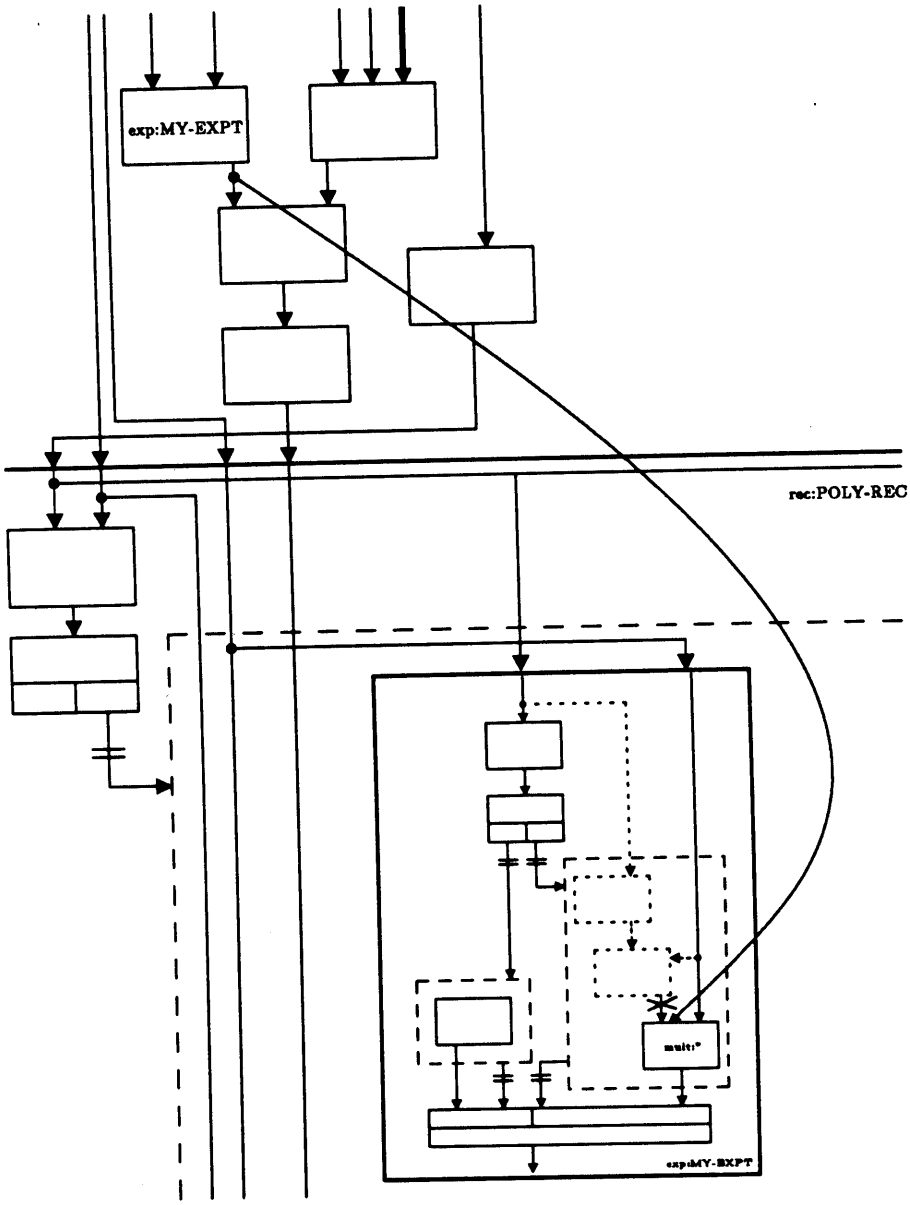
Invariants and Proofs. The specifications and proofs of `POLY` and `POLY-REC` were produced automatically by *default-proof*. The single-clause optimization invariant

```
{(= $POLY.POLY $POLY.POLY.ORIGINAL)}.
```

required that the outputs remain the same across optimizations.

Test case inputs. The single test case consisted of evaluating the degree-5 polynomial whose coefficients are $(1, 5, 10, 10, 5, 1)$ on the number 2. This is equivalent to computing $(2 + 1)^5 = 3^5$. The coefficient list was encoded in terms of memory cells and a store with only six allocated cells.

Optimizations. First, the system found the (source, target) pair (`$POLY.PR.EXP.MY-EXPT`, `$POLY.PR.REC.EXP.MULT.N2`), which eliminates the box `$POLY.PR.REC.EXP.REC`—saving a cost of $(3, 0)$. As a side benefit, this elimination in turn eliminates the collateral box `$POLY.PR.REC.EXP.SUB1`, saving an additional cost of $(2, 0)$. Here is a picture of part of the `POLY-REC` call within `POLY` showing the redistribution. Note that this is an identical-value redistribution, meaning that the new source is always equal to the old source.



In English, iteration $i+1$ of the loop (`POLY-REC`) in `POLY` calculates x^{i+1} by recursively calculating x^i and then multiplying by x. But iteration i of `POLY` just computed x^i, so this value (the source named above) could be shared correspondingly at each level of the recursion with the target named

above. This pair applies recursively, hence must be checked using the top-level invariant on the winner.

Next, the system found the pair

(**$POLY.PR.NTH.NTHTLL.LR1-NTHTLL**, **$POLY.PR.REC.NTH.NTHTLL.TL.C**).

This pair also applies recursively, hence must also be checked by the top-level invariant. This is very similar to the previous redistribution, so I won't show the diagram. It saves another $(3,0)$ box **$POLY.PR.REC.NTH.NTHTLL.REC** and another $(2,0)$ box **$POLY.PR.REC.NTH.NTHTLL.SUB1**.

In English, iteration $i+1$ of the loop in **POLY** traverses the list **COEFFS** from the beginning (within **LR1-NTH**, which calls **LR1-NTHTLL**)in order to find (**LR1-NTH** (i+1) **C S**). This computation finds (**LR1-NTHTLL** (i+1) **COEFFS S**) by recursively finding (**LR1-NTHTLL** i **COEFFS S**) and then taking the **LR1-TLL**. (**LR1-NTHTLL** i **COEFFS S**), however, was computed in iteration i of **POLY**, so could be shared for each i.

Note that **IBR** found a third redistribution as well (and thereby eliminated one more box). The $(2,0)$ box **$POLY.LNTH.SCN.MDATA** was eliminated via the pair (**$POLY.LNTH.SCN.SCN.SCAN-NEXTS**, **$POLY.LNTH.SCN.LR1-SCAN**). This is a context-independent optimization of **LR1-LENGTH**. The idea is that **LR1-SCAN** need not access the data fields of each cell in the list, because only the length of the series output is important. This is interesting in that it does not fall into one of the four classes discussed in the introduction.

Here is Lisp code for the optimized version of the program. Note that I have coded the altered subroutines in-line (for clarity only—the system maintains the modularity) and used local variables and explicit iteration in place of extra function arguments and tail recursion respectively.

```
(DEFUN POLY-2 (COEFFS X)
  (LET ((SUM 0)
        NTH-SAVE
        EXPT-SAVE)
    (DOTIMES (I (LR1-LENGTH COEFFS))
      (SETQ SUM (+ SUM
                   (* (CAR (IF (= I 0)
                               (SETQ NTH-SAVE COEFFS)
                               (SETQ NTH-SAVE (LR1-TLL NTH-SAVE))))
                      (IF (= I 0)
                          (SETQ EXPT-SAVE 1)
                          (SETQ EXPT-SAVE (* X EXPT-SAVE))))))))
    SUM))
```

POLY Statistics		
	IEBR	IBR
boxes examined	56	56
boxes eliminated	2	3
total src/trg pairs	836	836
syntactically pruned	358	358
no. target conditions	42	—
sum of trace sizes	1058	
no. of trace updates	60	478
Candidate Screening time (sec)	290	714
Total time (sec)	374	797

Notes.

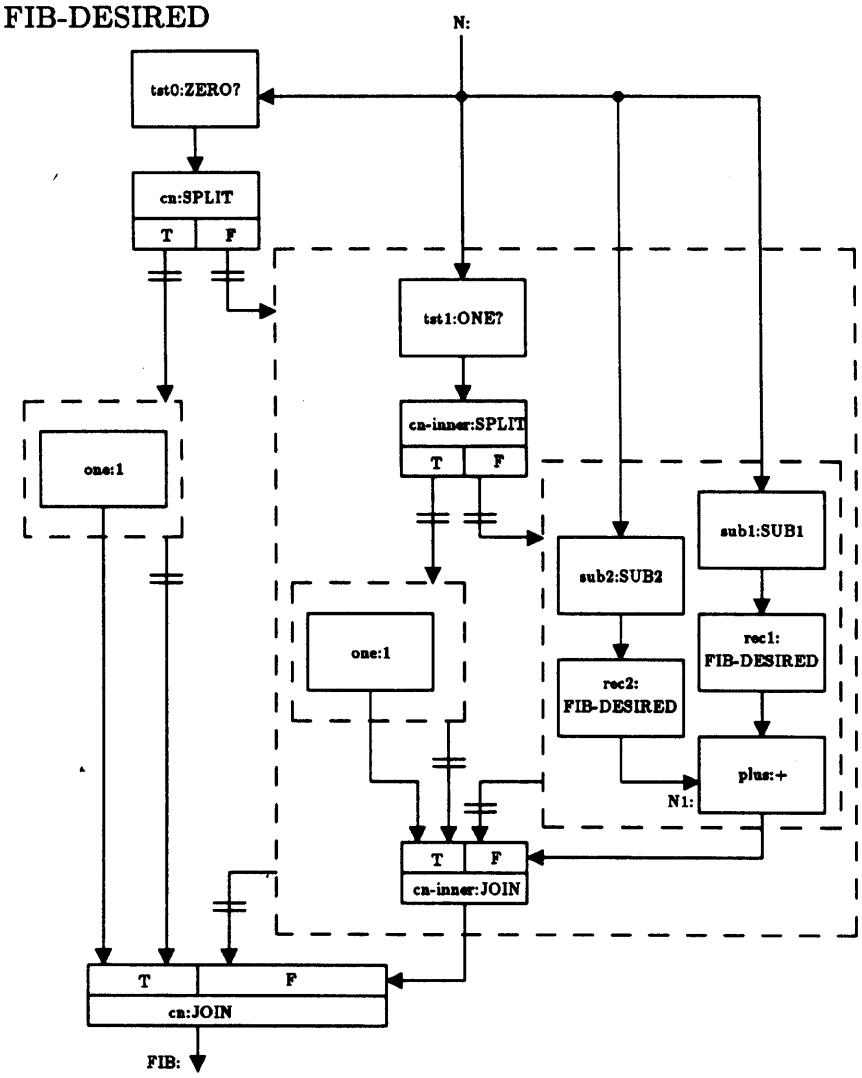
- POLY-2 requires only time linear in the length of the coefficient list, where POLY was quadratic.
- The additional optimization found by IBR not found by IEBR was a redistribution applying to LR1-LENGTH's implementation independently of context. It was not found by IEBR due to the weakness of the *default-proof* proof-generation technique used for LR1-LENGTH. (The structure of LR1-LENGTH is shown with its entry in the glossary.)
- All other redistributions found were identical-value type.

