INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the original text directly from the copy submitted. Thus, some dissertation copies are in typewriter face, while others may be from a computer printer.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyrighted material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is available as one exposure on a standard 35 mm slide or as a 17" × 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6" × 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
The role of verification in software reusability

Krone, Joan E., Ph.D.

The Ohio State University, 1988
THE ROLE OF VERIFICATION IN SOFTWARE REUSABILITY

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

BY

Joan Krone, B.S., M.S., M.S.

* * * * *

The Ohio State University

1988

Reading Committee: Approved By

Dr. W. F. Ogden
Dr. Sandy Mamrak
Dr. Neelam Soundararajan
Dr. Bruce Weide

Computer and Information Science
Dedicated to my father,

Stephen L. Neuhart
Acknowledgement

I want to express sincere appreciation for the support and encouragement of my committee members and for the active interest of my fellow graduate students, particularly the reusability group. I am grateful to my adviser for his example of professional integrity. Finally, I want to thank Matthew and Gil for their support and sense of humor.
Vita

Jan., 1985 - present  coordinator of a course in
computer literacy (12 sections)
taught systems programming and
data structures

June, 1984 -
Jan., 1985  taught courses in data structures and
computer literacy

April, 1983 -
June, 1984  RA; worked on $5 million proposal;
arranged 2 day conference on CAD/CAM

1963 - 1965  taught high school math

1967 - 1969  math instructor at Ohio Dominican College
taught every math course at least once

Sept., 1970 -
Jan., 1983  chairperson of Math Department at Ohio
Dominican College

1970  promoted to assistant professor

1972  granted tenure at ODC

1977 - 1981  headed computer committee. Results:
ODC purchased a computer and started
a program of study in computer science

1978 - 1983  helped acquire a FIPSE grant for the
Central Ohio Consortium

summer 1981  wrote programs to perform statistical
analysis for a project funded by the state
of Ohio

fall 1981  wrote programs for the ODC office of
1980 helped plan a tv course in computer literacy

**Fields of Study**

Major: Foundations of Computer Science  
Minor: Programming Languages  
Minor: Operating Systems
# TABLE OF CONTENTS

Acknowledgement .................................................................................... iii
Vita ............................................................................................................ iv

I. INTRODUCTION ........................................................................ 1
   1. Problems with Existing Software ........................................ 1
   2. A Multi-Objective Approach as a Solution .................... 5
   3. Language Issues .......................................................... 9
      1. Language Support for Types ........................................ 12
      2. Language Support for Specifications ....................... 14
   4. Verification in Perspective ............................................. 16
   5. A New Approach to Verification ...................................... 18
   6. Organization of the Thesis ............................................... 19

II. SPECIFYING AND VERIFYING IN REFORMS .................. 21
   1. Notation ......................................................................... 24
      1. Example of Straight Line Code ................................ 24
      2. Example of Iteration ................................................. 29
      3. An Example to Illustrate Procedures ....................... 32
      4. Notational Support for Syntax-Driven Verification .... 40
   2. Semantics ....................................................................... 41

III. TCOLS FOR SPECIFICATION OF MODULES .................... 44
   1. Object Oriented Programming ........................................ 44
   2. Module Structuring for Reusability ................................ 47
      1. Module Layering in REFORMIS ................................ 48
      2. Stack Example .......................................................... 48
      3. Type Creation ............................................................ 56
      4. Another Layered System ........................................... 59
   3. Comments about Primitive Modules ............................... 60

IV. RULES FOR VERIFYING BASIC MODULES ....................... 63
   1. The Conceptual Module Declaration Rule .................. 64
   2. Realization Module Declaration Rule ......................... 68
   3. Module Instantiation Rule ........................................... 76
   4. Variable Declaration Rule ............................................ 78
   5. Using the Rules ............................................................ 79
   6. Applying the Module Rules to the Stack Example ........ 80
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Observations</td>
<td>90</td>
</tr>
<tr>
<td>7. Enhancements</td>
<td>90</td>
</tr>
<tr>
<td>8. Primitive Modules</td>
<td>92</td>
</tr>
<tr>
<td>1. General Comments</td>
<td>92</td>
</tr>
<tr>
<td>2. Bootstrapping</td>
<td>93</td>
</tr>
<tr>
<td>3. A Boolean Module</td>
<td>94</td>
</tr>
<tr>
<td>9. A Hierarchy of Primitives</td>
<td>95</td>
</tr>
<tr>
<td>1. Integers</td>
<td>95</td>
</tr>
<tr>
<td>2. Characters</td>
<td>97</td>
</tr>
<tr>
<td>3. Strings</td>
<td>98</td>
</tr>
<tr>
<td>4. Records</td>
<td>98</td>
</tr>
<tr>
<td>5. Arrays</td>
<td>99</td>
</tr>
<tr>
<td>10. Primitive Library</td>
<td>100</td>
</tr>
<tr>
<td>V. COMMUNAL MODULES</td>
<td>101</td>
</tr>
<tr>
<td>1. Specifying Communal Modules</td>
<td>102</td>
</tr>
<tr>
<td>2. A Communal Stack Module</td>
<td>104</td>
</tr>
<tr>
<td>3. A Shared Storage Implementation</td>
<td>106</td>
</tr>
<tr>
<td>4. Enhanced Proof Rules</td>
<td>107</td>
</tr>
<tr>
<td>5. The Role of Adjunct Variables in Correspondences</td>
<td>110</td>
</tr>
<tr>
<td>6. Formal Treatment of Adjunct Variables</td>
<td>116</td>
</tr>
<tr>
<td>VI. ADJUNCT VARIABLES AND TOTAL CORRECTNESS</td>
<td>117</td>
</tr>
<tr>
<td>1. Total versus Partial Correctness</td>
<td>118</td>
</tr>
<tr>
<td>2. Termination of While Loops</td>
<td>119</td>
</tr>
<tr>
<td>3. Completeness</td>
<td>126</td>
</tr>
<tr>
<td>4. Computational Histories</td>
<td>128</td>
</tr>
<tr>
<td>1. Use of Strings</td>
<td>129</td>
</tr>
<tr>
<td>2. How Programs Look</td>
<td>130</td>
</tr>
<tr>
<td>3. Notation and Terminology</td>
<td>131</td>
</tr>
<tr>
<td>5. Application of the Termination While Rule</td>
<td>141</td>
</tr>
<tr>
<td>6. Recursive Procedures</td>
<td>141</td>
</tr>
<tr>
<td>VII. NESTED LISTS</td>
<td>152</td>
</tr>
<tr>
<td>1. Motivation</td>
<td>152</td>
</tr>
<tr>
<td>2. Tree Theory</td>
<td>154</td>
</tr>
<tr>
<td>3. Nested Lists</td>
<td>155</td>
</tr>
<tr>
<td>1. Conceptual Module</td>
<td>174</td>
</tr>
<tr>
<td>2. A Realization Module</td>
<td>176</td>
</tr>
<tr>
<td>3. Realization Module</td>
<td>180</td>
</tr>
<tr>
<td>4. An Enhancement for the Nested List Facility</td>
<td>182</td>
</tr>
<tr>
<td>VIII. A TOTAL CORRECTNESS PROOF</td>
<td>183</td>
</tr>
<tr>
<td>1. How the Adjunct Variable is Used</td>
<td>184</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Code Storage</td>
<td>156</td>
</tr>
<tr>
<td>2. Starting Position</td>
<td>158</td>
</tr>
<tr>
<td>3. The Effect of Advance</td>
<td>159</td>
</tr>
<tr>
<td>4. Open_sublist Example</td>
<td>160</td>
</tr>
<tr>
<td>5. Insert Example</td>
<td>161</td>
</tr>
<tr>
<td>6. Site S1</td>
<td>167</td>
</tr>
<tr>
<td>7. Site S2</td>
<td>168</td>
</tr>
<tr>
<td>8. j.Prec, j.Rem</td>
<td>169</td>
</tr>
<tr>
<td>9. Suture Example</td>
<td>171</td>
</tr>
</tbody>
</table>
CHAPTER I
Introduction

During the past 30 years our society has changed dramatically from one in which computer technology played a minor, almost negligible role to one which is irrevocably dependent on computers for its ordinary everyday functioning. Banks, public utilities, governments, academic institutions, and business organizations, including those which provide transportation and communication, rely on computers to the extent that, when their computer systems fail, their operations come to a virtual stand-still. Passengers are stranded in airports, production and delivery of goods and services are temporarily halted, banking transactions are delayed, and, in general, chaos reigns.

Hence, the design and development of reliable software to keep these computers performing their necessary jobs has become not just an interesting academic question, but a critical issue for the development of our civilization.

I.1. Problems with Existing Software

Computer hardware has gone through four generations in the past 30 years from vacuum tubes to transistors to integrated circuits to VLSI, and machines from the last three generations are still used in doing our applications today. There are government offices, for example, still using card readers for input to second generation computers,
while some research institutions are using the newest supercomputers, although most users have some variety of micro, mini, or mainframe from the third or fourth generation.

However, all these computer systems have one thing in common. They all depend completely on complex software to make them work. In fact, the reason many companies cling to outdated hardware is that they are dependent on certain software to perform critical tasks for them and they fear that comparable software for newer machines will be unreliable, unavailable, or unaffordable.

The issue of software cost is very prominent because, although the cost of computer hardware has decreased dramatically over the past three decades, software costs have continued to rise. Moreover while hardware has become more reliable with each generation, the same claim cannot be made about software. Experience has shown that in the life cycle of a piece of software more time is spent on program maintenance than on any other part of the cycle [7].

Maintenance includes making modifications as well as corrections to existing programs. For example, a payroll program may need to reflect a new policy such as making deductions for some charitable organization. That means a programmer, probably not the one who wrote the original program, must be able to find which part of the program needs to be changed and how to change it. This is a notoriously hard job. The available documentation (usually written in natural language), the programming language (whether it is structured or not), the form of the specifications (if any are available), the programming style, and the underlying implementation tools (built-in system features) are the basic factors influencing just how difficult maintenance is.

All of these circumstances point to one major concern of the software industry: productivity. Today there are more people designing and developing software than
ever before, but the fact that many of these people are spending large portions of their
time on maintenance is a symptom of the lack of adequate tools for programmers to
use in designing and developing software efficiently or effectively.

Those who do take up the challenge of creating software have some useful tools,
including high level structured languages, various design methodologies, a collection
of debugging tools and testing techniques, and a few sage writings distilling some of
the lessons learned in early software development experiences. However, while all of
these tools have helped programmers to some extent, with the exception of high level
languages, existing tools have failed to increase software productivity significantly.

When we look to see what the problems are, we find that much effort in
programming goes into writing software that closely resembles in form or function
software which has been written numerous times previously. Every project starts over
from the ground up, and very little software is reused. This is incredibly inefficient
and totally ignores the principle technique used for encouraging productivity across
engineering, that of reusing parts already available.

A second problem arises because in many shops, programmers provide no
documentation with their programs, or, at best, only a few cryptic comments inserted
at various places in the code. Although other programming organizations require
specifications for their programs, the specifications are usually written in natural
language either in free prose or in a form suited to some particular syntax. As a result,
when someone needs to modify a program segment or wants to use an existing
program segment, it may be difficult to tell exactly what the code does and what
restrictions there might be on the data for which it will work properly. Hence,
programmers spend inordinate amounts of time figuring out what program segments
do, deciding whether they can handle unusual data, discovering why they process
certain data incorrectly, and deciding whether they can be reused elsewhere. Equally
difficult are the questions of how to modify a given piece of code to fit new requirements, what changes may occur elsewhere as a result of such changes, and whether it is possible to give a precise and comprehensible description for the behavior of the resulting code.