C.1.2 Fibonacci Numbers: FIB

Program Structure. Fibonacci numbers are defined by the formula

$$\text{fib}(n) = \begin{cases} 1 & n = 0, 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

Thus, the most natural program to compute them looks like



This calls itself exponentially many times, hence is very inefficient since Fibonacci numbers can be computed using linearly many (or even logarithmically many) calls. Annoyingly, redistributions as I've described them here are insufficient to get rid of the inefficiency, though they *almost* do. The pair

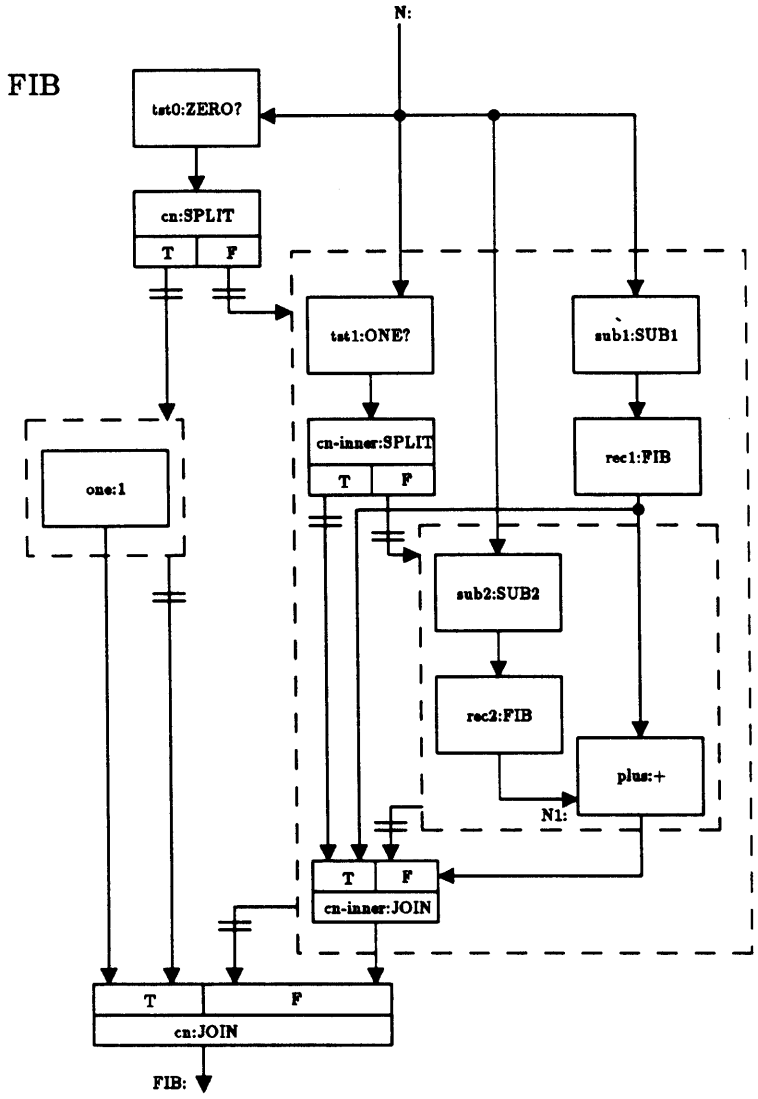
(`$FIB-DESIRED.REC1.REC1.FIB-DESIRED`, `$FIB-DESIRED.PLUS.N1`)

is correct recursively at every level, *except* the recursive call just before the base-case. In that call, the source is uninitialized, since it lies in the base-case

call which executes the wrong half of the conditional. For any invocation in which the REC1 box executes the false side of the conditional (i.e. all but one invocation), the pair works fine.

Though I have some ideas[2] about dealing with this problem (which has come up more than once), the current system cannot deal with it. Thus, I was forced to tweak the program structure slightly into a form that did work. The FIB program is shown below.

[2]... such as examining the traces of a failing recursive redistribution to decide whether all but the base case actually succeeded and, if so, searching for a base-case-only redistribution to perform simultaneously. This would be a limited case of non-trivial adaptation problem-solving.



Note that the only difference between `FIB` and `FIB-DESIRED` is that two boxes are moved out of the innermost conditional and one constant box is removed.

Invariants and Proofs. The specifications and proofs of `FIB` was produced automatically by *default-proof*. The single clause optimization invariant was

```
{(= $FIB.FIB $FIB.FIB.ORIGINAL)}.
```

Test case inputs. The single test case input was 4.

Optimizations. IBR and IEBR found exactly the same sets of pairs. The first optimization eliminated the $(3,0)$ box `$FIB.REC2` recursively, via the pair

(`$FIB.REC1.REC1.FIB`, `$FIB.PLUS.N1`)

This gets rid of the second recursive call, reducing the number of subroutine calls from exponential to linear in the input (the system, of course, only thought the reduction was from quadratic to linear, since it had underestimated the exponential complexity of `FIB`).

The second optimization eliminated the $(2,0)$ box `$FIB.TST2`, via the recursive pair

(`$FIB.REC1.TST1.ZERO?`, `$FIB.CN-INNER.TEST`)

This saves a test on each invocation—not very significant.

`FIB` Statistics	IEBR	IBR
boxes examined	7	7
boxes eliminated	2	2
total src/trg pairs	58	58
syntactically pruned	26	26
no. target conditions	4	—
sum of trace sizes	236	
no. of trace updates	4	32
Candidate Screening time (sec)	7	12
Total time (sec)	14	19

Notes.

- The cost estimate of $(3,0)$ for this program is misleading, as the program is exponential, not quadratic.
- The result program is linear, while the original is exponential. This is the greatest improvement demonstrated so far. The only optimizations were identical-value.

- The test case, 4, is the smallest that suffices to get all the correct optimizations and no others. It has to be at least 2 in order to exercise the interesting branches of the conditionals; and the input must be great enough for the output to differ from the input (otherwise the system would get rid of all boxes!).

- Depressingly, the most natural program structure for computing Fibonacci numbers does not yield the desired efficiency improvement, because of the base-case problem: a promising source/target pair is correct everywhere except one call above the base-case invocation. The small change made to get the **FIB** version (essentially, unfolding the call immediately preceding the base-case invocation into a special-case conditional) looks like it might be easy to automate.

C.2 Example Programs On LR1 Lists

These examples illustrate many interesting phenomena, such as copy elimination, identical value redistribution, and generalized loop fusion.

C.2.1 LR1-CONCAT-C+D (APPEND)

Program Structure. Structure for this program is shown in Figure 1.3. In experiments run on the system, the nomenclature was different: **APPEND** was named **LR1-CONCAT-C+D** for **LR1** list program doing **CONCAT**enation on lists via Copy and Destroy. (I also experimented with the standard recursive implementation of **APPEND**, which was named **LR1-CONCAT-REC**, but will not discuss it here.) **COPY-LIST** was named **LR1-COPY**, and so forth (the glossary in Appendix B contains cross-references). I will use the nomenclature in the diagrams here, but recall that the functionality is not exactly the same as Lisp's **APPEND**, because **LR1** lists are not exactly Lisp lists.

Invariants and Proofs. The invariants are essentially just the usual specification of **APPEND**, which I paraphrase here in English:

- If the input lists satisfy **LR1?**, then every cell allocated in the input store is bound to the same values in the output store.

- If the input lists satisfy `LR1?`, then the output cell and store satisfy `LR1?` as well.
- If the input lists satisfy `LR1?`, then the abstraction of the output list (via `LR1`) is the list concatenation of the abstractions of the input lists.

The proofs for this and for the key subroutine, `NCONC`, were all produced by hand.

Test Case Inputs. I used three test case input triples (cell-1, cell-2, store) each encoding two lists as follows (the notation `<C0>, <C1>, ...` indicates memory cells numbered 0, 1, etc.):

- `((), (<c1> 5289))`
- `((<c6> 343 <c1> 5289), (0 0 <c0>))`
- `((<c6> 343 <c1> 5289), (<c1> 5289))`

This is probably not a minimal set.

Optimizations. The single redistribution found by both algorithms is shown in Figure 3.6. Essentially, it is a loop fusion of the iteration within the copying and the iteration in `NCONC`. It saves the $(2, 0)$ box, `$APPEND.NC.LC.SCN`.

LR1-CONCAT-C+D (APPEND) Statistics		
	IEBR	IBR
boxes examined	15	15
boxes eliminated	1	1
total src/trg pairs	146	146
syntactically pruned	105	105
no. target conditions	18	—
sum of trace sizes	189	
no. of trace updates	1	41
Candidate Screening time (sec)	51	37
Total time (sec)	71	59

Notes.

- The optimization improved the efficiency to that of the standard, recursive implementation of APPEND. Thus, one might wonder why not simply give the system that implementation to begin with? The answer that the form given is advantageous in contexts where the copying is unnecessary; it is both easier to find and easier to justify optimizations that remove needless copying from the explicit-copy version than from the recursive version. (For an example of removing the copying in an appropriate context, see the MY-REVERSE example. For an example of removing copying that is not in the explicit copy-and-modify structural form, see the LR1-REM+REVAPPEND example to follow.) Thus, it is better to give the system a seemingly inefficient structural model and let it optimize to fit the context than to try to optimize first.

- In this case, IEBR took longer than IBR (even though both examined exactly the same sets of pairs) because the run-time was so small that the low-order run-time term corresponding to target condition computation dominated. Generally, there are roughly linearly many targets and quadratically many pairs that pass *prune-syntactically.* In this case, however, the 41 pairs passing the syntactic pruning stage required 18 target conditions to be computed. Thus, at roughly two pairs per target condition, the computation time became significant for this example.

- While the optimization probably did not require the relatively large time investment of a hand proof, the hand proof enabled many optimizations in other examples that used LR1-CONCAT-C+D. I basically treated this program more like a "library module" than like a "user program."

C.2.2 LR1-REV (MY-REVERSE)

Program Structure. This is the example discussed in the introduction (Section 1.3). Again, the nomenclature is different between the introduction and the examples run on the system. The MY-REVERSE program was named LR1-REV. Some of the subroutines were named differently as well; see the glossary for cross-indexing. Important structure is shown in Figure 1.2 and in Figure 1.3.

Invariants and Proofs. These are discussed in Section 1.3.

Test Case Inputs. These are discussed in Section 1.3.

Optimizations. These are discussed in Section 1.3.

LR1-REV (MY-REVERSE) Statistics		
	IEBR	IBR
boxes examined	24	24
boxes eliminated	3	3
total src/trg pairs	561	561
syntactically pruned	338	338
no. target conditions	31	—
sum of trace sizes	882	
no. of trace updates	11	223
Candidate Screening time (sec)	103	345
Total time (sec)	158	396

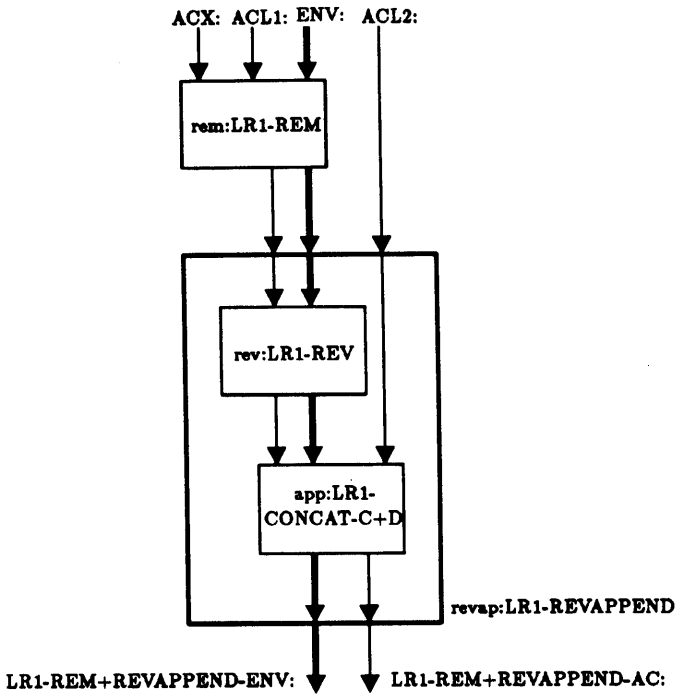
Notes.

- Space and time costs improved from quadratic to linear after the copy elimination and the identical-value redistributions.
- Both algorithms found the same optimizations and examined the same sets of pairs. However, IEBR was three times faster in candidate screening than IBR (about 2.5 times faster overall).

C.2.3 LR1-REM+REVAPPEND

Program Structure. This program takes in an object and two LR1 lists, removes the object from the first list, reverses the result, and appends the second list. The top-level program is shown here, with the LR1-REVAPPEND box's implementation shown. LR1-REV is a synonym for the MY-REVERSE program (Figure 1.2) and LR1-CONCAT-C+D is a synonym for the APPEND program (Figure 1.3). Note that no optimizations involved the internals of the LR1-REM box; it was included simply to provide a fresh list to demonstrate how side-effects can be introduced into programs not explicitly structured as copy-and-modify.

LR1-REM+REVAPPEND



Invariants and Proofs. The invariants used were APURE?ity and abstract equality to the original outputs. Proofs were produced by *lr1-proof*.

Test Case Inputs. I used seven test cases. This is probably not minimal, but it seemed "representative," *a priori*. In each case, I give the object, the first list and the second list. Each list is encoded in terms of cells and the store input.

- `0, (<c5> 0 <c0> 3 2 0), (<c3> true)`
- `<c3>, (), (<c3> true)`
- `0, (0), (2 0)`; lists share structure
- `1, (0), (2 0)`; lists share structure
- `<c0>, (<c0> 3 2 0), (<c3> true)`

- `0, (2 0), (<c3> true)`
- `1, (2 0), (<c3> true)`

Optimizations. Here are the boxes eliminated by IEBR. I have not shown the redistributions simply because there are so many. In all cases, the targets can be inferred directly from the box name and the program structure. I have indicated the justifications (hence implied the sources) in English.

- `$LR1-REM+REVAPPEND.REVAPP.REV.CONC.CPY` and `$LR1-REM+REVAPPEND.REVAPP.REV.CONC.NC.LC`. These are the same as the two `MY-REVERSE` optimizations.
- `$LR1-REM+REVAPPEND.REVAPP.APP.CPY`. This is a copy elimination within the `APPEND` box.
- `$LR1-REM+REVAPPEND.REVAPP.REV.SNG.CNS.NW`. This introduces side-effects into the `MY-REVERSE` box by eliminating the allocation of the new cells in favor of reusing the input cells.
- `$LR1-REM+REVAPPEND.REVAPP.APP.NC.LC`. This takes advantage of the last-cons pointer available within the last invocation of the `MY-REVERSE` box to avoid the last-cons computation within the `APPEND` box.
- `$LR1-REM+REVAPPEND.REVAPP.REV.SNG.CNS.RPA`. This eliminates the re-initialization of the `CAR` fields of the reused input cells to `MY-REVERSE`, since they are already set to the correct data.
- `$LR1-REM+REVAPPEND.REVAPP.REV.CONC.NC.NU` and `$LR1-REM+REVAPPEND.REVAPP.APP.NC.NU`. These are just avoiding the null tests within the implementations of the `APPEND` boxes by sharing with previous test results.

The boxes above were eliminated by both IEBR and IBR. IBR went farther and eliminated the following two boxes as well.

- `$LR1-REM+REVAPPEND.REVAPP.REV.SNG.CNS.RPD`. This eliminates the resetting of the `CDR` fields of the reused input cells to `MY-REVERSE` because

they are set later by **$LR1-REM+REVAPPEND.REVAPP.REV.CONC.NC.RPD** (non-end cells) and by **$LR1-REM+REVAPPEND.REVAPP.APP.NC.RPD** (end cell). This is a fundamental false negative for IEBR.

- **$LR1-REM+REVAPPEND.REVAPP.REV.REC.HD.CR.SL.** This is eliminated because its result is no longer used after the **$LR1-REM+REVAPPEND.REVAPP.REV.SNG.CNS.RPA** box was eliminated. This should not have been considered by **Try-to-eliminate-box?**, but due to a small bug in the code for marking boxes physical, the recursive version of the box (note the presence of ...**REC**...) remained marked as physical. This error does not significantly effect the results, as it just implies a few extra boxes were examined by both IBR and IEBR. Since any such box examined by IBR is eliminated, I can detect such instances. No others were found.

LR1-REM+REVAPPEND Statistics		
	IEBR	IBR
boxes examined	59	57
boxes eliminated	8	10
total src/trg pairs	1403	1345
syntactically pruned	937	881
no. target conditions	80	—
sum of trace sizes	2463	
no. of trace updates	14	464
Candidate Screening time (sec)	305	1202
Total time (sec)	517	1399

Notes.

- This demonstrates how IEBR screens out pairs that (even though they are correct) are likely to be hard to prove. IBR, on the other hand, makes no such distinction. (This is discussed fully in Section 9.6.3.) The redistribution missed by IEBR is a fundamental false negative.

- This example also demonstrates how the system can introduce side-effects into a program that is not as explicitly structured ("copy-and-modify") as, for example, **APPEND**. The key is that each "cons" operation

(allocation and initialization of a single memory cell) is treated as consisting of an allocation followed by explicit initialization of the fields. The system is therefore able to leave out any subset of these. In this example, the allocation is removed by substituting a pre-existing cell for the output of the allocation box within the cons within LR1-REV. Since this pair takes effect recursively, it removes all extra allocation.

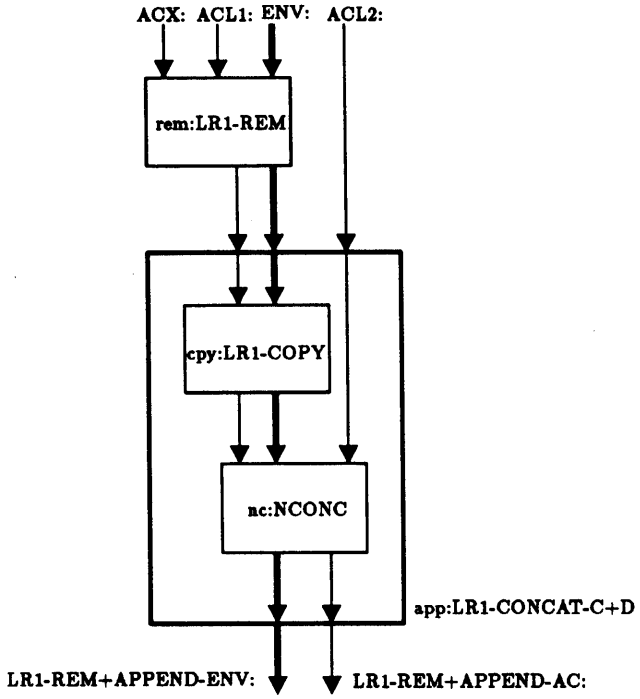
- Note that even though IBR removed a couple of more boxes than did IEBR, the resulting programs are essentially equally efficient. IEBR's result program is only a small constant factor ($\ll 2$) slower and uses no more space than does IBR's result.

- Note that the APPEND box has both its copy operation removed and its last-cons box removed. Essentially, only some tests and a single RPLACD operation remain. This is a third form into which the seemingly inefficient APPEND model can be transformed, giving further support to using that model, rather than the recursive one. The recursive one could not be transformed into this form, since there is no explicit, single RPLACD box present in it.

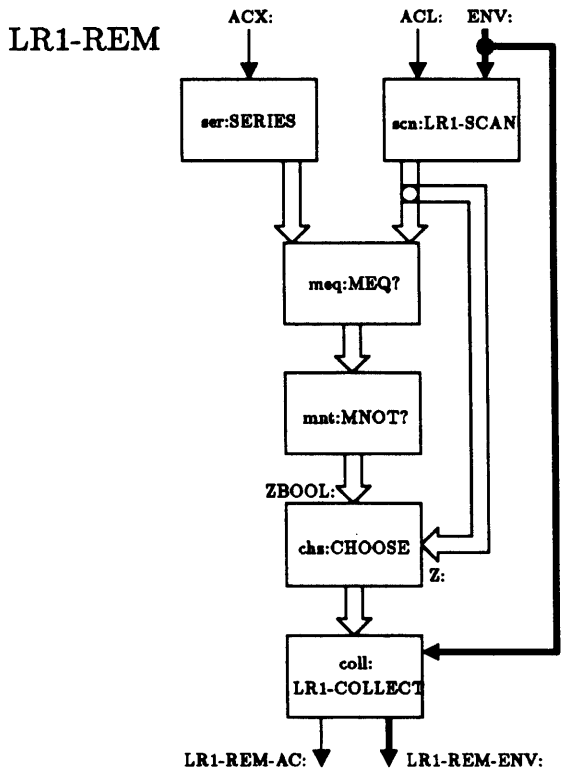
C.2.4 LR1-REM+APPEND

Program Structure. Here is the top-level structure of LR1-REM+APPEND

LR1-REM+APPEND



and here is the structure of **LR1-REM**:



The internal structure of **NCONC** is shown in Figure 1.3.

Invariants and Proofs. The invariants used were **APURE?**ity and abstract equality to the original outputs. Proofs were produced by *lr1-proof.*

Test Case Inputs. I used the same seven test cases as in **LR1-REM+REVAPPEND**. Again, this is probably not minimal. In each case, I give the object, the first list and the second list. Each list is encoded in terms of cells and the store input.

- `0, (<c5> 0 <c0> 3 2 0), (<c3> true)`
- `<c3>, (), (<c3> true)`
- `0, (0), (2 0)`; lists share structure
- `1, (0), (2 0)`; lists share structure

- `<c0>, (<c0> 3 2 0), (<c3> true)`
- `0, (2 0), (<c3> true)`
- `1, (2 0), (<c3> true)`

Optimizations. Here are the redistributions performing the same type of copy elimination as in `MY-REVERSE`.

- (`$LR1-REM+APPEND.REM.LR1-REM-AC`, `$LR1-REM+APPEND.APP.NC.C1`)
- (`$LR1-REM+APPEND.REM.LR1-REM-ENV`, `$LR1-REM+APPEND.APP.NC.ENV`)

Next, the system performed a fusion of the implicit loop in `LR1-REM` with the last-cons collection within `LAST-CONS` within `NCONC`. Here is the (series) pair:

- (`$LR1-REM+APPEND.REM.COLL.MSNG.MSNGLTN-Z`, `$LR1-REM+APPEND.APP.NC.LC.COLL.Z`)

These removed the following two boxes.

- `$LR1-REM+APPEND.APP.CPY`, a copy box.
- `$LR1-REM+APPEND.APP.NC.LC.SCN`, a list scanning box (representing an iteration).

`LR1-REM+APPEND` Statistics		
	IEBR	IBR
boxes examined	21	21
boxes eliminated	2	2
total src/trg pairs	298	298
syntactically pruned	211	211
no. target conditions	28	—
sum of trace sizes	739	
no. of trace updates	3	87
Candidate Screening time (sec)	41	120
Total time (sec)	77	156

Notes.

- This example illustrates the effectiveness of the series representation for iteration. The second optimization, a savings of $(2, 0)$ (a linear-time iteration), is made possible only by the fact that the scanning of the list is explicit. I performed this experiment also with the standard recursive definition of **LR1-REM** and the fusion was not performed. (The copy elimination was done in either case, as to be expected.)

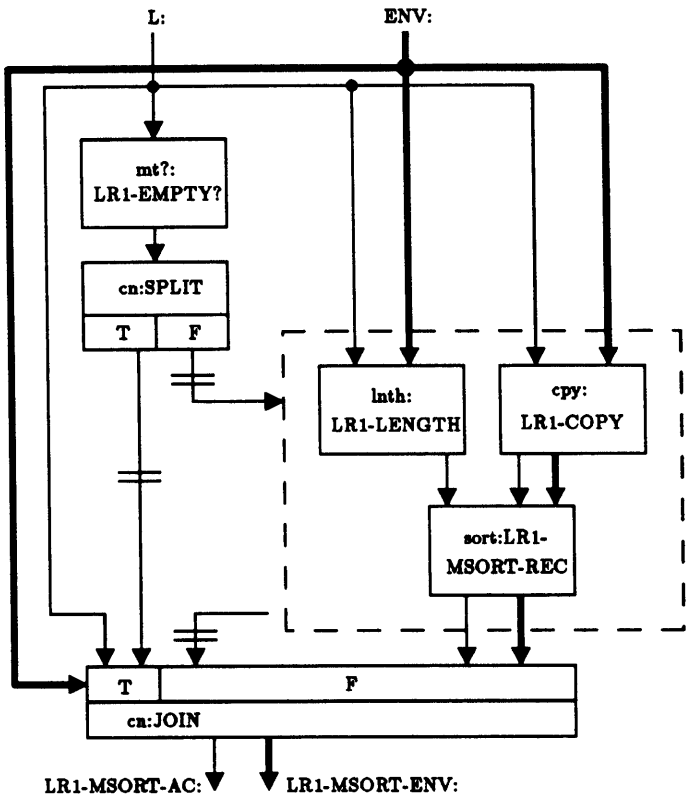
- Note that the **APPEND** implementation is reduced effectively to a simple **RPLACD** operation plus a **COLLECT-LAST** operation. Both of these are constant time and zero space, while **APPEND** was originally linear time and space. A similar thing happened in the **LR1-REM+REVAPPEND** example above, but in a different way—the entire **LAST-CONS** box of **NCONC** was eliminated there.

- IBR and IEBR examined exactly the same pairs and gave exactly the same answers.