Still another difficulty is that many programs, even those which have been extensively tested, fail during use. In general, determining whether or not a program is correct has proven to be a troublesome challenge. To address this issue, some programmer teams perform random testing, while others follow some prescribed regimen of boundary value checking. But, although program testing may serve to establish some confidence in a given program, except for very restricted classes of programs, there are no proofs that, for any given program, testing a particular data set can guarantee the correctness of that program.

What has been done to address these problems? During the past two decades, experts have been giving advice about how to write better programs [20, 29, 7, 13, 1, 15, 18, 21]. They tell implementors to take advantage of structured languages, to break up their programs into modules, to hide implementation details, to provide accurate specifications telling precisely what the program does, to write "reusable" code, to test the program to see if it does what it is supposed to do, and in addition to all this, to make the programs efficient.

However, despite the fact that software engineers have received this advice for many years, software continues to be poorly specified, hard to maintain, rarely reusable to any extent, and in general, not totally reliable.

This thesis will argue that one of the primary reasons we are not following the sage advice is that current programming languages and tools are simply inappropriate for the task. The inadequacies of current tools become obvious once the new tools presented have been investigated.
1.2. A Multi-Objective Approach as a Solution

In any discipline, research into a new area usually proceeds by tackling only selected parts of a broad problem while making certain plausible assumptions about, but otherwise ignoring, other aspects of the problem. For the past two decades people interested in the problem of increasing software productivity have carried out numerous research projects of this nature [42, 44, 61, 24, 6], some concerning themselves with software specifications, others with modularity, some looking at language support, others at efficiency, some developing logic to prove correctness, others striving to create reusable code.

There is a profusion of specialized research results, yet the software problems remain. To achieve one goal, such as efficiency, programmers have been forced to sacrifice others such as easy modifiability.

The most positive outcome of all this research is that we can now articulate what we would like to see in our software, namely that all the goals individually identified should be achieved in a single unified system. We want a complete system for creating programs and we want this system not only to support but to encourage the following characteristics:
• Software should be really reusable.
• Software should do what its specifications indicate.
• Software should be easy to maintain.
• Programs should run predictably efficiently.
• Programs should be easy to read.
• The system should be easy to use.

The first two goals, verification and reusability, seem inseparably linked. Certainly no one wants to reuse a program unless it does what it’s supposed to do, and the cost of verification is too great to justify unless it can be amortized over a number of reuses of the verified program as a component in various systems. It is these two goals which I have made the primary focus of my thesis.

At the same time, all the research leading to the results presented here was carried out with the intent of achieving all the other goals listed for good software, ignoring none, and with no intent of sacrificing one goal in order to achieve another.

To elaborate on the other goals, saying that a program is easy to maintain means that a programmer can tell what it is doing, so that it is comparatively simple to make modifications. Moreover, these modifications should cause no surprises. For example, if a programmer changes the way a payroll package computes dependency deductions, there should not be an unintended change in the way gross income is computed.

Every programmer recognizes the need for efficiency and strives toward achieving this goal. However, although complexity theory provides methods for assigning measures of complexity to a given piece of code, these methods are not generally used to assess the performance of large systems. The temptation exists to assume that if each of the components of a large system is separately tuned for efficiency, then the whole system will run efficiently, but experience has shown this assumption to be unfounded. Since software engineers do agree that large systems
must be built from judiciously selected modular components, what we propose is that these components include performance specifications as part of their documentation. Thus each program component such as a procedure or iteration statement would include a syntactic slot for recording its alleged performance specifications. The correctness of these specifications for larger components could then be proved based on the specifications of their constituent components.

Performance considerations also dictate that the language we use for programming not introduce built-in inefficiencies. In a later section, for example, we will see how parameter passing, as most languages perform it, introduces a substantial inefficiency when applied to the large objects that commonly occur in current programming practice. I will also propose a solution to this problem.

Having at first been forced to write all their programs directly in machine code, early programmers welcomed mnemonics provided by the first assemblers as remarkably easy to read and therefore a great enhancer of productivity. Shortly thereafter, Fortran and Cobol were developed, and eventually, literally dozens of high level programming languages emerged, each having features, such as the ability to include comments, which helped to make programs easier to read. The most recent of these languages, the so-called structured languages, such as Pascal and Ada, offer increased support for data types and for modularizing. However, even these new languages do not encourage programmers to include precise statements concerning what each section of the program does, nor do these languages promote information hiding (specific examples will be given later) to the degree that a programmer can tell what a section of code does without looking at implementation details.

It is a general observation about software development that a programming language and the constructs it provides govern to a large extent how easy it is to read programs. We insist that a language should support modularity, should provide
syntactic slots for specifications, and should encourage effective separation of
cconcerns, i.e., information hiding.

Easy to read and easy to use may sound like synonymous phrases, and certainly a
programming language which is easy to read is easier to use than one which is not.
However, being easy to read is not enough in itself to guarantee ease of use.

The issue of user-defined types, for example, illustrates this point. All existing
languages have what are called "built-in" types but the particular collection provided
varies from one language to another. Most languages include booleans, characters,
integers, and reals as built-in types, and most provide operations to apply to these
types.

When a user defines a new type, however, the new type must be treated
differently from the built-in types. For example, one can multiply two reals or apply
the "and" operator to two booleans, without knowing anything about the
implementation details. But, if a programmer wants to check to see if two stacks are
equal, he must know the implementation details in order to make the comparison. In
general, the fewer inconsistencies, the fewer exceptions that a system has, the easier
the system will be to use. In this thesis I will present a homogeneous treatment of
types which extend from the basic built-in types up to any arbitrary sophisticated type
that a user may introduce.

It is only when all these goals are addressed by a single system that software
productivity will show significant improvement. But, as we shall see, this
multi-objective approach necessitates the introduction of some new mechanisms, all
designed with enhancement of productivity in mind.

In order to validate the concepts presented here, I felt it was necessary to
introduce a fairly sophisticated reusable data structure which would serve as a useful,
non-trivial example in the context of the goals listed in this document. This new
structure had to be built hierarchically in order to provide an opportunity to illustrate information hiding. Moreover, this hierarchy needed to include both user-defined types and so-called built-in or primitive types to confirm homogeneity of treatment.

To realize this objective it was necessary to design and build such a piece of software, to develop formal specifications, to modularize carefully, and to make sure all underlying system support needed can be provided, and in so doing to identify and address some technical issues not previously solved, and to develop and apply some new proof rules.

1.3. Language Issues

A reasonable question to ask is "Why not try to verify a program written in a popular, existing language such as Modula 2, CLU, C, or Ada?" CLU has become popular as a language which provides the capability to create user defined types and to declare variables of those types, a desirable feature for programmers. However, since all variables are implemented as pointers, CLU programmers face at least one serious problem - - aliasing [36]. When an assignment, such as \( x := y \), is executed, the variable \( 'x' \) is not assigned the value of variable \( 'y' \) (the typical assignment), but rather \( x \) gets a pointer to whatever \( y \) points to. This can result in unseen, unexpected, and probably undesired effects. For example, suppose that \( x \) and \( y \) are variables which point to stacks of integers and \( i \) is an integer. Consider the following code:

\[
x := y;
\text{Push}(i,y);
\]

The assignment causes \( x \) to point to the same stack that \( y \) points to. The Push operation, then, results in pushing the integer, \( i \), not only onto the stack \( y \) points to, but also the stack which \( x \) points to.
Another language in which pointers play a major role is C. Building on C, the creators of C++ have introduced an enhancement of C which supports object oriented programming, by encouraging programmers to think in terms of classes and subclasses [54]. However, this enhancement does not alleviate the aliasing problems of C, since one way to create new types is by using an operator called "pointer to." This operator makes aliasing not only possible, but probable, thereby making it difficult for the programmer to know exactly what values the program variables have at any given point.

But aliasing is not the only shortcoming of existing programming languages. Modula 2 was developed in part to respond to a major problem of Pascal, namely the lack of language support for information hiding [11]. The introduction of the module concept made it possible for programmers to create autonomous structures in carefully delineated syntactic units. This structuring also permits separate compilation of modules, thus providing another advantage over Pascal. In standard Pascal any change to the smallest part of a program requires recompiling the whole program.

The modularization promoted by Modula 2 is a significant step toward the goal of information hiding, but it fails to support generic types. To provide generics, a programming language must permit a programmer to write a single stack module which could be used for stack of reals, stack of integers, stack of stack of integers, etc. But in Modula II, if both stack of reals and stack of integers are needed, two distinct modules must be written to create them. Moreover, if a programmer wants stacks to contain elements of other types, he must write a separate module for each stack, and these modules cannot share code, such as push and pop operations.

Ada offered generic packages to address this problem [8]. In fact, the intent of Ada designers was that by introducing generic packages, they would achieve the reusability goals of software engineering. However, although Ada does permit a high
degree of information hiding and some separation of concerns, it does not permit multiple realizations for the same type. For example, suppose a programmer writes a generic package for matrices. In the package header (the "specifications") the programmer must identify what realization (implementation) is to be used. This links the implementation to the specification, thereby precluding reuse of the same specifications for various implementations.

The requirement that a package header must include a binding to some particular implementation imposes serious obstacles to reusability. In the case of writing a matrix package, there are cases, such as those involving matrix addition or multiplication, for which an appropriate implementation is the use of a block of contiguous memory, because random access to matrix cells is achieved.

In other cases, such as those programs in which there are large matrices, say 10,000 by 10,000 in which only a few entries will be important (all but a few entries are 0), the use of a linked implementation will save enormous amounts of space worth the price of a more complicated access operation.

These two situations involving matrices require two different packages in Ada. However, in both cases, the concept of what matrices are is the same. Moreover, the user of a matrix package should not need to do his thinking in terms of implementation details, but in terms of the matrices themselves.

The languages discussed so far are representative of current, widely known programming languages which are referred to as "imperative" languages. The logic languages, Prolog being a typical example [12], focus on programming in a language which resembles the language that one uses for specifying what the program does. Programs are written in some subset of predicate logic, and since these logic statements must be executed, there are limitations on what one may write. For example, quantifiers are generally omitted or, at best, used in highly specialized cases.
Other contemporary languages, popular particularly among AI programmers, are the so-called "functional" languages, Lisp being the most popular. There are many versions and outgrowths of Lisp, but in all these functional languages, it appears that efficiency is achieved by the sacrificing of strong type checking. In fact, pure Lisp is untyped, and although many versions offer some tools for creating and checking types, the capabilities for handling types falls short of those provided by most imperative languages.

1.3.1. Language Support for Types

In this discussion about languages, the word "type" has occurred many times. It is clearly a term which evokes strong reactions from both language designers and programmers. In the literature we find that the word "type" is used to represent several different concepts which really need to be distinguished.

First, there is the mathematical concept of a type, (i.e., reals or integers) which, for clarity, we will call a *domain*. Associated with each domain is some mathematical theory. In our programs, then, a conceptual module specifications may define one (or several) computational types in terms of such mathematical domains, and that conceptual module is said to export the type(s) it defines. To support reusability, the conceptual module has a parameter list. For example, if we write a conceptual module for integers, the parameter list will probably include the maximum and minimum values that the integers may reach, hence making the conceptual type independent from its implementation.

There may also be a variety of implementations for a given conceptualization, each called a realization module and each making reference to everything in the conceptual module, including its parameter list. For example, a conceptual stack
might be implemented by an array with a top pointer or alternately a linked list. But both realizations must satisfy the specifications set forth in the conceptual stack module. To promote more reusability and to provide separation of concerns and data hiding, the realization module may have additional parameters of its own. For example, to achieve efficiency, a realization for a dictionary might import a hashing function as a parameter, but, since a binary search tree realization would not need a hashing function, this is clearly a parameter about which the conceptualization should not be concerned.

In any case, neither the conceptual parameters nor the realization parameters need to be bound until a specific instance of a given type is actually required. Using a stack again to illustrate what this means, we observe that at the conceptual level it is reasonable to expect a parameter to identify the type of the entries which will be stored in the stack. This parameter may remain unbound even at the realization level, i.e., it is possible to write and prove correct a realization for a conceptual stack module without knowing the type of the entries in the stack. The fact that this can be done is an excellent indication of just how well modularization works in the new system this thesis supports.