C.2.5 LR1-MSORT (MERGE-SORT)

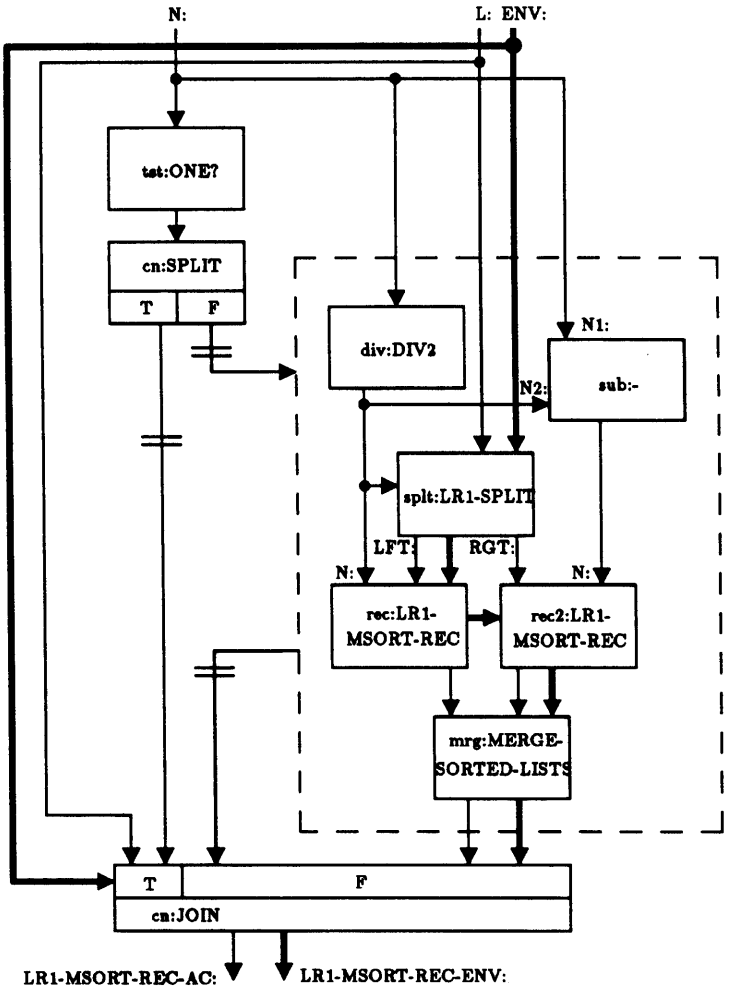
Program Structure. Here is the top-level structure of **LR1-MSORT**.

LR1-MSORT



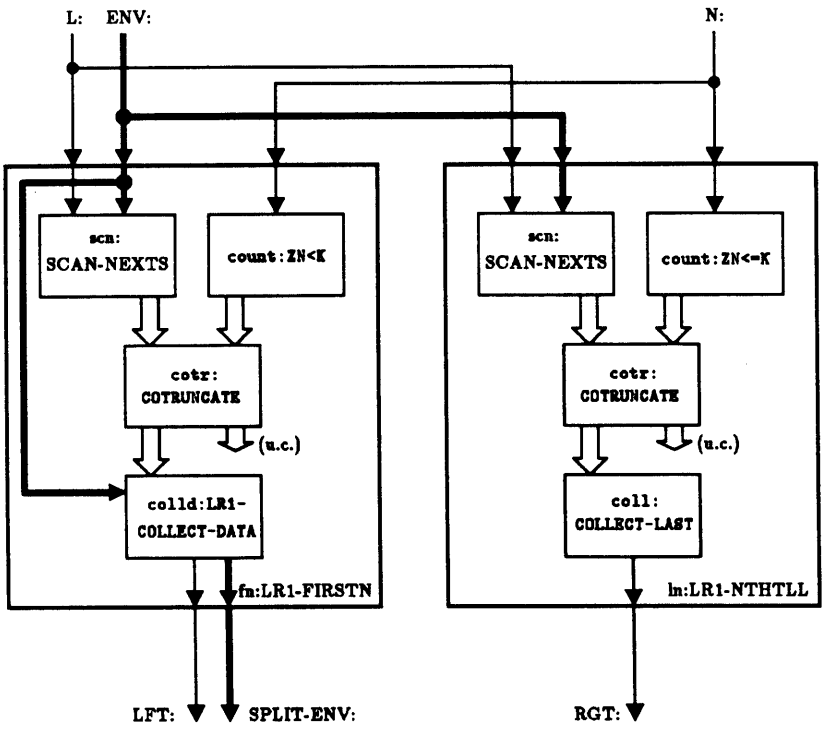
Here is its key recursive subroutine, `LR1-MSORT-REC`, which implements the familiar split-sort-merge sorting algorithm.

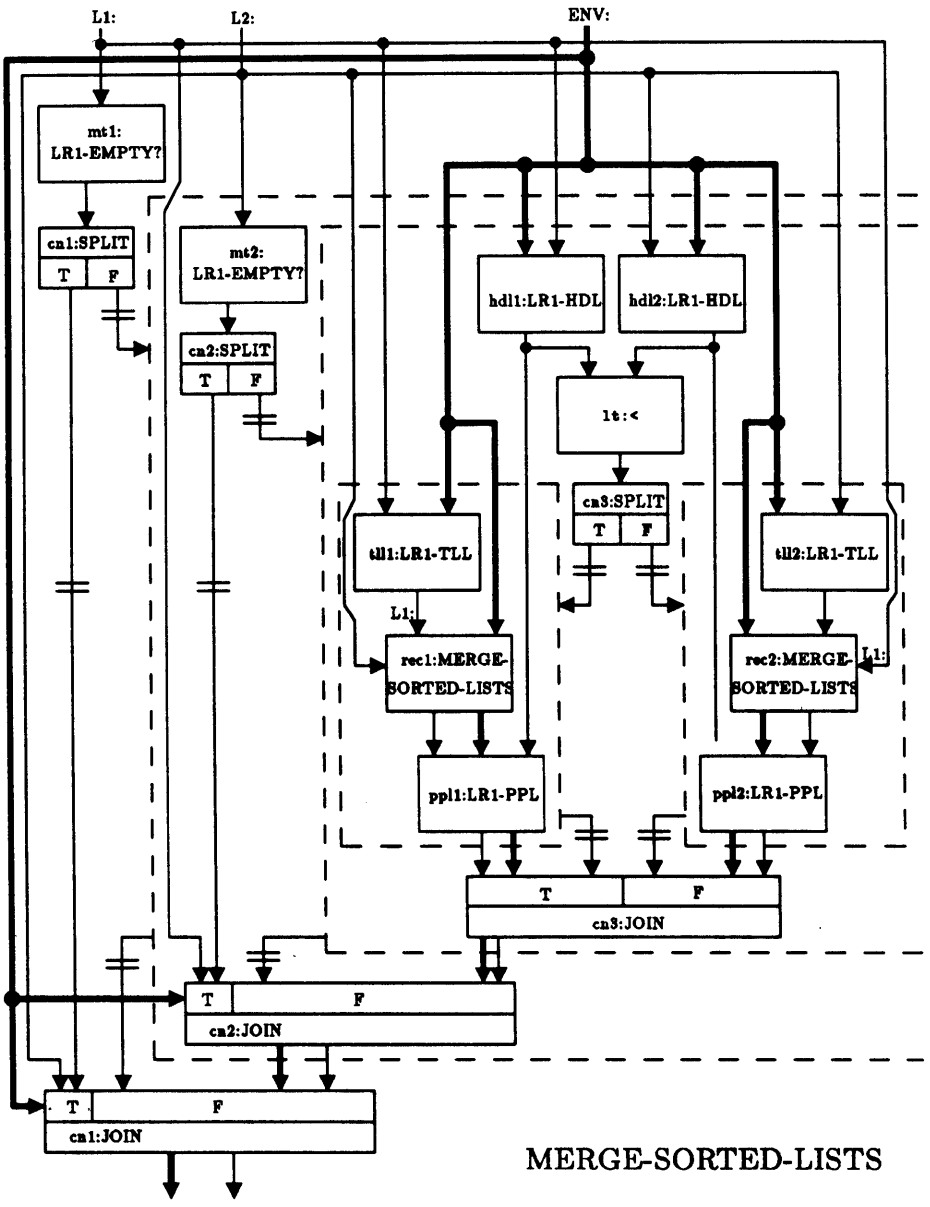
LR1-MSORT-REC



Here are the other important subroutines.

LR1-SPLIT


L:
ENV:
N:
scn:
SCAN-NEXTS
count:ZN<K
cotr:
COTRUNCATE
(u.c.)
colld:LR1-
COLLECT-DATA
fn:LR1-FIRSTN
scn:
SCAN-NEXTS
count:ZN<=K
cotr:
COTRUNCATE
(u.c.)
coll:
COLLECT-LAST
ln:LR1-NTHTLL
LFT:
SPLIT-ENV:
RGT:



Invariants and Proofs. The invariants used were APURE?ity and abstract equality to the original outputs. Proofs were produced by *lr1-proof*. Note that the program is correct only on lists of numbers. This is not captured by any of the invariants or proofs, but all test cases are lists of numbers,

so it is not necessary. In theory, this omission could result in less optimization power (fewer explicit preconditions conditionalizing things), but it did not effect the results in this case.

Test Case Inputs. The test cases are all single LR1 lists encoded in cells and stores. Here are the ten test cases used. The set is probably not minimal, though they were not chosen completely at random: the last three were chosen to exercise previously unreached portions of the program.

- `()`
- `(0)`
- `(0 1 2 3 3 2 1 0)`
- `(3 3 2 1 0)`
- `(1 0)`
- `(0 2 4 6 1 3 5 7)`
- `(6 1 3 5 7)`
- `(9 2 3 8 0 6 1 4)`
- `(7 10 5 9 2 3 8 0 6 1 4)`
- `(1 6 0 2 4 7 5 3)`

Optimizations. There are too many redistributions to show here, so I show only the boxes eliminated, together with some intuition as to why. Here are the boxes eliminated by IEBR:

- **`$LR1-MSORT.SORT.MRG.PPL1.CNS.NW`**
  and **`$LR1-MSORT.SORT.MRG.PPL2.CNS.NW`**. The memory allocations within the two different ways of adding an element to the result list in the merge box are eliminated by reusing the input structure to the merge box.

- $LR1-MSORT.SORT.SPLIT.FN.COLLD.ZCC.SNG.CNS.NW. The memory allocation within the FIRSTN box's iteration is eliminated by reusing the input cells.
- $LR1-MSORT.SORT.SPLIT.FN.SCN. This fuses the iteration within FIRSTN with that of LR1-NTHTLL.
- $LR1-MSORT.SORT.MRG.PPL1.CNS.RPA and $LR1-MSORT.SORT.MRG.PPL2.CNS.RPA. Since we are reusing the input lists to merge (see above), the cells already have the CAR fields set correctly, so we needn't reinitialize them.
- $LR1-MSORT.SORT.SPLIT.FN.COLLD.ZCC.PUSH. This is mostly an artifact of the formalism; it is just the observation that the output series (of cells) of the ZCOPY-CELL box is the same as the input series after we've eliminated the memory allocation.
- $LR1-MSORT.SORT.SPLIT.FN.COLLD.ZCC.SNG.CNS.RPA. This avoids reinitializing the CAR fields of the reused cells within the FIRSTN box's ZCOPY-CELL.
- $LR1-MSORT.CPY.SCN. This fuses the loop within the length operation with that of the first copy operation.

The only difference in the boxes eliminated by IBR is that the box

- $LR1-MSORT.SORT.SPLIT.FN.COLLD.ZCC

is eliminated instead of three of its internal boxes as before.

LR1-MSORT Statistics		
	IEBR	IBR
boxes examined	370	318
boxes eliminated	9	7
total src/trg pairs	22858	16425
syntactically pruned	16157	11532
no. target conditions	475	—
sum of trace sizes	22736	
no. of trace updates	254	4893
Candidate Screening time (sec)	11944	55955
Total time (sec)	14980	59081

Notes.

- Note that the copy operation at top-level of LR1-MSORT would appear to be useless in that LR1-MSORT-REC, as defined, is side-effect-free. This is analogous to the program structure insight in APPEND: putting in seemingly extraneous structure allows better optimizations. Without the top-level copy, the system could not get rid any internal copying done within the recursive procedure. This would result in (altogether) two copies being constructed of the input list instead of exactly the one needed.[3] This can be viewed as an instance of a problem with the approach: small changes to the input program have relatively large effects on the optimization results.

- The system had a choice of eliminating the first copy operation (not within LR1-MSORT-REC) or to do what it did (eliminate the copies within the recursive part). It could not do both, else it would destroy the input list. It made the right choice, because the system estimated the costs of the inner copies higher, hence it considered them first.

- The system estimated the cost of the entire program at $(3, 3)$, which is the same as a quadratic program. The program is actually $n \log n$. In this case, the discrepancy didn't matter, but if we had a program that contained both $n \log n$ boxes and n^2 boxes, the system could consider them in the wrong order. This is a drawback of the crudeness of the cost estimates.

- LR1-MSORT required the most time to optimize (for both algorithms).

- The optimizations were copy eliminations, loop fusions, and avoidance of unnecessary data invariant operations (the eliminations of RPLACA boxes when the CAR fields are already correct). The run-time improved by a (significant) constant factor and the space usage dropped to the optimal space usage for a non-destructive sort. (Space usage was cut by a factor of three over the original program.)

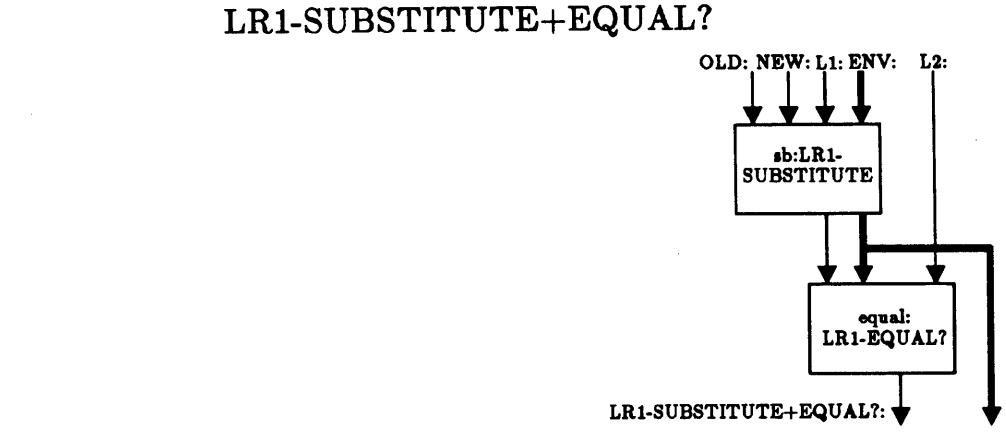
- IBR was able to eliminate a box that IEBR couldn't, but IEBR was able to compensate by eliminating most of its insides. Thus, the relative

[3]Counting up the copies within LR1-MSORT-REC gives one a total of two, but they are spread out piecemeal within the recursion.

weakness of IEBR wasted time, but didn't result in a significant loss of optimization power.

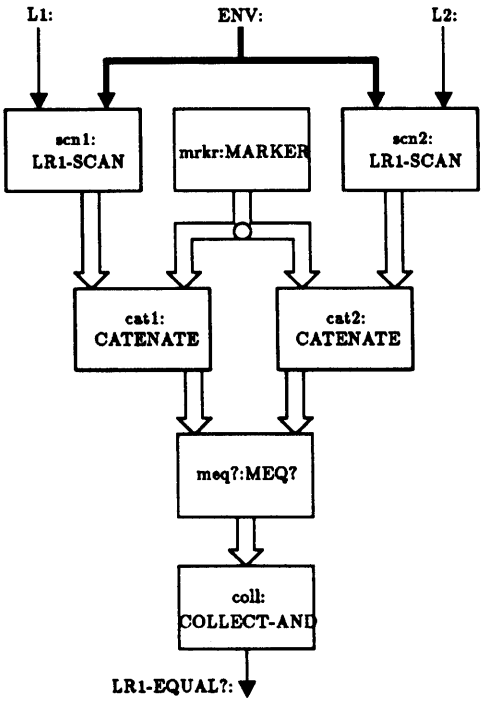
C.2.6 LR1-SUBSTITUTE+EQUAL?

Program Structure. Here is the top-level program structure for LR1-SUBSTITUTE+EQUAL?:

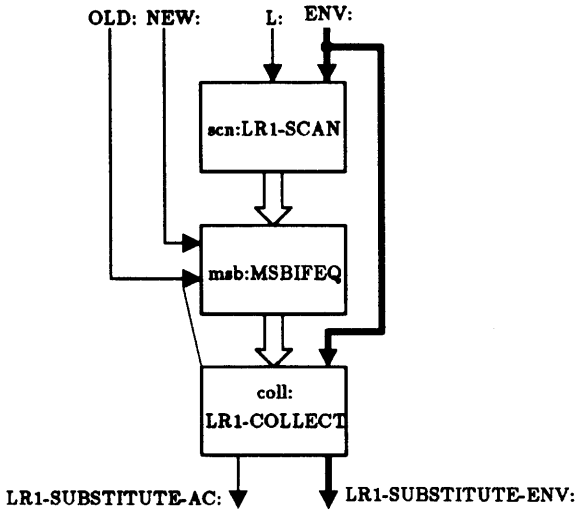


and here is the structure for its two library subroutines:

LR1-EQUAL?



LR1-SUBSTITUTE



Invariants and Proofs. The invariants used were APURE?ity and simple equality to the original outputs. Proofs were produced by *lr1-proof*.

Test Case Inputs. I used five test cases (not known to be minimal). I give them in the form new-object, old-object, substituted-list, test-list. Again, all lists were encoded in terms of cells and a store.

- `1, true, (0 true 0), (0 1 0)`
- `1, true, (0 true 0 true 0), (0 1 0)`
- `1, 0, (0 true 0), (0 1 0)`
- `1, true, (0 true 0), (0 true 0)`
- `true, 1, (true 0), (true 0 true 0)`

Optimizations. Here, there was a significant difference in performance between IEBR and IBR. Here is the redistribution found by both:

- (`$LR1-SUBSTITUTE+EQUAL?.SB.MSB.MSBIFEQ`, `$LR1-SUBSTITUTE+EQUAL?.EQUAL.CAT1.Z1`). This eliminated the box `$LR1-SUBSTITUTE+EQUAL?.EQUAL.SCN1`.

This is a loop fusion.

Here are the pairs found additionally by IBR:

- (`$LR1-SUBSTITUTE+EQUAL?.ACOLD`, `$LR1-SUBSTITUTE+EQUAL?.SB.COLL.LR1-COLLECT-AC`)
- (`$LR1-SUBSTITUTE+EQUAL?.ENV`, `$LR1-SUBSTITUTE+EQUAL?.SB.COLL.LR1-COLLECT-ENV`)

These eliminated the box

- `$LR1-SUBSTITUTE+EQUAL?.SB.COLL.COLL`

which allocates and builds the substituted list. Thus, the result of IBR uses no extra space, but IEBR was unable to eliminate this box, hence still uses extra space.

LR1-SUBSTITUTE+EQUAL? Statistics		
	IEBR	IBR
boxes examined	17	17
boxes eliminated	1	2
total src/trg pairs	219	210
syntactically pruned	140	134
no. target conditions	20	—
sum of trace sizes	370	
no. of trace updates	2	76
Candidate Screening time (sec)	14	75
Total time (sec)	37	101

Notes.

- This is an example where the weakness of IEBR made a significant difference in the optimization results. IEBR's version ended up having a cost of $(2,2)$, because it still creates the substituted list output of the LR1-SUBSTITUTE box, while IBR's $(2,0)$ version doesn't. The extra $(2,2)$ box is significantly costly.
- The optimization missed by IEBR is a fundamental false negative.

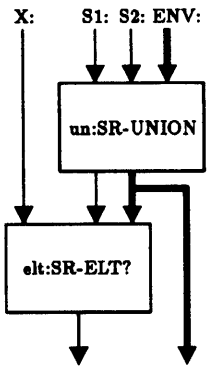
C.3 Example Program On SR Sets

This example demonstrates data invariant suspension and a good test case coincidence (a "false true positive").

C.3.1 SR-ELT?+UNION

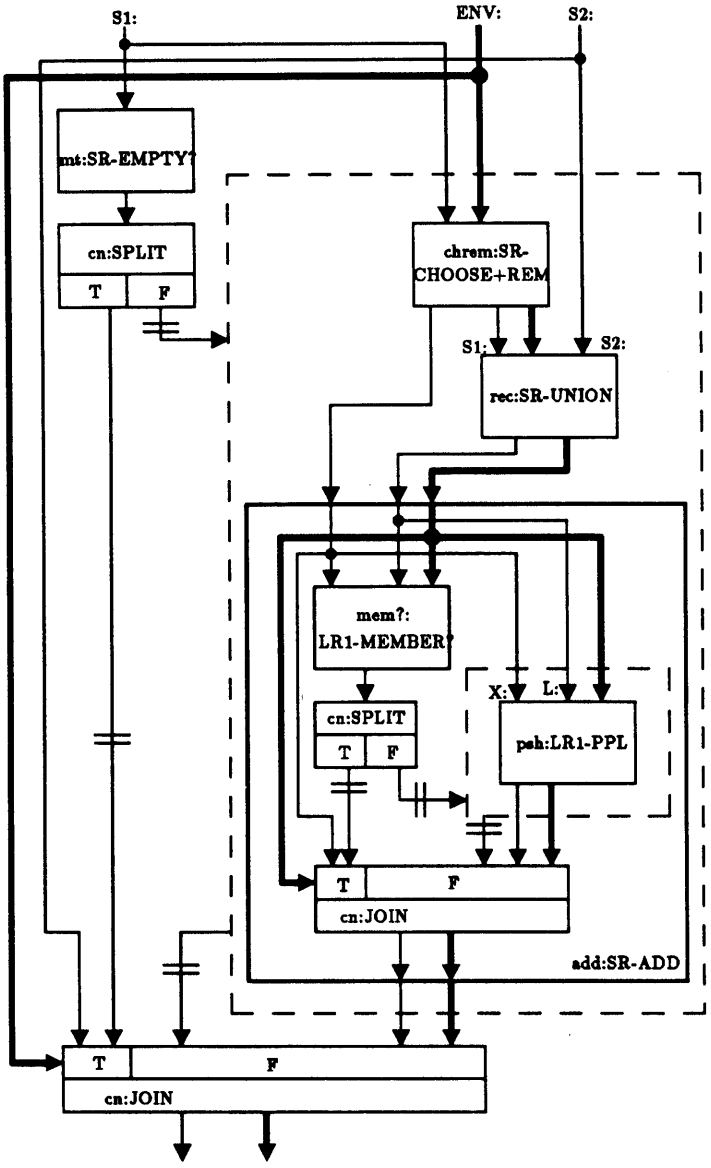
Program Structure. Here is the top-level program structure.

SR-ELT?+UNION



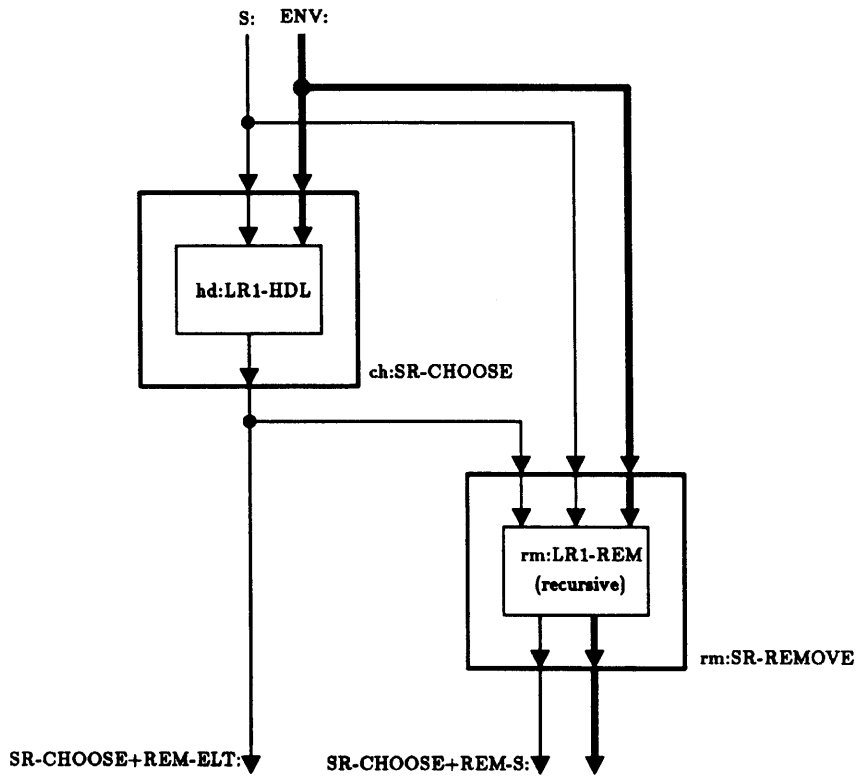
Here is the key subroutine, **SR-UNION**.