So far, we have seen that a single conceptual module may have multiple realizations. Similarly, each realization may give rise to a variety of specific instances, such as Integerstack realized by array with top pointer, Realstack realized by array with top pointer, or Treestack realized by array with top pointer. The precise syntax used for such declarations will be given in detail, together with examples to illustrate all these ideas, in Chapter 3, but the idea should be clear.

When such a type instance is declared, the parameters, which up until this declaration have remained unbound, are fixed. To use another illustration, suppose that the conceptual module and some realization module for a queue have been built.
When the user needs an instance of a queue, say one in which the entries are to be stacks of integers, he declares such an instance, and in so doing, fixes the parameters, including the one which identifies the type of queue entries.

Once a type instance has been declared, then any number of variables of that type may be declared, and appropriate operations performed on them. One obvious advantage to creating variables this way is that a client programmer (one using the module) need not think of a given variable as an integer stack realized by an array with top pointer, but rather can do his reasoning in terms of whatever mathematical domain was used in defining the stack at the conceptual level, hence making it unnecessary to know any implementation details at all. Moreover, efficiency does not suffer. In fact, whatever clever methods the programmer can think of to make the operations run fast may be employed with no penalty at the abstract (conceptual) level.

Existing programming languages simply do not permit us to express the distinct notions of mathematical domains, conceptual types, multiple realizations, type instances, and variable instances which are essential for creating truly reusable software. To make progress on the subject, it is necessary to clarify these various type related notions.

I.3.2. Language Support for Specifications

Another reason for believing that new language features are needed is that, in order to establish the correctness of a program, one must have a precise statement telling what the program is supposed to do. For example, if the specifications for a program say that it computes real square roots of real numbers, the program is doomed to failure, because upon receiving a negative real, it cannot compute a real square root. Accurate specifications must say that the program computes real square roots of
positive real numbers. This obvious need for precision leads us to the conclusion that a specification language must permit the programmer to talk about whatever objects his program is manipulating, identifying exactly what restrictions the input must meet and describing precisely what constraints the output is guaranteed to meet. Since natural language fails to provide this level of precision, the need for a more formal specification language is evident.

Moreover, to make correctness proofs realistic, we believe that the task of verifying programs should be mechanizable. A natural way to automate this job is to introduce an "assertive" programming language, i.e. a programming language which provides syntactic slots for formal assertions comprising the specifications for a given program.

We have seen, so far, that current languages are inadequate for writing and verifying programs which adhere to the principles which software engineering research and experience have taught us are essential to the production of quality software. Consequently, we see the need for a completely new programming system which addresses uniformly the issues of reusability, formal specification, verification, efficiency, and modularity.

A group of researchers at The Ohio State University have undertaken the development of such a system. We call the system REFORMS: Reusable, Efficient, Formally specified and verified, Modular Software system. While some group members are addressing the problems of real time programming, others are putting their efforts into the development of user-friendly human interfaces [23, 57, 58]. Several members are investigating programming language issues such as parameter passing and syntactic shortcuts for representing functional computations. As the title of this thesis indicates, my own work has focussed on reusability and verification. My work in verification and reusability can be viewed as being supportive of and
supported by REFORMS.

In the next section I will explain my approach to verification. My approach to reusability and how to achieve it is new and encompasses much more than merely reusing executable code. My examples will illustrate this multi-faceted reusability throughout the thesis, showing the reuse of conceptual types, realizations (implementations), mathematical theories, and module instances.

1.4. Verification in Perspective

To attack the problem of program correctness formally one looks to rigorous proofs of mathematics as a paradigm. As early as 1969 Hoare triples [26] appeared on the scene, paving the way for the creation of new logic systems to deal with the various constructs of programming languages in a manner analogous to the treatment of various mathematical constructs by mathematical logic. In this thesis we will be following this school of work. However, this is not the only way people have tried to use a mathematics based approach in order to establish correctness of programs.

There is, for example, the transformation approach [4, 3, 30, 38] which is based on the idea that the program is its own specification. In this approach the question of whether or not a program meets its specifications is trivially answered, since the original program is the specification. However, to achieve efficiency, one must perform a series of transformations on the originally coded program, and the problem is to be sure that the transformations performed on the program preserve the functionality of the original code. Moreover, questions arise concerning what transformations lead to efficiency and whether or not such transformations exist. More fundamentally, it seems dubious that the original specifications are expressable in a natural enough way to make it obvious that they solve the problem at hand.
This concern leads us to examine another approach, that of writing the specifications for a program in a language which is different from the programming language, a language such as first order predicate logic which is accompanied by a calculus for reasoning about it. In this approach, first introduced by Hoare, one writes the code in a programming language and then applies a system of axioms and proof rules to determine whether or not the program meets its specifications. Many proof systems in this category have been developed, some for specific languages such as Pascal [32], many of which deal only with a restricted set of types [9, 27, 30, 45, 49, 22, 27, 28, 33, 35, 39, 43, 47, 49].

Since abstract data types have emerged as a particularly important component of most current programming paradigms, we consider still another approach to verification, the algebraic one, which is restricted to specifying and verifying data structures. Again looking to mathematics for formalism, the algebraic approach requires that specifications be written in terms of algebraic axiomatization [60, 18]. For example in specifying a stack, one axiom might be that push followed by pop is the identity. Because of the equational nature of these specifications, expressiveness may be a problem. Moreover, each new type requires the development of a new theory with all the accompanying issues such as soundness and completeness.

One term often used to categorize the above approaches to specification and verification is "definitional," because in each case an attempt is made either to define what certain abstract data types are or to define what the properties of a given program are. Another alternative is the "operational" approach in which one specifies a program's meaning by giving a computational method of constructing that meaning [60]. Reasoning is then carried out using the operational model in terms of which meaning of the program has been expressed. Since one normally chooses some existing model with which much prior work has been done, this seems, on the surface,
like a good idea. However, the models turn out to be needlessly detailed and extremely tedious to reason about [51, 52, 5, 32].

Having examined these existing approaches to specifying and verifying programs and having identified some of the unresolved problems, I propose a new approach to verification, an approach which maintains mathematical rigor without sacrificing expressiveness and which adheres to the goals of REFORMS. To validate this new approach, I have specified, built, and verified a three layered hierarchical data structure adhering to the objectives of REFORMS.

I.5. A New Approach to Verification

My approach to verification is a formal one, and it has several objectives. Although the primary goal is to provide a set of rules which are mathematically rigorous, such rules and axioms would not be useful in practical programming unless they were applicable to a language which permitted a programmer to write modular, efficient, reusable programs dealing with an appropriate variety of variable types. Indeed, a programmer should be able to define new types when they are needed and to be able to depend upon their behavior in the same way he depends upon the behavior of so-called "built-in," or system defined types. Consequently, there must be a specification language in which a programmer can describe, rigorously, concisely, and in some sense naturally, the functionality of his program.

I believe that to achieve these goals, verification mechanisms must be viewed as part of a complete programming support system. Throughout this chapter, limitations and shortcomings of current approaches to verification have been pointed out, and arguments in favor of alternatives have been made. In summary, I conclude that in the complete programming support system recommended, the specification language must
be more powerful than the programming language, and the programming language must promote a programming style which encourages both efficiency and modularity. Types must be uniformly treated and strong type checking must be enforced. The proof system must permit the addition of new rules to accommodate new constructs, if needed.

Even with all of the above characteristics, a verification system may fail to be attractive for practical use because the application of the rules is tedious and time consuming. Automation is the obvious answer, and the verification component of REFORMS is syntax-driven, thereby making it readily mechanizable.

The goals of this thesis have now been set forth and the significance of these goals has been explained. The compass of the thesis is fairly large, and the next section lays out the plan of exposition.

1.6. Organization of the Thesis

The next chapter explains how program verification works within the framework of REFORMS. There are several examples illustrating the use of the language constructs in this system and some correctness proofs showing how the rules are applied.

In Chapter 3, syntax for the specification and implementation of modules is introduced, emphasizing the impact of our approach on reusability. To illustrate these ideas conceptual and realization modules for a stack are presented.

Chapter 4 presents the rules which check whether module implementations meet their specifications. These rules are applied to the stack example from Chapter 3 to illustrate their use. In addition, primitive modules, upon which the implementation of the stack module depends, are introduced to show that there is homogeneity of
treatment of types at all levels in the system.

Chapter 5 enriches our capabilities for writing and verifying modules by making it possible to write assertions about all variables of a given type. In illustrating these new ideas, adjunct variables make their appearance. An ensuing discussion of adjunct variables and their use leads us to the issue of termination.

Chapter 6 follows up these notions of adjunct variables by introducing total correctness rules for while loops and recursive procedures.

Chapter 7 contains the specifications and a realization for Nested Lists, a new data structure with applications in editors and formula manipulation systems. The use of adjunct variables in defining the correspondence, linking the realization for Nested Lists to their conceptualization, is explained.

In Chapter 8, the total correctness rules presented in Chapter 6 are applied to the Nested List realization. The correctness proofs also show how the verification system takes advantage of the fact that supporting modules have been verified previously.

The last chapter draws some conclusions about the work presented and summarizes the contributions to software engineering in general and to the areas of verification and reusability in particular.
CHAPTER II
Specifying and Verifying in REFORMS

Specifications for programs come in a variety of forms ranging from brief, natural language statements which set forth general program requirements to precise mathematical assertions which describe the exact functionality of the program. While we accept natural language as a useful way to give informal descriptions of program behavior, in REFORMS we also require mathematical specifications of program behavior for some very practical reasons which reflect the needs of both programmers and users.

From the programmer's point of view, the more exactingly the specifications are written, the more likely it is that he will be able to produce a program which performs as desired. By the same token, from the user's perspective, when he faces a selection of existing pieces of software, he can choose the most appropriate components only when he knows precisely what functionality they provide.

In addition to these obvious reasons for using formal specifications, there is another compelling motive for requiring mathematical precision when specifying programs. The critical goal of establishing program correctness cannot be achieved unless we have formal specifications for our programs.

Since we maintain that reusability of program components is only sensible when the correctness of those components has been established, we consider the use of
formal specifications for establishing program correctness more than an interesting academic exercise. It is an essential feature of any system for producing reliable, reusable software.

Writing formal specifications for programs so that mathematical proofs of correctness are possible requires us to face some challenging issues which earlier systems overlooked or deliberately ignored. This section identifies some of those issues and tells what choices REFORMS has made regarding them.

One issue left unresolved by other systems is the provision of a proof system which is adequate to establish the total correctness of programs. For some very good reasons which will be discussed later, most existing proof systems, developed during the past two decades, are only partial correctness systems. This means that their rules allow one to prove that if the program terminates, then it is correct. The obvious drawback of such systems is that, in the case of non-termination, they are inconclusive. In response to this missing part in other proof systems, Chapter 6 introduces rules for carrying out total correctness proofs in REFORMS.

Another reality of current programming is the need for abstract data types. For the sake of simplicity most early verification systems confined themselves to programs which manipulated only integers. As the art of programming has progressed, it has become increasingly obvious that our understanding of large programs depends, to a significant degree, upon the introduction of a variety of mathematical theories to specify and explain program behavior (i.e., the use of abstract data types). In REFORMS, to deal with abstract data types, there is a layered approach, described in this chapter and the next, which provides programmers with the capability of treating all types homogeneously, from primitive, built-in types to complex, hierarchically constructed user defined types.
The choice of a specification language has a major impact on both the capability of programmers to express program functionality accurately and naturally, and the ability of verifiers to develop mathematically rigorous proof systems. But this choice is not one which should be made without giving consideration to the programming languages which will be used for coding the programs to be specified. The literature singles out three contemporary styles of programming languages and suggests a primary rationale for each style: imperative languages for efficiency (because of the fit with the Von Neumann architecture), functional languages for expressiveness (behavior is expressed in terms of mathematical functions), and logic languages for formal reasoning.

In Chapter 1, these language styles were discussed in the context of the tools they provide to programmers. Now we will look at them from the perspective of reasoning about correctness of programs written using them. Since a program cannot be labeled as correct or incorrect unless there are specifications telling what the program is supposed to do, it becomes obvious that we need to associate with any given programming language a specification language in which to write whatever assertions are necessary to describe the functionality of a program written in the programming language.

There are two possibilities: the specification language is the same as the programming language or the two languages are distinct. Most verification systems for functional and logic languages have chosen the first alternative, and, as a result, they encounter either efficiency problems or limitations on what their programs can do easily. These difficulties were discussed in a previous section.

In REFORMS we choose the second alternative, that of using one language for specifying and another for programming. In fact, the REFORMS philosophy includes the belief that not only is it unnecessary to use the same language for specifying,
coding, and reasoning, but it is undesirable to do so. Since mathematics already provides a multitude of useful, mature theories, it seems expeditious to use those theories for describing the situations encountered in programming. Logic supplies formal reasoning tools which can be enriched with proof rules covering a wide variety of programming language constructs. Moreover, imperative languages support efficiency, thereby upholding the REFORMS philosophy which suggests using whatever programming constructs lend themselves to implementations in which computations are performed as efficiently as possible.

II.1. Notation

In Chapter 1, an assertive programming language was defined as a programming language which provides syntactic slots in which programmers can write the formal specifications for their programs. In order to introduce the particular notation used in REFORMS, I will present some examples which illustrate what these assertions look like in some simple programs. In writing our programs we write the keywords of our programming language in bold face, capitalizing those which appear at the beginning of a construct, i.e. If, and using lower case for those which appear within a construct, i.e. then.

II.1.1. Example of Straight Line Code

The first example is a trivial assertive program which computes absolute values for quotients of real numbers. For simplicity, I am omitting variable declarations, but it can be assumed that \( w, y, z \), and \( \text{abs} \) have been declared to be real numbers. Later examples will deal with declarations. In fact, the whole issue of how types are defined
and variables of those types are declared is a major topic in this thesis, and an exhaustive explanation will appear in Chapter 3. However, for now, we accept the existence of types and declarations and concentrate on understanding the specifying, coding, and verifying of the more procedural parts of REFORMS programs.

Note that in our syntax, an Assume statement will be used to record what must be true at the beginning of the program in order for it to work correctly. Similarly, Confirm statement is used to record what should be true after the code is executed.

```
Assume y ≠ 0;
z := w/y;
if z ≥ 0 then abs := z
else abs := -z
endif;
Confirm abs = |w/y|;
```

This program computes a real quotient, so we must be sure that the divisor y is nonzero. The Assume clause accomplishes this. The Confirm statement claims that, upon execution of the code, the value of the variable abs will be the absolute value of the quotient w/y.

We can see that the Assume and Confirm clauses together serve to specify what the program does, by first screening out unsatisfactory input and finally by stating what will be true after program execution. In addition to providing formal specifications, these assertions permit verification by forming a basis for developing appropriate proof rules.

In the absolute value program, the constructs used are the If then else and the assignment statements, and so we need proof rules for these constructs and an explanation of what they mean. The form of typical proof rules in REFORMS is illustrated by the rule for if then else statements:
code; Assume B; code1; Confirm Q;
code; Assume ¬B; code2; Confirm Q;

The meaning of this rule (and of all future rules) is that the correctness of the bottom line can be deduced from the correctness of the top lines. The code at the beginning of each of the above lines is a sequence of statements, the first of which is usually an Assume statement. Similarly, code1 and code2 also represent sequences of statements.

To illustrate this rule we apply it to the absolute value example, but first we need a basic understanding of the overall proof process. In general, we will have a proof rule to cover each different kind of statement in our programming language, and our proof construction process will involve the creation of a succession of lemmas. This process begins with the statement immediately preceding the final Confirm statement and progresses backward through the code, applying the appropriate rule at each point. More will be said about this order of rule application later.

In our example, the code preceding the If then else statement consists of an Assume statement and an assignment. Applying the If then else rule backwards yields two assertive programs which must then be proved correct:

(1) Assume y ≠ 0;
    z := w/y;
    Assume z ≥ 0;
    abs := z;
    Confirm abs = lw/yl;

(2) Assume y ≠ 0;
    z := w/y;
    Assume ¬ (z ≥ 0);
    abs := -z;
    Confirm abs = lw/yl;

We note that when we applied this rule, the If then else construct itself disappeared, having been replaced by two hypotheses, each containing fewer
programming constructs than the original program. Since both these hypotheses contain assignment statements, we look next at the proof rule for assignments:

```
  code; Confirm Q[x -> exp];
```

```
  code; x := exp; Confirm Q;
```

As usual, the meaning of the rule is that in order to prove the bottom line, it is sufficient to prove the top line. The symbol "\(\rightarrow\)" can be read as "replaced by." This rule says that we omit the assignment statement and rewrite the Confirm clause \(Q\) replacing all the instances of \(x\) by the expression \(exp\), which was to have been assigned to \(x\). Applying this rule to our absolute value proof, we get:

1. Assume \(y \leq 0\);
   \[z := w/y; \]
   Assume \(z \geq 0\);
   Confirm \(z = lw/yl\);

2. Assume \(y \neq 0\);
   \[z := w/y; \]
   Assume \(- (z \geq 0);\)
   Confirm \(-z = lw/yl\);

We now need a rule for Assume statements:

```
  code; Confirm P => Q;
```

```
  code; Assume P; Confirm Q;
```

Applying the rule for Assume, we obtain:

1. Assume \(y \neq 0\);
   \[z := w/y; \]
   Confirm \(z \geq 0 \Rightarrow z = lw/yl\);

2. Assume \(y \neq 0\);
   \[z := w/y; \]
   Confirm \(z < 0 \Rightarrow -z = lw/yl\);

Now we apply the assignment rule to each branch:

1. Assume \(y \neq 0\);
   Confirm \(w/y \geq 0 \Rightarrow w/y = lw/yl\);

2. Assume \(y \neq 0\);
   Confirm \(w/y < 0 \Rightarrow -w/y = lw/yl\);
Another application of the Assume rule yields:

(1) Confirm \( y \neq 0 \Rightarrow (w/y \geq 0 \Rightarrow w/y = |w/y|) \);

(2) Confirm \( y \neq 0 \Rightarrow (w/y < 0 \Rightarrow w/y = -|w/y|) \);

To complete the proof we need a rule for Confirm:

\[ Q \]

\[ \text{----------} \]

\[ \text{Confirm } Q; \]

Applying the Confirm rule produces the following mathematical propositions:

(1) \( y \neq 0 \Rightarrow (w/y \geq 0 \Rightarrow w/y = |w/y|) \)

(2) \( y \neq 0 \Rightarrow (w/y < 0 \Rightarrow -w/y = |w/y|) \)

As the example illustrates, every reverse rule application produces one or more new hypotheses, each of which has fewer programming constructs than the conclusion line. Ultimately, all the programming language syntax disappears, leaving hypotheses written strictly in the language of the underlying mathematical theory. In this case that theory happens to be real number theory which allows us to conclude that both of these assertions are true from the definition of absolute value.

The part of the proof involving only the mathematical theory may strike the reader as particularly easy, and one may fear that only such obviously contrived examples as this will be so simple to verify. However, throughout this thesis, example after example will show that this phenomenon is not peculiar to this simple program, but rather is a common occurrence. In fact, we have not yet found any example in which the proof of program correctness required more than simple use of mathematical definitions and straightforward applications of the appropriate theory. This is really no surprise if one stops to consider that, in order to write the correct code, the programmer must know at an intuitive level whatever theorems underlie the reasoning he is using for program development.

This first example has illustrated the proof rules covering a few simple programming constructs in REFORMS. Appendix A gives a complete list of these
rules, but much more explanation is necessary if one is to understand what is going on in the more sophisticated part of our programming language.

II. 1.2. Example of Iteration

Since society prizes computers highly for their speed in carrying out repetitive tasks, most useful programs involve iteration or recursive procedure calls or both. To illustrate how to prove the correctness of iterative programs, we will next verify a simple program which computes the sum of the first $n$ integers.

Since a loop, although considered to be a single construct, has a body made up of a sequence of statements, the loop body has the effect of being a program itself. Hence it seems reasonable that some assertion about this "loop" program should accompany the loop.

In the verification literature, associated with every loop is a clause called the "loop invariant." As the name suggests, the loop invariant is an assertion which is true both before and after each iteration of the loop. The syntactic marker for the loop invariant is the keyword Maintaining. A simple example illustrating the while loop construct is:

Assume $n \geq 0$;
sum := 0;
i := 0;
Maintaining $i \leq n$ \wedge \sum = \sum_{j=1}^{i} j$
while $i < n$ do
  i := i + 1;
  sum := sum + i;
end;
Confirm \sum = \sum_{j=1}^{n} j

Here the invariant states that $i$ never exceeds $n$ and that at the beginning or end of any iteration, the variable $sum$ contains the total of the first $i$ integers. This exactly
describes what the loop is doing, namely computing a sequence of partial sums until it finally has the sum of the first $n$ integers. For convenience in what follows, we will name this particular loop invariant "Sum_Inv." That is,

$$\text{Sum_Inv} = \"i \leq n \land \text{sum} = \sum_{j=1}^{i} j\".$$ 

In order to verify this program, we will need the proof rule for while statements:

code; Confirm Inv;
Assume Inv $\land$ B; body; Confirm Inv;
Assume Inv $\land$ $\neg$ B; Confirm Q;

-----------------------------
code; Maintaining Inv while B do body end; Confirm Q;

Here, the first hypothesis is that the invariant $Inv$ be true before the loop is executed. The second hypothesis requires that if $Inv$ is true and the conditional $B$ for the loop is true and the body is executed, then $Inv$ is true after execution, i.e., that $Inv$ truly is an invariant. The third says that the truth of $Q$ should follow from the truth of $Inv$ and $\neg B$, since they will both hold true when the loop terminates.

We will now use this rule in establishing the correctness of the summation program. As before, we will take for granted that the variables have been declared. Since the while loop is the last construct of this program, we apply that rule first, obtaining three hypotheses:

1. Assume $n \geq 0$; sum := 0; $i := 0$; Confirm Sum_Inv;
2. Assume Sum_Inv $\land$ $i < n$; $i := i + 1$;
   sum := sum + $i$; Confirm Sum_Inv;
3. Assume Sum_Inv $\land$ $i \geq n$;
   Confirm sum = $\sum_{j=1}^{i} j$;

To prove correctness of the program, we must apply the appropriate proof rules to each of these hypotheses. For the first, applying the assignment rule to $i := 0$ yields:

1. Assume $n \geq 0$; sum := 0;
   Confirm $0 \leq n \land \text{sum} = \sum_{j=1}^{0} j$;

Applying the assignment rule to $sum := 0$ leads to:

Assume $n \geq 0$;
Confirm $0 \leq n \land 0 = \sum_{j=1}^{0} j$;
Finally, using the rule for Assume, followed by the Confirm rule, we obtain:

\[ n \geq 0 \Rightarrow 0 \leq n \wedge 0 = \sum_{j=r}^{i} j; \]

This follows from the definition of \( \Sigma \).

In hypothesis (2), the body of the loop consists of two assignments, so we apply the assignment rule twice to obtain:

(2) Assume \( i < n \wedge \text{Sum}_\text{Inv}; \)
    Confirm \( i + 1 \leq n \wedge \text{sum} + i + 1 = \sum_{j=r}^{i+1} j; \)

Applying the Assume and Confirm rules, we get

\[ i < n \wedge i \leq n \wedge \text{sum} = \sum_{j=r}^{i} j \Rightarrow \]
\[ i + 1 \leq n \wedge \text{sum} + i + 1 = \sum_{j=r}^{i+1} j \]

This can be seen to be correct by adding \( i + 1 \) to both sides of the equation for \( \text{sum} \) in the hypothesis.