SR-UNION



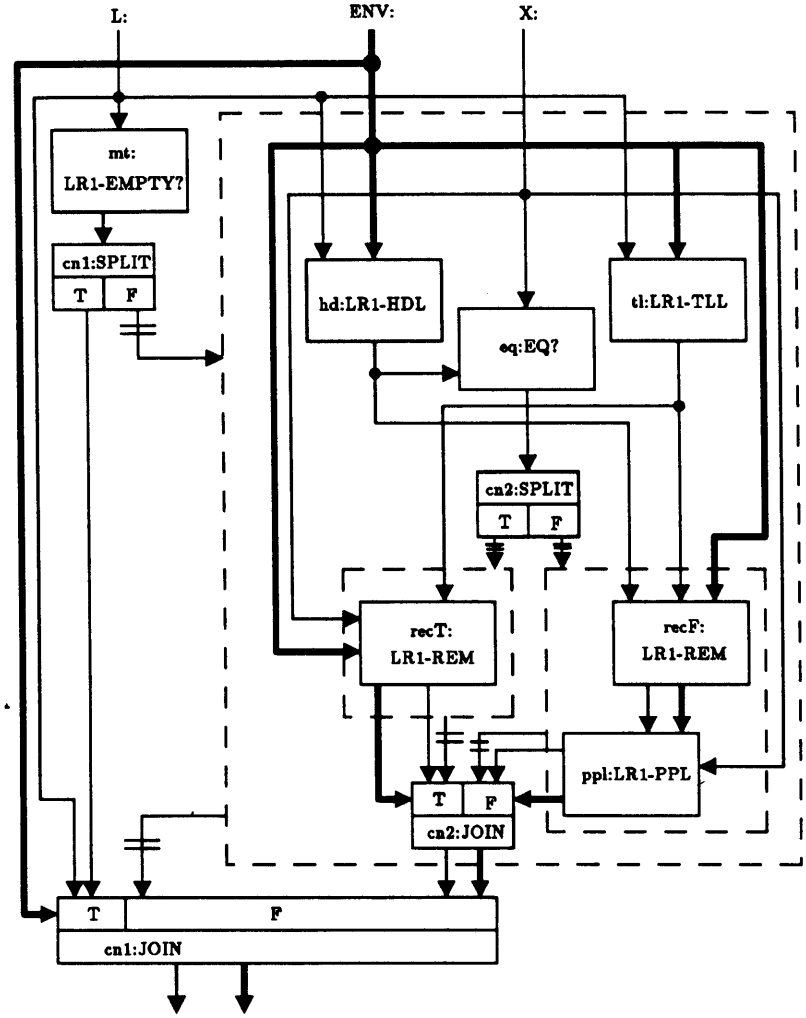
Here is the key subroutine of `SR-UNION`, `SR-CHOOSE+REM`.

SR-CHOOSE+REM



And here is the key (recursive) implementation of `LR1-REM`:

LR1-REM (recursive version)



Invariants and Proofs. The invariants used were APURE?ity and simple equality to the original output. Proofs were produced by (hand-simulation of) *sr-proof*.

Test Case Inputs. I used seven test cases (with degeneracy, see below) and added one later to avoid the degeneracy. Here are the seven original test cases. I state them as test element, set-list-1, and set-list-2. All are

represented in terms of cells and stores. I give lists rather than sets, because the list representation is important to the examples (i.e. the ordering and whether the list has duplicate entries).

- `7, (7 0 <c0> 3 NIL 0), (7 2 11 1).`
- `11, (7 2 11 1), (7 0 <c0> 3 NIL 0).`
- `3, (7 2 11 1), (7 0 <c0> 3 NIL 0).`
- `false, (7 2 11 1), (7 0 <c0> 3 NIL 0).`
- `0, (), (7 2 11 1).`
- `0, (1), (7 2 11 1).`
- `2, (0), (7 2 11 1).`

Here is the extra test case that gets rid of the "good" degeneracy. Note its similarity to the first test case above.

- `7, (0 <c0> 3 NIL 0), (7 2 11 1).`

Optimizations. Using the original seven test cases, both IBR and IEBR gave exactly the same answers. Here are the boxes eliminated.

- `$SR-ELT?+UNION.UN.CHREM.REM.RECT`
- `$SR-ELT?+UNION.UN.ADD.MEM?`
- `$SR-ELT?+UNION.UN.CHREM.REM.HD.CR`

Adding the eighth test case causes IEBR to fail to eliminate the first box above, resulting in a less efficient result program. It still gets the other two, however, so some improvement remains.

SR-ELT?+UNION Statistics		
	IEBR	IBR
boxes examined	111	111
boxes eliminated	3	3
total src/trg pairs	2218	2192
syntactically pruned	1623	1610
no. target conditions	82	—
sum of trace sizes	6137	
no. of trace updates	22	582
Candidate Screening time (sec)	524	3356
Total time (sec)	881	3652

Notes.

- This illustrates data invariant suspension. The no-duplicates invariant is suspended within the SR-UNION box since SR-ELT? doesn't require it. This is illustrated by the elimination of the $SR-ELT?+UNION.UN.ADD.MEM? box. Note that both algorithms eliminated this box under both sets of test cases.

- The statistics above are for the run with only the original seven test cases. Those seven illustrate a peculiar phenomenon: "good" test case coincidence. This is where, instead of the coincidence causing a false positive, it causes a true positive, but adding a test case results in a fundamental false negative! (I whimsically term this a "false true positive.") Using the original seven, IEBR was able to eliminate the recursion box within the SR-REMOVE box, because all seven test cases had the property that the first element of the first set-list was not duplicated later in the same set-list. Thus, the local specification of SR-REMOVE was maintained by simply taking the tail of the set-list to remove the first element (this implied the satisfaction of a particular target condition). Of course, the SR-REMOVE is *not* maintained that way if the first element is duplicated; subsequent occurrences remain in the list. But on the other hand, the top-level optimization invariants remain satisfied, because the abstract set properties of this program are not effected by duplicates in the list. Thus, using the original test case set, IEBR was right, but for the wrong reasons! Adding the eighth test

case exposed the fundamental false negative for IEBR. IBR accepted the redistribution using either test case set, of course.

- Note that the above optimization (within the **SR-CHOOSE+REM** box) is only possible because **LR1-REM** is implemented recursively. The series form given earlier (**LR1-REM+APPEND** example) does not have the recursion box which is eliminated here. This shows that neither series nor recursion can be used exclusively.

- Note that even though the outputs of the two routines (on the seven test cases) were the same, this does not imply that the screening procedures gave exactly the same answers everywhere. (**Inv-Screen** can answer "true" while **EB-Screen** answers "false", yet the resulting redistribution may eventually be retracted because the corresponding box was not eliminated. This explains why such statistics as total src/trg pairs aren't exactly the same.

Appendix D

Proof Restructuring

Proof structure has a significant impact on target condition generality, as shown in Chapter 9. This appendix gives an algorithm for improving the weakened relative conditions of given leaves by computing a "better" proof. In addition to its use in this research, the restructuring algorithm given here may be of interest to anyone attempting to integrate the PCCU approach to generalization into traditional EBG systems.

The intuition is that for a weakened relative condition to be larger (contain more disjuncts), the path from it to the root of the tree should contain more disjuncts. This can be accomplished by re-ordering resolution steps within the proof. The basic technique will be to apply a set of local proof transformations to the tree that combine to push the leaves of interest downward, away from the root. It will be convenient to *mark* the leaves of interest, together with every node on the paths to the root. Marking provides a means of controlling the transformation process in order to avoid looping and to ensure progress toward a better tree.

Notation. It will be useful to represent local tree restructuring rules in the following compact form.

```
(I{A/?x} [j(A,B)]                    (I{A/?x}o{B/?y} [j(A,B)]
   (I{B/?y} [j(?x,B)]        -->                    (* [j(?x,?y)]))
         (* [j(?x,?y)])))
```

A tree node is represented by a list of the form

(operator{substitutions if applicable} [clause] subtree list)

where operator is I for :INSTANTIATION nodes, R for :RESOLUTION nodes, S for :SUBSUMPTION nodes, DA for :DEFINITION-APPLICATION nodes, and D for :DEFINITION nodes. An asterisk in the operator position means any operator is allowed there. The term $j(A,B)$ in the clause denotes any *clause* (not necessarily a single term) some of whose terms have A or B as subterms. As usual, identifiers starting with a question mark are free variable terms. A clause of the form [xAB~C] denotes the clause, $x \vee A \vee B \vee \overline{C}$, where lower-case letters denote clauses and upper-case letters denote terms (disjuncts). The substitution A/?x indicates replacing occurrences of the variable ?x with the term A. Substitutions may be composed, as in the right-hand tree schema above, by the notation $s \circ t$, which means first perform substitution t, then perform substitution s on the result.

D.1 Preprocessing

To simplify the algorithm, we would like to assume certain regularity conditions on the tree which may not be present in the given proof. Thus, the system first preprocesses each tree to establish the following conditions.

D.1.1 Eliminate Macro Nodes

For ease of use, I allowed the user to insert various notationally convenient proof node types such as :UNARY, which allows many unit resolution steps to be compressed into one; and :EQUALITY, which allows single paramodulation steps. Each such operator is a straight-forward macro for some combination of resolutions and tautologies. The first preprocessing step expands these macros.

D.1.2 Instantiation Pushing

A disjunct with a free variable appearing in a weakened relative condition is logically stronger than a corresponding disjunct with the free variable instantiated. Thus, we would like the clauses above marked leaves to be in terms of the instantiated versions rather than the open forms. We can achieve the

Instantiation Regularity Condition: Instantiation nodes appearing in the tree have only leaf children.

by starting from the root and recursively walking the tree top-down, applying the following local transformations where applicable. Note that no two are applicable at the same node.

```
(I{A/?x} [j(A)]          -->   (T [j(A)])
      (T [j(?x)]))

(I{A/?x} [j(A)]                   (S [j(A)]
      (S [j(?x)]          -->        (I{A/?x} [k(A)]
         (* [k(?x)])))                    (* [k(?x)])))

(I{A/?x} [j(A,B)]                 (I{A/?x}o{B/?y} [j(A,B)]
    (I{B/?y} [j(?x,B)]    -->           (* [j(?x,?y)]))
          (* [j(?x,?y)])))

(I{A/?x} [l(A)]                   (R [l(A)]
     (R [l(?x)]                      (I{A/?x} [j(A)]
        (* [j(?x)])       -->             (* [j(?x)]))
        (* [k(?x)])))                 (I{A/?x} [k(A)]
                                        (* [k(?x)])))

(I{A/?x} [j(A)]                    (DA [j(A)]
      (DA [j(?x)]         -->          D
          D                           (I{A/?x} [k(A)]
          (* [k(?x)])))                     (* [k(?x)])))
```

D.1.3 Non-T Tautology Elimination

The system next searches the tree for any (non-`:TAUTOLOGY`) node whose clause is a propositional tautology and transforms it into a `:TAUTOLOGY` node, discarding any subtrees. The system has a switch that enables the more powerful operation of finding all PE-tautologies, but this appeared to take much more time than it was worth, and the only regularity condition needed for the restructuring algorithm is the

P-Tautology Regularity Condition: Any node whose clause is a propositional tautology has node type `:TAUTOLOGY`.

D.1.4 Subsumption Pulling

For simplicity in implementing the rest of the restructuring algorithm, we would like the

> **Subsumption Regularity Condition:** No :SUBSUMPTION nodes appear in the tree, except possibly a single one at the root.

This condition is established by a single, bottom-up tree walk applying the following transformations:

```
(S [k]                  -->     (* [k])
   (* [k]))

(S [xyz]                        (S [xyz]
   (S [xy]              -->        (* [x]))
      (* [x])))

(R [xy]                          (S [xy]
   (* [~Ay])            -->         (* [x]))
   (S [Ax]
      (* [x])))

(R [xyz]                        (S [xyz]
   (* [~Ay])                       (R [xy]
   (S [Axz]             -->           (* [~Ay])
      (* [Ax])))                      (* [Ax])))

(DA [jkPy]                      (S [jkPy]
    (D P=x)             -->        (* [j]))
    (S [jk]
       (* [j])))

(DA [Pkl]                       (S [Pkl]        ;assumes k not in x
    (D P=x)             -->        (DA [Pl]
    (S [jk]                           (D P=x)
       (* [j])))                      (* [j])))

(DA [Pkl]                        (DA [Pkl]      ;assumes k in x
    (D P=x)             -->          (D P=x)
    (S [jk]                          (* [j])))
       (* [j]))
```

D.1.5 Local Inefficiencies

We would like also to eliminate certain limited forms of inefficiency in the proof structure. Of course, we would like to eliminate all inefficiency, but the set below is easy to compute and still simplifies the rest of the algorithm. The following two rules are performed during a top-down tree walk:

```
(R [xyzB]                    (S [xyzB]
   (* [xAB])          -->       (* [yB]))
   (R [yz~A]
      (* [yB])
      (* [z~B~A])))

(R [xyzB]                    (S [xyzB]
   (* [xAB])                    (R [xyB]
   (R [yz~A]          -->          (* [xAB])
      (* [y~AB])                 (* [y~AB])))
      (* [z~B])))
```

To establish the

> **Efficiency Regularity Condition:** No pair of resolution steps can be replaced by a subtree using a subset of the same three starred subtrees that also uses fewer resolution steps.

we must iterate this procedure together with the S-pulling procedure until no inefficiencies exist and the only subsumption nodes are at the root.

Procedure **Remove-local-inefficiencies** (TREE) : void
(Side effects to the input tree only)
While local inefficiencies remain
Remove them using the tree walk described above
Pull S nodes to the root

Clearly, since the number of resolution nodes strictly decreases with each pass, this iteration terminates. In practice, it is rare to have even a second iteration, much less more than two.

D.1.6 Preprocessing Pseudo-code

Procedure **Preprocess-tree** (TREE) : void
(Side effects to the input tree only)
Remove macro nodes.
Push instantiations down to leaves.
Remove non-T tautologies.
Remove-local-inefficiencies(TREE)

D.2 The Restructuring Algorithm

Before giving the algorithm, it is necessary to define a few more tree transformations.

D.2.1 Removing Unneeded Marked Leaves

An important way to generalize a target condition is to remove an occurrence of the flow arc leaf node from the tree if possible. Clearly, if the PCCU algorithm would produce a weakened relative condition for a node that is a P-tautology, that leaf may be replaced by a `:TAUTOLOGY` node, assuming the ancestor nodes of the leaf have their clauses suitably enlarged. The procedure **Remove-unneeded-marked-leaves** carries out this transformation. I will not give the pseudo-code here, because for efficiency the actual implementation is uninterestingly more complicated than a simple application of PCCU plus a check. It turns out that once this is applied, it may be necessary to re-establish the subsumption regularity condition, so the S-pulling procedure is included as a post-pass.

D.2.2 Pushing Marked Leaves Down

This transformation is the heart of the algorithm. It has the effect of making the paths longer from marked leaves to root, thereby enlarging the resulting weakened relative conditions. Up to this point, all transformations have either reduced the number of nodes in the tree, or left it the same. Here, there are two rules that leave the number the same, and one that increases the number. I have indicated which nodes are marked using the letter M.

Here are the two nonincreasing rules:

```
M  (R [xyz]                      M  (R [xyz]
M     (* [xA])                   M     (R [xz~B]
      (R [yz~A]          -->  M        (* [xA])
         (* [yb])                      (* [z~B~A]))
         (* [z~B~A])))              (* [yB]))

M  (R [Pyx]                      M  (DA [Pyx]
M     (* [x~A])                       (D P=w)
      (DA [PyA]          -->  M       (R [xyw]
          (D P=w)                M        (* [x~A])
          (* [wyA])))                     (* [wyA])))
```

Here is the increasing rule (a.k.a. "the exponential rule"):

```
M  (R [xyz]                      M  (R [xyz]
M     (* [xA])                   M     (R [xz~B]
      (R [yz~A]          -->  M        (* [xA])
         (* [yB~A])                    (* [z~B~A]))
         (* [z~B~A])))           M     (R [xyB]
                                 M        (* [xA])
                                          (* [yB~A])))
```

I also included analogous rules with all combinations of the signs of the various resolving literals. All of these are applied in a top-down tree walk to form the procedure **Push-marked-leaves-down**.

D.2.3 Tree Restructuring Pseudo-code

Procedure **Restructure-tree** (TREE) : void
(Side effects to the input tree only)
Preprocess-tree(TREE)
Mark desired nodes.
Remove-unneeded-marked-leaves(TREE)
Mark desired nodes.
While **Push-marked-leaves-down** would change something
Push-marked-leaves-down(TREE)
Remove-local-inefficiencies(TREE)
Remove-unneeded-marked-leaves(TREE)

If all three rules are used in **Push-marked-leaves-down**, then at termination no resolution nodes remain unmarked, unless the process is blocked by a `:DEFINITION-APPLICATION` node that couldn't be transformed by the single DA rule. In practice, this blocking has not been a problem, though I haven't used `:DEFINITION-APPLICATION` nodes much. Unfortunately, the exponential resolution restructuring rule can cause an exponential blowup in the size of the tree, since it doubles a subtree. Thus, one way of restricting the algorithm would be to rule out the use of the exponential rule. This would allow cases of blocking on resolution nodes not satisfying the constraints of the other rule. The reward, however, would be a guaranteed polynomial time algorithm, since the number of marked nodes strictly increases, but the total number of nodes cannot increase.

In practice, I have not had to discard the exponential restructuring rule, as it almost never fires on trees of interest. This restriction may, however, need to be imposed for use with other sets of test cases.

Appendix E

Programmability versus Checkability

This subsection explores the theoretical relationship between problems that are computable, checkable, and relatively checkable. It is tangential and may be skipped without loss of continuity.

A computational problem is *checkable* if there exists a program for deciding whether a given data value is a correct output for a given input value. It is *relatively checkable* if it is checkable by a program that is allowed to use an oracle for the problem. An *oracle* for a problem is a black box that gives a correct output in a single time step for any input; note that it is not required to give *all* correct outputs for a single input value.

For a computational problem, I will rate its computability, checkability, and relative checkability on a qualitative scale of

- EASY: has a polynomial-time computational solution.
- HARD: NP-complete or worse, but still computable
- IMPOSSIBLE: uncomputable.

Note that I assume the relative checker has a constant-time oracle for the correct program's outputs; thus, I do not count the time to find them originally in the time for the relative checker. This is realistic with respect to the system's use of relative checking.

The following general observations follow easily.

- Obviously, the relative checkability of a program is no harder than the checkability, since any checker can be used as a relative checker.

- If a program is easy to compute and easy to check relatively, then it is easy to check absolutely by simply computing the problem and then relatively checking it. Thus, the difficulty of checking is no worse than the maximum of computing and relative checking.

- If a problem with denumerable range, which includes all problems computable on a digital computer, is checkable, then it is computable. Simply enumerate all possible input/output pairs for the given input and check them; eventually one will be correct, so return the output component.

- For any decision problem, checking and computing are the same, since the outputs of computing the one-bit answer give no additional information.

- Any problem for which there is only one correct, polynomial-sized output per input is obviously trivial to check relatively: just compare the output to that of the known-correct program.

Aside from these facts, however, there is no relationship between the difficulties of the three types of problem measures, as Table E.1 shows. The table covers the ten possibilities not ruled out by the above observations.

Meta halting problem. By the "Meta halting" problem, I mean computing for any n some program, $mhp(n)$ that takes in any Turing machine m and if m is of size n then $mhp(n)$ decides whether m halts on blank tape, otherwise if m is not of size n then $mhp(n)$ simply puts out FALSE. The meta halting problem must be impossible to compute, because otherwise we could compute the usual halting problem by first finding the size s of the Turing machine, then computing $mhp(s)$, and applying the result to the original Turing machine. Meta halting is impossible to check because it is impossible to know even whether $mhp(n)$ halts on all inputs, much less whether it puts out the correct answer.[1] Meta halting is impossible to check relatively

[1]Note that for any particular n, $mhp(n)$ is computable since there are only finitely many Turing machines of size n. If the problem were simply to put out, for any n, a

problem	computability	checkability	relative checkability
Meta halting	IMPOSSIBLE	IMPOSSIBLE	IMPOSSIBLE
Halting-SAT	IMPOSSIBLE	IMPOSSIBLE	HARD
Halting	IMPOSSIBLE	IMPOSSIBLE	EASY
SAT-halting	HARD	IMPOSSIBLE	IMPOSSIBLE
NPD-SAT	HARD	HARD	HARD
NPD	HARD	HARD	EASY
Restr. NPS	HARD	EASY	EASY
Compiler	EASY	IMPOSSIBLE	IMPOSSIBLE
Approx. BP	EASY	HARD	HARD
Sorting	EASY	EASY	EASY

Table E.1: Comparative difficulties of programming, checking, and relative checking for selected problems. The text explains the rating scheme, defines the problems, and justifies the ratings. The table covers all combinations of computability, checkability, and relative checkability not ruled out by the general observations given in the text.

because knowing whether the output halts is still impossible. Knowing a correct answer does not help at all with this.

Halting-SAT. This is the problem of deciding whether the input Turing machine halts on blank tape and putting out a boolean formula that is satisfiable if the machine halts and unsatisfiable if it doesn't. Obviously, this is at least as hard to compute as the halting problem, hence impossible. Moreover, it can't be checked, because it is essentially just a decision problem, i.e., the extra output gives no actual extra information. It can, however, be relatively checked because the satisfiability of the known-correct output can be determined and compared with that of the output to be checked, but at a cost of NP-hardness.

Halting problem. Well-known to be uncomputable, the halting problem is also impossible to check because it is a decision problem. It is trivial to check relatively, however, since it is a single-valued function.

SAT-halting. If the input propositional formula is satisfiable, put out a Turing machine that halts on blank tape; otherwise put out one that doesn't

program to decide halting on all inputs then the program would be trivially incorrect on all inputs, hence checking would be trivial.

halt on blank tape. This is obviously computable, but NP-hard. Checking and relative checking are impossible, however, since the halting problem is undecidable.

NPD-SAT. Take any NP decision problem. On any input decide it, then put out two values: first, a flag indicating the decision; and second, a satisfiable boolean formula if it was true, otherwise an unsatisfiable formula. This is NP-hard, of course, since merely computing the first value is NP-hard for any NP decision problem. Checking and relative checking are also hard, however, since SAT is NP-hard.

NPD. This denotes any particular NP-hard decision problem. Its checkability is the same as its computability because it is a decision problem. It is relatively checkable trivially because it is single-valued.

Restricted NP search. "Restr. NPS" denotes restricted NP-search problems. NP-search problems are extensions of NP-complete decision problems where the program must put out a polynomial-size "proof" of positive instances. For example, for SAT-search the program must give a satisfying assignment for positive instances. Restricted NPS problems are ones where the input is known to be a positive instance. Any restricted NPS problem must be hard to compute, because if there were a polynomial time program for it taking $p(n)$ steps, then we could solve the corresponding unrestricted NPS problem in polynomial time as well by running the program for time $p(n)$ and then either returning FALSE if it hadn't halted or checking its result (which we can do quickly) and reporting the result of our check. Restricted NPS is easy to check because any answer comes with a polynomially-checkable proof. Restricted NPS problems are easy to check relatively because they are easy to check absolutely.

Compiler problem. This is the problem of constructing a low-level machine code program equivalent to a given high-level source program. It is easy to compute in polynomial time for most standard languages. Computation theory shows, however, that it is impossible to write a program that can tell for every (low-level program, high-level program) pair whether the high-level program computes the same function as the low-level one. Moreover, knowing one low-level implementation of a given high-level program won't help at all in checking another; thus, relative checking is impossible as well.