Hypothesis (3) expands out to:

(3) Assume \( i \leq n \wedge \text{sum} = \sum_{j=r}^{i} j \wedge i \geq n; \)
    Confirm \( \text{sum} = \sum_{j=r}^{i} j; \)

Applying the rules for Assume and Confirm, we obtain:

\[ i \leq n \wedge \text{sum} = \sum_{j=r}^{i} j \wedge i \geq n \Rightarrow \]
\[ \text{sum} = \sum_{j=r}^{i} j \]

From \( i \leq n \) and \( i \geq n \), it follows that \( i = n \), and we have the desired result.

To write the loop invariant, the programmer needed to know that the loop will have the total of the first \( i \) integers in \( \text{sum} \) after \( i \) iterations and that the highest value \( i \) will achieve is \( n \). But, of course, had the programmer not known both of those facts implicitly, he would not have been able to write this program. The point here is that loop invariants are not mysterious, nor do they require deep mathematical insights which most programmers are unlikely to have. Loop invariants are simply descriptions of what the loop does.
II.1.3. An Example to Illustrate Procedures

The first two examples have shown how the proof rules of REFORMS are used to verify straight line code and iterations. Next we turn our attention to procedures -- declarations, calls, and recursion. The pleasant, possibly unexpected surprise is that recursion presents no special problems as far as partial correctness is concerned.

Leaving the issue of total correctness for Chapter 6, we examine syntax and partial correctness rules for procedures. As a programmer writes code, there are two ways in which procedures arise. He may want either to declare a new procedure or to call an existing procedure to work on appropriately selected parameters.

Just as a compiler must keep information about procedures so that it can do type checking of parameters and can find appropriate code into which to transfer control, the verifier must know certain information about procedures in order to be able to generate correctness proofs.

When a procedure call is made in a program, the verifier does not need to see the code for that procedure at all, but rather it must know what the code does, i.e. what its specifications are. So whenever a procedure is declared, the heading, which contains its specifications, is recorded in what we will call the program's "context."

To illustrate how procedures look, what the heading is, and where the specifications appear, we will present an example of a program fragment in which the number of permutations of \( n \) objects taken \( r \) at a time is computed. To facilitate the computation, a procedure for calculating factorials is used. The declaration of the factorial procedure is shown first:

```
Proc Find_Factorial(const n: integer, var fact: integer)
  require n ≥ 0;
  ensure fact = n!;
```
The require clause is a programmer supplied assertion which tells what restrictions must be met in order that, if the code is correct, the ensure clause will hold upon completion of the procedure body. Together, the require and ensure clauses form the specifications of the procedure. The heading of the procedure consists of the first line and the specifications:

\[
\text{Proc Find\_Factorial(const n: integer, var fact: integer);} \\
\text{\hspace{1em} require n \geq 0;} \\
\text{\hspace{1em} ensure fact = n!;} \\
\]

For any given block of code, the verifier will first look at the declarations and put appropriate information into the context, i.e., save information which will be needed in order to apply the proof rules. Then the verifier begins at the penultimate statement and proceeds toward the beginning of the program (usually an Assume statement), applying the appropriate rule as we have seen in the preceding examples.

So, in this example, the verifier will have to put the heading of Find\_Factorial into the correct context before applying the other rules necessary for proving correctness. Hence, if and when Find\_Fact is called, the specifications telling what Find\_Factorial does will be available. It is clear that we need two new proof rules, one to handle procedure declarations and one for procedure calls. The following is a simplified version of the procedure declaration rule:
Both hypotheses of the declaration rule indicate that the heading of the procedure being declared (but not its body) is to be placed in the context. This makes all the necessary information about the procedure available so that it can be called from any part of the program, including from within itself. The first hypothesis of the rule establishes that the procedure works as specified by requiring that, if the require clause is met, then the ensure clause is true upon completion of the body of the procedure. The require clause \textit{pre} is called the precondition of the procedure and the ensure clause \textit{post} is called the postcondition.

In applying the procedure declaration rule to \texttt{Find\_Factorial}, we note that the first hypothesis to consider will be:

\begin{verbatim}
C \cup \{\text{\texttt{Find\_Fact\_Heading}}\} \setminus \text{Assume } n > 0; \text{\texttt{Find\_Fact\_Body}};
\text{Confirm } \texttt{fact} = n!;
\end{verbatim}

Here \textit{C} stands for whatever context already exists at the point of declaration of \texttt{Find\_Factorial}, and as the rule shows, the \texttt{Find\_Fact\_Heading} is added to that context by the verifier. The \textbackslash{} mark separates the context from the assertive program to be proved correct. \texttt{Find\_Fact\_Body} is an abbreviation to stand for the code in the body of the \texttt{Find\_Factorial} procedure. We note that this hypothesis is in an already familiar form because it looks just like the preceding examples. Indeed, in order to establish the stated hypothesis, we proceed exactly as we did in the examples already given, and we will find that the rules we need are ones we have seen before.
Since the body of Find_Factorial ends with a while loop, it is the while rule that we apply first, thereby generating three hypotheses to check:

1. Assume $n \geq 0$; fact := 1; i := 0;
   Confirm fact = i! \land i \leq n;
2. Assume fact = i! \land i \leq n \land i < n;
   i := i + 1; fact := i \ast fact;
   Confirm fact = i! \land i \leq n;
3. Assume fact = i! \land i \leq n \land i \geq n;
   Confirm fact = n!;

For (1) we apply the assignment rule twice, obtaining:

1. Assume $n \geq 0$; Confirm 1 = 0! \land 0 \leq n;

Applying the rules for Assume and Confirm, we get:

$n \geq 0 \Rightarrow 1 = 0! \land 0 \leq n$;

The fact that $1 = 0!$ is a definition from number theory.

Hypothesis (2) also requires two applications of the assignment rule and use of the rules for Assume and Confirm. The result is:

fact = i! \land i \leq n \land i < n \Rightarrow
(i + 1) \ast fact = (i + 1)! \land i + 1 \leq n

The first part of the conclusion can be proven by multiplying both sides of the equation fact = i! by i + 1. The other part of the conclusion is obvious since i < n.

For hypothesis (3) we need to verify:

3. Assume fact = i! \land i \leq n \land i \geq n;
   Confirm fact = n!;

Applying the assume and confirm rules yields:

fact = i! \land i \leq n \land i \geq n \Rightarrow
fact = n!;

Since $i \leq n$ and $i \geq n$, $i = n$. Hence fact = i! implies that fact = n!.

We have seen that an application of the proof rule for procedure declarations causes the context to be enriched with the procedure heading. Such an application also establishes that if the parameters to the procedure meet the require clause, then upon completion of the procedure body, the ensure clause is met.
Next we will see how procedure calls work by examining a call to Find_Factorial. The following program fragment computes the number of permutations of $n$ objects taken $r$ at a time:

```
Assume $n \geq r \geq 0$;
Find_Factorial(n, nfact);
d := n - r;
Find_Factorial(d, dfact);
Perm := nfact/dfact;
Confirm Perm = \binom{n}{r};
```

To show correctness of this program fragment, we will need to apply the assignment rule first, followed by two applications of the call rule with an assignment between the two calls.

The following is a simple version of the proof rule for procedure calls:

```
\text{Confirm } \pre{x \rightarrow a} \land \forall ?a: \text{T, post}[\#x \rightarrow a, x \rightarrow ?a] \implies Q[a \rightarrow ?a]
```

Since the declaration rule establishes that if the require clause is satisfied, then the ensure clause is met upon completion of the procedure body, the first hypothesis of the call rule checks that the require clause $\pre{}$ holds when the actual parameter $a$ is substituted for the formal parameter $x$. The second hypothesis asks that the ensure clause $\post{}$ imply $Q$. The $\#$ sign in front of the variable $x$ refers to the value of $x$ at the beginning of the procedure. This distinguishes the old value of $x$ from the current value. The extra complication here is the introduction of a new variable $?a$ to stand for the value of the actual parameter $a$ after the procedure $p$ has modified it.

The call rule is simpler to understand in the version presented above, but in normal usage we don't want to break out two separate hypotheses when a single, slightly lengthier one will do. So we ordinarily combine this rule in a single hypothesis rule:
With this rule available to us, we are ready to establish correctness of our program fragment. Since the last executable statement in our fragment is an assignment, we apply the assignment rule first, obtaining:

\[
\text{C } \text{Assume } n \geq r \geq 0; \\
\text{Find } \text{Factorial}(n, nfact); \\
d := n - r; \\
\text{Find } \text{Factorial}(d, dfact); \\
\text{Confirm } nfact/dfact = nP_r;
\]

Next we need the call rule:

\[
\text{C } \text{Assume } n \geq r \geq 0; \\
\text{Find } \text{Factorial}(n, nfact); \\
d := n - r; \\
\text{Confirm } d \geq 0 \land \forall \ ?dfact: \text{integer}, \ ?dfact = d! => \\
nfact/dfact = nP_r;
\]

Applying the assignment rule results in:

\[
\text{C } \text{Assume } n \geq r \geq 0; \\
\text{Find } \text{Factorial}(n, nfact); \\
\text{Confirm } n - r \geq 0 \land \forall \ ?dfact: \text{integer}, \ ?dfact = (n - r)! => \\
nfact/dfact = nP_r;
\]

Another application of the call rule yields:

\[
\text{C } \text{Assume } n \geq r \geq 0; \\
\text{Confirm } n \geq 0 \land \forall \ ?nfact, \ ?nfact = n! => \\
n - r \geq 0 \land \forall \ ?dfact: \text{integer}, \ ?dfact = (n - r)! => \\
?nfact/dfact = nP_r;
\]

Upon applying the rules for Assume and Confirm, we get an implication:

\[
n \geq r \geq 0 => n \geq 0 \land \forall \ ?nfact, \ ?nfact = n! => \\
(n - r \geq 0 \land \forall \ ?dfact: \text{integer}, \ ?dfact = (n - r)! => \\
?nfact/dfact = nP_r);
\]

Since \( n \geq r \geq 0 \), \( n \geq 0 \). Since \( ?nfact = n! \) and \( ?dfact = (n - r)! \), by a definition in combinatorics, \( ?nfact/dfact = nP_r \), since \( nP_r = n!/(n - r)! \). Hence our program fragment is correct.
The two examples we have just presented performed calculations on integers in a programming style which is typical for manipulating small objects, i.e., a constant parameter was passed to a procedure which performed some calculations and then assigned the resulting value to a variable parameter. This is typical and reasonable because small objects take up little space, and psychologically it seems reasonable to have a new variable to represent the result of the operation while the original variable retains its value. In the case of \textit{Find\_Factorial}, the constant parameter \(n\) remained fixed while the variable parameter \textit{fact} had a new value after the computation was completed. Our simplified procedure declaration rule was sufficient for proving the correctness of procedures written in this style.

However, this programming style is not desirable when programming with large objects, or even small ones in some cases. For example, in sorting the elements of an array, a programmer would not want to create a new array each time a permutation of the given elements is made. In order to consume both time and space, it is preferable to modify the existing array. This means that there must be a way to specify what modification has been made, and this requires us to reference values of variables both before and after the change takes place. To accommodate this need we have introduced the use of the symbol "\#" to be placed in front of a variable \textit{var\_name} to indicate that its value at a given point in the program must be remembered as \#\textit{var\_name}.

To illustrate the use of this idea, consider a trivial procedure which increments a given integer by a constant value. Here, even though it may not require a lot of space to create a new integer to hold the incremented value, psychologically we expect the given variable to have its old value destroyed and replaced by its incremented value. This procedure might be part of an integer package:
procedure Increment(var n: integer, const c: integer);
  require min_int ≤ n + c ≤ max_int;
  ensure n = #n + c;
  n := n + c;
end;

The require clause makes sure that the result is within the bounds permitted for integers. The ensure clause states that the new value of n will be the old value of n incremented by c.

Verification of procedures whose clauses refer to old values requires us to supplement our declaration rule as follows:

C ∪ \{p_heading\} ∖ Remember; Assume pre; body; Confirm post;
  code; Confirm Q;
---------------------------------------------------------------
C \Proc p(var x, const y); require pre; ensure post;
  code; Confirm Q;

The new keyword which appears here indicates that values of all variables are to be saved as of this point in the program. The proof rule for Remember is:

C \code; Confirm Shift(Q);
---------------------------------------------------------------
C \code; Remember; Confirm Q;

where Shift(Q) is defined by:
  Shift(x) = x
  Shift(c) = c for any constant c
  Shift(#x) = x
  Shift(f(e1,e2)) = f(Shift(e1), Shift(e2))

The Shift function removes one pound sign from any given variable if one is there.

At this point, we can now make some general observations about how our proof system works. Although the programmer must supply certain clauses specifying program behavior, the generation of intermediate assertions is mechanical. An automated verifier can determine syntactically what rule to apply at each step of the proof. Moreover, when all the pertinent rules have been applied, the statements which remain are ones involving only the mathematical theory for the given program, and the
proof reasoning can then be completed using only traditional mathematical reasoning. I will comment further on the reasoning within a given mathematical theory and on automatic theorem proving in the next section.

II.1.4. Notational Support for Syntax-Driven Verification

From the examples and proof rules presented so far, we can see that assertive programs in REFORMS will generally take the form:

(1) \[ C \backslash \text{Assume } P; \text{ code; Confirm } Q \]

\( P \) and \( Q \) denote predicate calculus statements about the values of program variables. Here \( \text{code} \) denotes an assertive program, which is a sequence of declarations and program statements. \( C \) denotes the context, which consists of specifications for all the external procedures called by \( \text{code} \). We say that such an assertive program is correct or valid if, in those states in which \( P \) is true, executing \( \text{code} \) in the context of \( C \) will cause \( Q \) to be true.

The form of typical proof rules in REFORMS has already been illustrated. In general, each rule has two parts, a lower line, on which code ending with a certain type of statement followed by a \text{Confirm} appears. The top part of a rule lists the hypotheses needed to ensure the correctness of the bottom line.

In our system, the basic programming language constructs are supplemented by key worded syntactic slots which are not needed by the compiler, but rather are introduced into the language for the purpose of allowing the programmer to supply appropriate assertions at specific points of the program so that the proof system can be automated. \text{Assume} and \text{Confirm} are examples of such key words.

Another example, as seen in the \texttt{while} construct of example 2, is the key word, \text{Maintaining} which serves to alert the programmer that an assertion, namely the loop
invariant, is needed for this construct. Similarly, the require and ensure clauses in
the procedure declaration construct are key words alerting the programmer to fill in
the necessary specifications and alerting the verifier that those specifications are there
to use.

The programmer must supply the assertions which REFORMS provides key
words for, but notice that the generation of intermediate assertions is mechanical. An
automated verifier can identify syntactically what rule to apply at each step of the
proof. Moreover, after all the pertinent rules are applied, the statement which remains
is one over the underlying theory for the given program, and the reasoning can be
done using only that theory.

An automatic verifier will perform three major tasks: (1) generation of assertions
by applying the rules, (2) simplification of these assertions, probably as the verifier
proceeds through the code, rather than all at once, and (3) application of axioms and
theorems of the underlying mathematical theories in an attempt to prove the final
assertion. Much work is being done in the second and third areas [30, 46, 40, 17]. My
work in verification is concerned with the first task. The proofs in this thesis will
emphasize only the mechanical generation of assertions. The simplification of the
hypotheses and the application of the underlying mathematical theories will be done
without automation, leaving the automation of those tasks to others [46, 40, 17]

II.2. Semantics

The development of axioms and proof rules which can be applied mechanically
to assertive programs to establish their correctness is a significant step toward formal
reasoning about computer programs. However, two important questions about such a
proof system remain to be addressed. One of these questions is obvious, yet
absolutely vital; the other is more subtle, but nevertheless important. The vital issue - - whether or not the rules are sound - - can be informally expressed by the question, "Might the rules permit one to prove that a program is correct when, in fact, it is not?"

The second, more subtle, issue - - whether or not the rules satisfy the completeness property - - can be informally expressed by, "Are the rules adequate to prove the correctness of every valid program?"

Our system deals with both of these issues by providing, in addition to the proof rules for establishing correctness, semantics for assertive programs. The intuitive notion of program validity can be defined formally and then used to check the system for soundness and completeness. Soundness, for example, is proved inductively by establishing for each proof rule that if its hypotheses are semantically valid, then its conclusion line must also be semantically valid.

To understand the meaning of "valid," one needs to know the REFORMS explanation of states. States are Cartesian products $St \times AS$ where $St$ is the usual set of mappings from Variable Names to Values. The Value set will be the set of mathematical objects used in a program. The assert status set $AS = \{VT, CF, N\}$ contains three elements: VT to stand for vacuously true, CF for categorically false, and N for neutral. Each member stands for a possible assert status of an assertive program in this system. The assert status is initially neutral. It becomes vacuously true when an Assume predicate evaluates to false, and categorically false when a Confirm predicate is false. So, for example, if the initial state does not satisfy the pre-condition for a program, then the assert status changes to VT. For a given assertive program we insist that initially all variables have some value of the appropriate type.

An assertive program is correct or valid if, for all starting states with assert status N, the assert status does not change to CF during execution. This semantic
explanation is for partial correctness. In chapter 6, some additions will be made to accommodate total correctness.

The semantics of the usual programming constructs are given elsewhere, [16] but I include an example, the semantics for the assignment construct, to illustrate the form which the semantics take:

$$\text{Sem}(\text{code}; x := \text{exp})(S) = S''$$ where

$$S' = \text{Sem}(\text{code})(S)$$ and

$$S''(id) = S'(id) \quad \text{if } id \neq x$$

$$\text{eval(exp}, S') \quad \text{if } id = x$$

$S$, $S'$, and $S''$ are states. $\text{Eval(exp}, S')$ means the value of the expression $\text{exp}$ in state $S'$ and $id$ represents an arbitrary program variable identifier. The metavariable $\text{code}$ stands for the program statements which precede the assignment.
Chapter 2 presented most of the programming language constructs used in REFORMS to deal with control aspects of programming and illustrated both the constructs and their proof rules with examples. This chapter lays out the syntax for writing modules to define new data types and for creating variables of those types to use in programming. In addition, we set forth the rationale for our new approach to variable creation and explain the philosophy on which it is based.

In Chapter 1, the idea of promoting reusability by using a four layer mechanism to create variables was introduced. Before giving the details of how this layering works, a few comments which put this new view of reusability in proper perspective are in order.

III.1. Object Oriented Programming

A number of programming systems have recently achieved high visibility in the literature under the banner of being “object oriented.” Creators of these systems claim that an important rationale for using these systems is that they permit a high degree of reusability. The Smalltalk language is a typical example [19].
In Smalltalk and systems like it, each object belongs to a particular class of objects, and the classes form a hierarchy, with objects in any particular class inheriting many of their operational capabilities from other classes of which their own class is a subclass. In fact, a given class may be a subclass of several unrelated classes, thereby inheriting quite diverse capabilities.

In a traffic simulation program, for example, cars, trucks, and busses may be considered as subclasses of the class of vehicles, and whatever operations, such as refuel, can be performed on vehicles may automatically be performed on elements of any subclass. So objects are reused in the sense that they belong to multiple classes and thereby inherit attributes of those classes, and operations are reused in the sense that they may be applied to any members of classes on which such operations have been defined.

In such systems, operations are carried out by sending messages to objects. So a truck \( T \) might be told to speed up by 10 miles per hour by sending it the message, 

\[ T.\text{Change\_Speed}(+10) \]

If \text{Change\_Speed} is an operation associated with the class of either \textit{vehicles} or \textit{trucks}, then this message makes sense and can be performed.

However, suppose that the new command is used to create an instance of the class \textit{vehicle}, say \( V \). Suppose that the code provides that various classes of vehicles may be moved into \( V \). These may include cars, trucks, motorcycles, etc. Then suppose that elsewhere in the code, a message is sent to \( V \) to \text{Close\_Door}. This message may or may not be appropriate, depending on what particular object is currently assigned to \( V \). Moreover, it is not possible to check this statically. It is only during the running of the program that it becomes possible to determine to which subclass of vehicle the current value of \( V \) belongs.

In other words, by attempting to reuse both objects and operations on those objects in this way, one loses the ability to check whether or not a given instruction
has any sensible semantics whatsoever. This has the effect of eliminating altogether the possibility of enforcing strong type checking, and leaves only the choice of weak run time checking or no checking at all [10, 31, 53, 56].

Both Smalltalk and REFORMS focus on reusability as an important goal of any software development system, yet the two systems embody very different ideas about what reusability really means as well as how to achieve it. The immediate question we must ask here is not whether inheritance is a good mechanism for permitting a programmer to reuse what he has already developed. The question of how to achieve reusability is meaningless until we first determine exactly what it is that we want to reuse. What is it about software that we want to identify and capture so that we can avoid duplication of effort?

Traditionally, reusability of code has been what most programmers have worked toward as their goal. Certainly the inheritance mechanism of Smalltalk provides for reusability of code at a superficial level. Each subclass inherits properties and operations from its superclasses. Unfortunately, the price we pay for using inheritance is the sacrifice of not only static type checking, but of intellectual manageability as well. If the inheritance hierarchy for a given class is complex enough to provide its intended flexibility and reusability, a programmer, in order to take advantage of the inheritance, must keep track of all the superclasses and subclasses, as well as sibling classes. In fact, if the user fails to be completely familiar with the particular class hierarchies of the program he is working on, it is entirely possible that he may define a new class which duplicates an existing but forgotten class (forgotten or unrecognized in spirit, if not in name). Such an oversight clearly defeats one of the purposes of having inheritance to begin with.

The development of REFORMS is based on the observation that what programmers really need to reuse are the conceptualizations of their objects and
operations. The reuse of code takes care of itself in this approach. Examples in this thesis will illustrate the fact that it is possible to provide programmers with a high degree of reusability of both concepts and executable code without sacrificing strong type checking.

For example, a single conceptualization of a given type, such as a stack, will be sufficient to permit programmers to create stacks of characters, stacks of reals, stacks of stacks of queues of integers, etc. All of these instances are made possible by the use of parameterization in this conceptualization.

Hence, we achieve flexibility while maintaining intellectual manageability. With only one concept and one conceptual name to remember, a programmer may define a large number of computer types which differ from each other only with respect to what appears on their parameter lists and what their implementation details are. By developing our system in this way, we gain not only reusability, but separation of concerns as well, and we make it relatively easy for programmers to identify what conceptualizations are available for their use.

III.2. Module Structuring for Reusability

In Chapter 1, considerable attention was given to the limitations and shortcomings of contemporary languages with respect to providing users with tools for defining and using their own abstract data types. This section addresses these shortcomings by illustrating how types are specified and implemented in REFORMS.

An example of a module which exports a well-known data structure, the stack, will serve as a vehicle to introduce our syntax and to clarify what we mean by reuse — reuse of theories, specifications, and implementations.
III.2.1. Module Layering in REFORMS

An important distinction to note between the approach to specifications in REFORMS and that in most other systems is our complete separation of the conceptual module (specification) from the realization module (implementation). This separation provides the foundation for information hiding, separation of concerns, and most importantly, reusability.