Approximate Bin-packing. This is the problem of finding a bin packing within a factor of two of optimal. Approximate bin-packing is easy to compute via a first-fit strategy. It is hard to check because easy checking would

imply knowing the optimal length, which in turn would imply being able to solve the bin-packing decision problem, which is NP-complete. It is forced to be difficult to check relatively because its checkability is no harder than the maximum of its computability and its relative checkability.

Sorting. Sorting is easier to check ($O(n)$) than to compute ($O(n \ lg \ n)$, but only just barely; both are easy.

Appendix F

Series Subtleties

There is a subtlety raised by the fact that series expressions as defined by Waters behave according to *lazy semantics*, which is not always equivalent to the eager evaluation semantics assumed by my system. The difference arises when series operations are allowed to have side effects on the store: side effects of a causal successor may effect some of the series elements of a causal predecessor (creating an implicit flow arc cycle in the program). Figure F.1 gives an example of a program that is compiled into a loop that does not obey eager semantics.[1] The forward pointing DATA (field-0) pointer of cell c0 causes COLLECT-SETNEXT to alter the NEXT (field-1) value of cell c1 before SCAN-NEXTS has accessed it. This causes SCAN-NEXTS to put out a length-two series instead of the length-three series predicted by an eager interpretation. Note that this is not a defect in the Series Macro Package; there is no way to compile all series expressions with side-effects into efficient, eager loops. Moreover, it can even be difficult to determine whether the resulting loop will behave consistently with eager semantics.

The implemented system assumes eager semantics everywhere, so it is theoretically possible for a proposed redistribution to be correct when the

[1]The particular lazy semantics obeyed in this example is determined by the particular low-level implementations of the SCAN-NEXTS and COLLECT-SETNEXT primitives. For those interested in running this example using Waters's macro package, SCAN-NEXTS is implemented as SCAN-SUBLISTS, NDATA is implemented as S:CAR, and COLLECT-SETNEXT is implemented as

```
(DEFUN COLLECT-SETNEXT (Z)
  (DECLARE (OPTIMIZABLE-SERIES-FUNCTION))
  (COLLECT-NCONC (MAP-FN T #'(LAMBDA (ELT) (WHEN ELT (SETF (CDR ELT) NIL)) ELT) Z)))
```

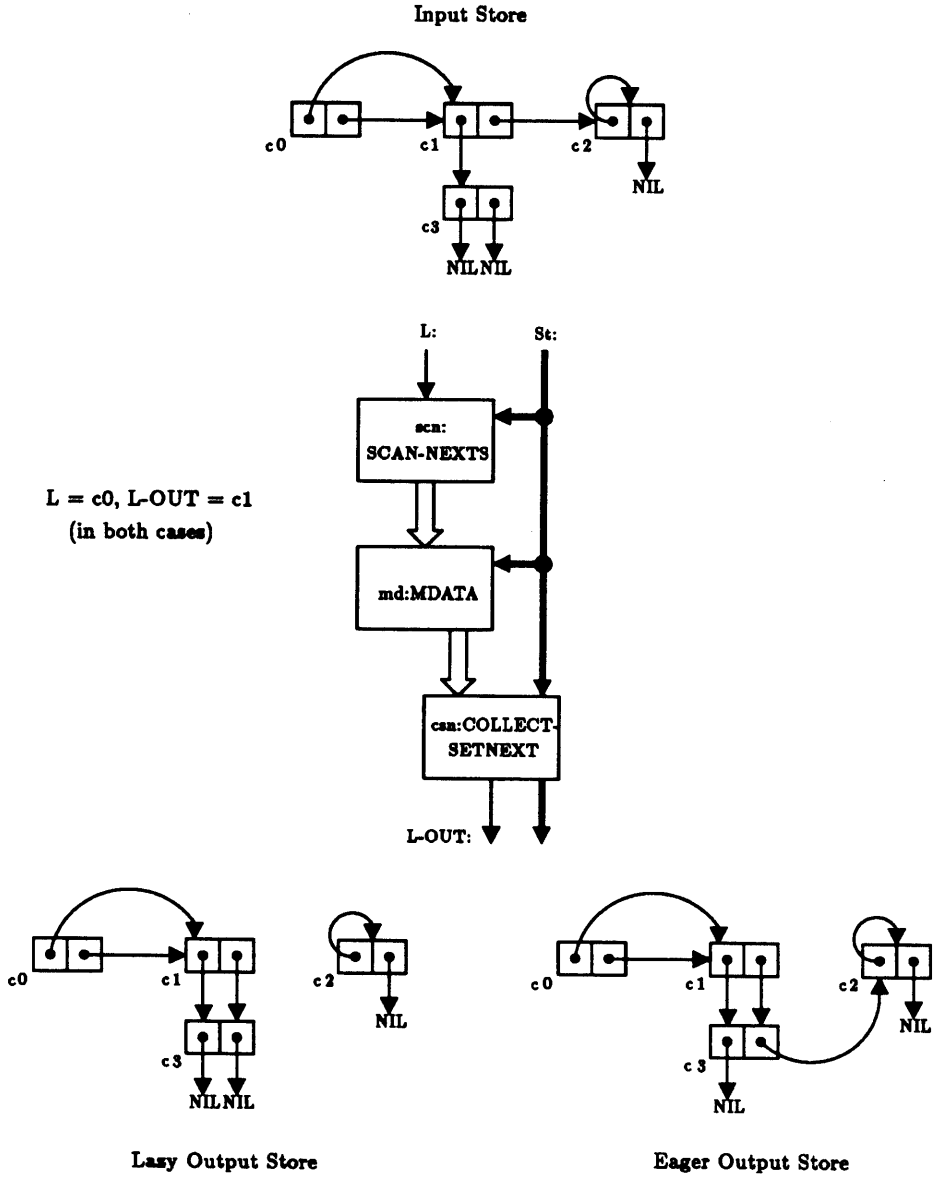


Figure F.1: A program whose compilation by the Series Macro Package does not preserve eager operational semantics. `L-OUT` is `c1` in either semantics, but the output stores are different as shown. `SCAN-NEXTS` returns a series of the cells in the input list; `MDATA` selects the DATA fields; and `COLLECT-SETNEXT` destructively assembles its input's elements into a list, ignoring non-cell elements.

resulting program is interpreted according to eager semantics, but be incorrect when compiled into a loop conforming to the lazy semantics of series expressions.

F.1 The Current Approach

The current system can rule out this problem by an extra syntactic check, highly dependent on the choice of computational primitives, in addition to those given in Section 3.2.3. I have not implemented it, but it is straightforward. Interestingly, none of the redistributions passing the screening phase of either algorithm would have failed this check. Note that when applying the check, the program is viewed flat with respect to series operations; that is, all non-primitive boxes with series inputs or outputs are opened to expose implementations.

The intuition behind the following check is to conservatively rule out all situations where some destructive operation could possibly alter some datastructure before it is accessed by a series primitive. For this set of primitives, the only destructive series operation is `COLLECT-SETNEXT`, so I will be ruling out situations where its use could possibly change a field value before it is accessed by some consumer primitive within the same enclosing series subprogram (see below).

Eager semantics preservation check:

- The only series primitives allowed in a program are `SCAN-NEXTS`, `COLLECT-SETNEXT`, `MDATA`, `MSNGLTN`, `CATENATE`, `COTRUNCATE`, `CHOOSE`, `MSBIFEQ`, `MARKER`, `MEQ?`, `MNOT?`, `ZN<K`, `ZN<=K`, `COLLECT-AND?`, `COLLECT-OR?`, `COLLECT-SUM`, `COLLECT-LAST`, `SERIES`, and `FSERIES`.

- For every `COLLECT-SETNEXT` box c in the program, let its *enclosing series subprogram* be the collection of all program elements (boxes and flow arcs) connected to c by some directed path of series flow arcs ending at c. Then the enclosing series subprogram of c can contain no more than one `SCAN-NEXTS` box s_c, and every maximal series flow arc path ending at c must pass through one of the following classes of ports after passing either through a `CATENATE` box or any box taking a non-series (or series-of-stores, if such exist) input:

- the output port of a **MSNGLTN** box;
- the output port of s_c, if it exists; or
- the **ZBOOL** input port of a **CHOOSE** box.

Note that we treat **COTRUNCATE** boxes somewhat specially: all paths of interest entering the box through the **Z1** port exit through the **cotruncate-z1** port and, similarly, all paths of interest entering through **Z2** exit through **cotruncate-z2**.

- $s_{c_1} = s_{c_2} \Rightarrow c_1 = c_2$, for all **COLLECT-SETNEXT** boxes c_i.

One can see that this check preserves eager semantics as follows. It is enough to guarantee that no cell is destructively modified before its **NEXT** field is read by **SCAN-NEXTS**. This is because **SCAN-NEXTS** is the only series operation that accesses this field. The only destructive primitive operation is **COLLECT-SETNEXT**. The cells in the input series to such a box originate in one of two types of places: in one case, the cell comes out of a **MSNGLTN** box, in which case the cell is fresh and so cannot be found by the **SCAN-NEXTS** box corresponding to the **COLLECT-SETNEXT** box; alternatively, the cell is produced by the **SCAN-NEXTS** box itself, processed by some sequence of the primitives **COTRUNCATE** and **CHOOSE**, then passed to **COLLECT-SETNEXT**. In the latter case, the cell must already have been seen by **SCAN-NEXTS** and will not be visited again, since it was originally found by **SCAN-NEXTS** and the allowed primitives can only delay a cell's appearance at the input to **COLLECT-SETNEXT**, never accelerate it.[2] **SCAN-NEXTS** is implemented so that the *current* cell has its **NEXT** pointer read and stored prior to the cell itself passing on to be processed, so that altering the **NEXT** field of the current cell does not interact with **SCAN-NEXTS**.

The set of allowed primitives can be extended based on this reasoning in several ways. For example, any primitives that do not alter the store can be added as long as the rest of the check is not changed. Addition of other destructive operations would require significant thought, however.

The third condition simply guarantees that a given series subprogram cannot have two or more **COLLECT-SETNEXT** boxes destructively modifying

[2]This is true as long as the input cons structure to **SCAN-NEXTS** is not circular. If it is circular, the eager interpretation is undefined, so I assume it does not matter whether the behavior is preserved.

series of cells possibly containing some of the same members. It may be possible to relax this as well, if other checking is done.

F.2 A More General Approach

Even though I believe it is possible to express many useful implementations in terms of a relatively limited set of series primitives, in which case a simple syntactic check is both feasible and desirable, it may nevertheless be necessary in the future to allow more liberal sets of series primitives, ones for which there is no simple syntactic check to guarantee eager semantics preservation. All is not lost in that case, as IBR and IEBR can be reimplemented as follows to handle the "lazy" semantics of fully general destructive series expressions.

Inv-Screen's only change is that when executing a test case it must obey "lazy" semantics with respect to series values, in a manner consistent with Waters's definition. This is conceptually simple to incorporate into the program evaluator, since it amounts to little more than adding a call to Waters's macro package.

IEBR need not be changed at all (except insofar as **Inv-Screen** must be changed as just discussed); the input proofs need only be different. Instead of proving the specification of a program using eager semantics (as currently reflected in the legality conditions of `:PROGRAM-STRUCTURE` nodes), prove it using lazy semantics. This would presumably require changing the form and legality conditions of `:PROGRAM-STRUCTURE` nodes, as well as altering the underlying logical domain theory to reflect lazy semantics.

Since lazy semantics is more complex and less intuitive than eager[3], it may be more desirable for a proof to assume eager semantics, but also incorporate a subproof showing that the program actually obeys eager semantics. Of course, **Inv-Screen** would still need to implement lazy semantics in program evaluation.

Compile-time certifiers must, of course, take into account the semantic interpretations of series objects and operations in checking redistributions. I envision that the typical way of using series will be to reason in terms

[3]Programs obeying lazy operational semantics must be thought of operationally—in terms of which series elements are generated and used when—whereas programs obeying eager semantics can be viewed simply as composed mathematical functions.

of eager semantics;[4] the system then must only avoid making changes to the program that cause it to violate eager semantics. (This is as opposed to frequently writing programs whose behavior depends on lazy semantics.) This will involve somewhat more complexity in screening (as outlined above), but the only "lazy" reasoning required of the compile-time certifier will be showing that the program still obeys eager semantics. This would appear to be a smaller change to the certifier than extending it to handle arbitrary reasoning in the more complicated lazy semantics.

[4]... except possibly with respect to input/output behavior which may be best thought of operationally.