A user of a given module needs to read only the conceptual version of the module (the specifications) to know what the module does. Implementation details are provided in a completely separate component. This encourages programmers to think at the conceptual level, allowing them to view objects as mathematical entities, rather than collections of bits, pointers, or other implementation minutiae.

The conceptual module is pure specification, in that it contains no executable code. Its mathematical formality permits the application of rigorous mathematical reasoning to prove that the implementation (realization module), which will be given later in this chapter, meets the specifications.
III.2.2. Stack Example

The conceptual description of a stack facility would take the following form in our notation:

Conceptual Module Bounded_Stack_Template (type Item; val max_depth: integer);
    require max_depth > 0;

type Stack :: Str(Item);

    exemplar S;
    constraints |S| <= max_depth;
    initially S = \lambda;

procedure Push (var valu: Item; var S: Stack);
    require |S| < max_depth;
    ensure S = #S o #valu and Item.init(valu);

procedure Pop (var S: Stack; var res: Item);
    require S \neq \lambda;
    ensure #S = S o res;

procedure Is_Empty (const S: Stack): Boolean;
    ensure Is_Empty = ( |S| = 0 );

procedure Is_Full (const S: Stack): Boolean;
    ensure Is_Full = ( |S| = max_depth );

procedure Clear (var S: Stack);
    ensure S = \lambda;

end Bounded_Stack_Template;
The syntax Conceptual Module indicates that this module provides formal specifications, in this case for a Bounded_Scatt Template. Associated with this module are parameters to provide maximum flexibility in reusing the module. In particular, the type parameter Item allows the user to adjust for the type of values which will reside in the stack, while max_depth allows him to specify the maximum number of items he ever plans to put in a stack.

The keyword require is followed by a clause which puts restrictions on the parameters to the module. In this example, max_depth is required to be a positive integer, thereby excluding the possibility that one might specify a stack which is always empty - - not a very useful structure.

The next section of the module comprises code which defines the objects and operations provided by the module. The keyword type defines a computational type stack which will be in terms of some mathematical domain. In a conceptual module, a type must be identified with a subdomain of a mathematical domain, in this case, strings of Items. A programmer uses the exemplar syntactic slot to choose a symbol to represent a prototypical object of the type being defined. The programmer also specifies constraints which all objects of the given type must meet and states what values variables of the given type will have when they are first defined.

Following the type definitions are the procedures which manipulate variables of the given types. Associated with the procedures are parameter lists, var to indicate a parameter whose value may change during execution of the procedure and const to mean that the value of the variable will be preserved so that it is the same at the end of the procedure execution as at the beginning.

The require clause of a procedure states restrictions on the incoming values of parameters, and the ensure clause specifies what should be true upon completion of the procedure, if the input conditions have been met. For purely functional procedures
(operations which are total functions), the require clause is omitted.

The conceptual module provides a complete formal specification for the Bounded_Stack_Template. There may be multiple implementations of these specifications, each called a realization. The syntax for realization modules can be illustrated using an array realization for bounded stacks.

Realization Module Array_with_Top_Pointer for Bounded_Stack_Template:

Uses Array_Template, Record_Template;

Module Item_Array_Fac is Array_Template(Item, max_depth)
Realized_by Standard_Array_Realiz;

Module Stk_Record_Fac is
Record_Template(Item_Array_Fac.Array, integer)
renaming P1 as Contents,
P2 as Top,
Swap1 as Contents_Swap,
Swap2 as Top_Swap;
realized_by Standard_Record_Realiz;

Type Stack = Stk_Record_Fac.Record;

conventions
S.Contents.l_bd = 1 ∧ S.Contents.u_bd = max_depth ∧
0 ≤ S.Top ≤ max_depth ∧
for all i: Integer, if S.Top < i ≤ max_depth, then
Item.init (S.Contents.valu(i));

correspondence
conc.S = ⊠

initialization

duration O(max_depth * S.Item.initialization + 1);

with T do
    Set_Bdd(Contents, 1, max_depth);
end;

procedure Pop (var S : Stack; var res : Item);

duration O(1+T.Item.initialization);
var fresh val: Item;
with S do
    Swap_Entry(Contents, Top, fresh_val);
    res := fresh_val;
    Top := Top - 1;
end;
end;

procedure Push (...)

end Array_with_Top_Pointer;

Before examining this syntax in detail, it will be advantageous to see the syntax for creating an instance of a given type using the modules of this example and for declaring variables of that type. For example, suppose that we want to create an instance of the Bounded_Stack_Template which provides stacks whose entries are integers and whose maximal depth is to be limited by 100. We write:

Module Integer_Stack_Facility is Bounded_Stack_Template(integer, 100)
    Realized_by Array_with_Top_Pointer;

The module declaration syntax serves to link this particular instance explicitly to both a particular implementation and to its specification. These links allow reuse of the specifications in the conceptual module and the code in the realization module. By using this syntax, the programmer, in addition to setting up these links, also adapts the module to his particular needs by selecting appropriate parameters.

Finally, to create a variable of this newly defined type, the programmer uses familiar notation:

var S: Integer_Stack_Facility.Stack;

Integer_Stack_Facility.Stack means the type Stack exported by the Integer_Stack_Facility module instance. The Module name Integer_Stack_Facility is affixed to the type name stack because there may be several other stack modules in use in the program.
Understanding how variables are created will facilitate our examination of the realization module syntax, since immediately following the heading are two module instance declarations. These module instances export types Item_Array_Fac.Array and Stk_Record_Fac.Record, respectively, which are considered global to this realization module. We have named the realizations to suggest that for commonly used types such as arrays and records, there will normally be some standard realization provided in the library of reusable parts.

Returning to the first line (the heading) of this module, we find a link to the conceptual module via the keyword for. This particular realization happens not to have a parameter list, but if realization-specific parameters were needed, the realization module syntax provides that a list of them may be included. It is understood that the realization has access to everything in the conceptual module, including its parameter list.

The next section of code which begins with the keyword type sets up the computer implementation of the conceptual type defined mathematically in the specifications. In our example we have chosen to define the type stack mathematically as a collection of strings which satisfy certain constraints, and to represent variables of this type on a computer by using an array and a top pointer. Our reason for doing so is that conceptually any stack (or stack instance) may be thought of as a string and all reasoning to be done by stack users will be easier when performed at this level of abstraction. However, to achieve efficiency, we will select another representation for the implementation.

The implementation must provide actual stacks, using the lower level objects which are available. The chosen representation may require various restrictions on the realization data structures and these are specified by the conventions. In the stack example, reading the conventions requires us to know something about the
Array_Template and the Record_Template because those are the structures used in our implementation. Both structures are included in Appendix B along with several other primitive modules, such as integers and characters. For now it is sufficient to know that any record is a cartesian product and that Stk_Record_Fac.Records happen to be pairs consisting of an integer named Top and an array named Contents. Contents is an array exported by the Item_Array_Fac module instance. An array is a triple made up of a function called valu and lower and upper bounds, \( l_{bd} \) and \( u_{bd} \) respectively.

With this information about records and arrays, we can read the clauses of the conventions which state that the array we are using to implement a stack has \( I \) and \( max\_depth \) as its lower and upper bounds, that the top pointer must be constrained to stay between those bounds, and that the array must remain initialized above the top position.

Since our specifications define a stack in terms of mathematical strings while our realization implements one as an array with a top pointer, if we want to do formal reasoning about the correctness of our operation implementations, we will need a mathematical formula stating the precise connection between the conceptual and realization views. The correspondence provides this connection. In this example, it is possible to recapture the conceptual view of a stack as a string of items from its array implementation by concatenating the entries in the array, starting at the first element and ending with the one whose index is Top.

The last part of the section dealing with the implementation of the type stack is initialization code. Its purpose is to perform whatever action is necessary so that the initially clause of the conceptual module will be satisfied and so that the realization conventions will be met as well. A proof that this has been accomplished be presented in the next chapter, along with other proofs concerning the stack module. Here we will concentrate on the syntactic aspects of the initialization construct.
The **duration** keyword introduces into our language a syntactic slot for stating complexity claims. Another language construct, **with**, borrowed from Pascal, appears in this initialization. Since types defined in REFORMS frequently involve use-of records, it is important for the programmer to have easy access to record fields. The **with** construct permits one to refer to any field of a record by its field name, avoiding the necessity of prefixing the field name with the record name. The proof rule for **with** is in Appendix A and illustrations of its application appear in Chapter 4.

One can think of the initialization as a special procedure, one which is associated with every module which exports a type. It is a procedure which is automatically invoked by the system whenever a new variable of a given type is declared.

The rest of the code in the realization consists of all the procedures that the conceptual module promised to stack users: Push, Pop, Is_Empty, Is_Full, and Clear. We have chosen to show as an example the Pop procedure because it happens to have in it a language construct not previously introduced and because it is an interesting procedure for which to prove correctness.

The language construct we introduce here is one called **swap**, symbolized by `:=:`, reminiscent of the familiar assignment, but with an important difference. The syntax, `var1 :=: var2` means that the values of the two variables `var1` and `var2` are exchanged. The reason for introducing this construct is efficiency: swapping can always be done in constant time, but for large objects, assignment cannot. By providing the **swap** construct, we assist programmers to write efficient programs by encouraging them to avoid the making of copies whenever possible. In contemporary programming practice, assignments are performed in many situations when a **swap** would do.

For example, in parameter passing in Pascal and similar languages, val parameters are copied into a new location and then destroyed when the procedure returns. As long as a val parameter is of a type which doesn’t take long to copy, that’s
fine. However, if the type happens to be a stack of stacks of integers, the copy process may be quite time-consuming, if it's possible at all. In REFORMS most parameter passing is by swap so that no copies need to be made. This swapping must actually be accomplished by an exchange of pointers, but the pointers are kept by the system and are invisible to the programmer.

Of course, there are situations in which one really needs a new version of some structure and so the assignment construct is retained in REFORMS. The syntax for assigning the value of a functional expression $fval\_exp$ to the variable $v$ is given by: $v := fval\_exp$, and the meaning of this is that the value of $fval\_exp$ is computed and assigned to $v$. In a strict interpretation of our language, the more traditional $var1 := var2$ would not be allowed. You would have to write $var1 := \text{Replica}(var2)$, so that copying and its attached cost would be explicit. In general, the amount of time and space required to carry out an assignment depends on how complex $fval\_exp$ is. Choosing to swap rather than to copy in appropriate circumstances can save programmers significant amounts of time and space.

### III.2.3. Type Creation

The stack modules provide a concrete example to keep in mind as we now examine reasons which have led us to develop this layered approach for module specification. One philosophical belief behind REFORMS is that programmers gain both efficiency and comprehensibility by performing small operations on large objects as opposed to the other way round. The programmer needs to distinguish between the conceptualization and the realization of any object and the operations on that object. Just as a programmer prefers to think about mathematical integers as
opposed to bit patterns which represent them in a computer, a programmer will reason more effectively about any other type of variable if that type is presented at a suitably high level of abstraction. Moreover, by permitting the reasoning to take place at the conceptual level, REFORMS can often take advantage of existing mathematical theories to supply both the language for formal specification and the accompanying theory to support formal reasoning.

For example, rather than developing a whole new theory of bit-patterns to prove results about computer integers, REFORMS has mechanisms which mechanically transform constructs about bit patterns to statements over the theory of integers.

This approach does not deny the need for a rich collection of predefined types to be available to the programmer. In fact, REFORMS suggests that a substantial library of verified types be available to programmers. To show how these types can be defined and how variables of these types can be declared, we need to examine in detail the four declaration layers, discovering what job each layer accomplishes, what support each layer requires, and how the layers are related to each other.

At the conceptual level, one defines a type in terms of a mathematical domain which is a reasonably close match to the desired programming type. For example, programming integers are mathematical integers which lie between some set of bounds. So conceptually, to get computational integers, one can use mathematical integers constrained to take value within the specified range. The idea is to choose a mathematical domain whose theory is rich enough to encompass the desired computational type, but not so rich as to complicate the job of identifying the constraints necessary to pick out the required subdomain. For example, one would not want to define computational integers in terms of real number theory because, although the integers can be identified within the real number domain, listing the constraints to do so is unnecessarily complicated and would lead to a very
cumbersome system for establishing results - especially in view of the fact that integer theory has already been worked out in mathematics.

In the stack example, the stack is defined as a string and the operations are described in terms of string theory. This enables a user of the stack module to do his reasoning by picturing a stack as a string of items where the items are all of some type which is a parameter to the stack module. For a given stack $S$, one can check to see if $S$ is the empty string or if $S$ is as large as allowed, and one can add or remove items at one end of the string.

Each operation is specified according to the syntax for procedures. For example, the Push operation receives as parameters a stack $S$ and a variable value of whatever type the elements of the stack belong to. To screen out undesirable input, the requires clause states that the stack on which something is to be pushed has a length less than the specified upper bound. The ensures clause states that after the push has been performed, the new stack $S$ can be expressed as the old stack ($S$ concatenated with the element given, val.

It is important to note that the conceptual module includes everything that a stack user needs to know about a stack without revealing any implementation details, as well as everything a programmer needs for designing an implementation. In fact, there may be many implementations for a single conceptual module, but each must satisfy the same set of specifications - those supplied in the conceptual module.

With these philosophical observations in mind, we summarize our four level process for the instantiation of each variable. At level 1, the conceptual module gives a definition of a type, leaving open what values the parameters to the module will actually take. For example, the stack defined in our conceptual module might be a stack of integers or reals or stacks or some exotic user-defined type introduced elsewhere. Hence, we need only one conceptual module to specify any kind of stack.
At level 2 is a realization (implementation) for the specifications. The realization consists of code written in the programming language. At this level the parameters are still left open and the realization can be checked by the proof rules without knowing what the values of parameters will be, other than the fact that they are of the specified types.

At level 3 the parameters are filled in and we get a particular instance of a stack module, such as one exporting the type stack of integers or stack of queues. This third level declaration also specifies what particular realization for the given type is being used. This allows still another level of reusability since there may be many realizations for some types, each promoting efficiency within some particular set of circumstances.

Finally, at level 4 is the declaration of a variable to be of the type exported by the instance declared at level 3. For example, in the Pop procedure, \textit{fresh\_val} is declared to be of type \textit{Item}.

III.2.4. Another Layered System

Now that we have seen an example illustrating how module specifications are done in REFORMS and have become acquainted with the underlying philosophy for using this approach, we will compare it to another way of specifying modules proposed by the Larch project [59]. This is of interest because, in Larch, one also writes specifications in layers which are called tiers.

The Larch two tier approach separates the specification of a module into two components: one which is language independent and which defines the theory of whatever type is to be introduced, and another layer which is language dependent. This separation allows reusability of the theory tier for a variety of languages, but
requires that a new theory be defined every time a new module is written.

In our four layer approach to specifying types and creating variables, reusability is introduced at all the layers. Moreover, rather than defining new theories for each type, REFORMS reuses existing mathematical theories, promoting still more reusability.

III.3. Comments about Primitive Modules

To show further how the REFORMS approach to module specification promotes reusability and uniformity in treatment of modular components, we will now look more closely at primitive types, those which have traditionally been considered as supplied by the programming language, that is, built in types. In the stack module we need instances of the types boolean and integer so that we can use the integers zero and one. In all likelihood, implementations of REFORMS will not require the programmer to declare these primitive types, but rather it will make them part of a universally available library. For now we will assume that there is such a library.

An important question concerning such a library has to do with whether or not primitive types need to be treated differently from user defined types. This chapter has proposed a solution for the problem of creating types. One way to test how well a solution fits a problem is to check for exceptions, special cases which the suggested solution fails to accommodate. The fewer such exceptions the better, no exceptions being the desired result.

With this goal of homogeneity of treatment in mind, we need to examine both user defined and primitive types to see if they can be treated the same. Many recently developed languages have set out to address the desirable property that user-defined types can be achieved so that strong-typing is maintained and these user-defined types
can be treated as though the language had provided them, thereby promoting reusability. Pascal, and more recently, Ada have taken steps toward this goal. However, in these languages and others, certain types, usually integer, character, boolean, and string, must be present in the language in order that additional types may be defined at all, thereby distinguishing them from user-defined types and taking away the possibility of boot-strapping the type collection. In REFORMS all types are given a homogeneous treatment, thus allowing the boot-strapping effect.

This is not to suggest that everyone who ever writes a program in REFORMS will want to or should do this. On the contrary, the plan is that a library of useful types will be developed to provide types which will be reused many times. The point is that their development can proceed exactly the same way as the development of any other modules.

This approach has the effect of substantially increasing the collection of types available to a given programmer without the need for the programmer to develop every one of them himself. This way everything from a simple boolean type to a complex hierarchically built nested list facility become available for any programmer to use. If one believes that there are several, but not a huge number, of structures about which programs can be naturally written, this approach makes a giant step toward reusability.

To illustrate these ideas we have put together a collection of modules which often come built-in to a programming language and hence show how these modules can be hierarchically combined to boot-strap a programming environment. We will discuss these at the end of the next chapter. A few comments about arrays are perhaps appropriate here, however, so that our stack realization can be completely understood.

In most strongly typed languages an array of length 10 is a different type from an array of length 100. This means that if the size of an array is changed, all the code
dealing with it must be recompiled. This contradicts our reusability goal for REFORMS. To avoid being forced to set the bounds and initialize a given array at the conceptual or realization level, we put into the module which exports the type array a procedure called Set_Bounds which receives as parameter a particular array together with whatever upper and lower bounds the programmer desires. This means that the actual setting of array size can be delayed until the array is declared. When the programmer declares an array, it is initialized so that the upper bound is less than the lower bound. Hence no space need be allotted. Only when the Set_Bdds procedure is called is the space allocated. Additional comments about arrays appear in the next chapter, and a complete module specification appears in Appendix C.
CHAPTER IV
Rules for Verifying Basic Modules

Chapter 2 explained how our four layer approach to creating variables promotes a high degree of reusability. However, as the title of this thesis suggests, we are not satisfied with reusability alone. The reuse of incorrect programs or program segments leads only to additional incorrect programs, and so in REFORMS, when we talk about reuse, we encourage reusing programs only when they have been formally verified.

Our proof rules for all of our programming language constructs, except for the module constructs, were discussed in Chapter 2 and catalogued in Appendix A, and their use was illustrated with example programs. This background will be important as we look next at proof rules which enable us to establish that a realization module meets the specifications laid out in its associated conceptual module. We will actually need a proof rule for each of the constructs in the four layers of the variable creation process, but the rule for realization declarations embodies the central concepts that make the whole system work.
IV.1. The Conceptual Module Declaration Rule

In our variable oriented view of programming, "modules" are the most important structural unit in programs. Syntactically, as well as logically, the specification of a module precedes a realization (implementation) of it, and so the first module rule we present is the conceptual module declaration rule. Because of the relative sophistication of this and subsequent rules, we will follow a pattern of presenting the rule in an abbreviated form followed by a detailed expansion of each component and then an explanation.

By looking at the rule, in addition to learning what hypotheses must be established to prove the bottom line of the rule, one also gains familiarity with the general form of a conceptual module. This general form reveals what types of variables may appear on parameter lists and what may belong to which argument lists. The rule follows:
Conceptual Module Declaration Rule

\[ C \cup \{C_M\} \{code; \text{Confirm Q} \}
\]

\[ C \setminus C_M; \{code; \text{Confirm Q} \}
\]

where \( C_M \) is:

\[
\text{Conceptual Module } C_M N(C_Param_List);
\]

\[
\text{Require } C_M\_Req\{cu, cg\};
\]

\[
\text{Uses } C\_Use\_List
\]

\[
\text{Type } T \subseteq \text{Math}\_T;
\]

\[
\text{exemplar } t;
\]

\[
\text{constraints } C\text{strt\_Cond}(t, cu, cg, ch);
\]

\[
\text{initially } C\text{init\_Cond}(t, cu, cg, ch);
\]

\[
\text{Proc } p(\text{var } cx: T, \text{const } cy: T);
\]

\[
\text{require } C\text{pre}(cx, cy, cu, cg, ch);
\]

\[
\text{ensure } C\text{post}(cx, cy, #cx, cy, cu, cg, ch);
\]

end

and \( C_Param\_List \) is:

\[
(\text{val } cu: CT1, \text{Type } CT2, \text{Domain } CD1, \text{def } cg(\text{val } z: CT3): CT4)
\]

and \( C\_Use\_List \) is:

\[
\text{def } ch, \text{Type } CT25; (*\text{math theories}*)
\]

The effect of this rule is that conceptual module declarations pass directly into the context without any further checking. Since the conceptual module contains no executable code, but rather a formal specification for any realization, the proof system requires only that the specification be available so that the realization code can be verified.

In order to understand the syntax used in describing this rule, note that parameter lists are enclosed in parentheses ( ) whereas \( \_ \_ \) brackets are used to enclose lists of variables which may occur within assertions. These latter lists are not essential, but do serve to remind us what objects a particular assertion may concern. Abbreviations which are expanded below the module template are given in italics.

The parameter list, \( C_Param\_List \), of a conceptual module may contain variables of known (previously defined) types, types themselves, domains, and definitions. By
definition, we simply mean a mathematical expression which defines a function from one mathematical domain to another. As to requiring that variables be of previously defined types, this limitation avoids the complex and apparently unnecessary circular reasoning that might occur should modules be allowed to be recursively defined.

The conceptual module requirement clause $C_M_{\text{Req}}$ is an assertion which puts restrictions on the variables of the parameter list appropriate for this particular module. Frequently, these $C_M_{\text{Req}}$ clauses set bounds on the imported variables. In the stack example, the module requirement states that the maximum depth of the stack must be positive.

On the Uses list, $C_{\text{Use}_\text{List}}$, may appear definitions, types, and domains (this includes their accompanying theories). In most systems there will probably be some domains which will automatically be considered as part of the Uses list. Examples include integers, characters, and booleans.

The Type section of the specification defines a computational type in terms of some mathematical domain. The REFORMS view is that every computational type belongs to the set of mathematical domains, but not vice versa. However, it is likely that any domain can be simulated as a computational type by introducing bounds and specifying what limitations such bounds impose on the type. For example, mathematical integers can be simulated as computational integers by specifying what are the smallest and largest integers possible to realize on a given computer. The integer module at the end of this chapter does this by specifying a $\text{min\_int}$ and a $\text{max\_int}$.

In writing these constraints and initialization clauses for a new type, it is convenient to be able to talk about a prototypical variable of the new type. The syntax, exemplar, provides the capability of identifying such a variable. The constraints clause $Cnstrnt_{\text{Cond}}$, tells what restrictions variables of the newly
defined type must always satisfy. For example, in the case of the stack, every stack must have its length remain within the specified bounds. The initially assertion \textit{Init\_Cond}, specifies what initial conditions must be true for variables of this type when they are declared.

Following the definition of the type which the conceptual module specifies are the procedures which operate on variables of that type. Each procedure has a parameter list, \texttt{var} indicating parameters whose values may be changed by the procedure and \texttt{const} identifying those which will be returned unchanged.

The \texttt{require} clause \texttt{pre}, tells what assumptions must be met in order for the procedure to guarantee its ensure clause \texttt{post}, upon termination. I will point out later how REFORMS can be used in either a partial correctness mode or a total correctness mode. For now it is sufficient to know that for partial correctness, when a procedure is designated as correct, this means that if the procedure terminates, then the ensure clause is guaranteed, but in the case of non-termination, nothing is guaranteed. When used in total correctness mode, verification in REFORMS checks for termination and treats programs which do not terminate as invalid.

The rule for Conceptual Module declarations is a simple one because all it really calls for is enriching the context with the newly defined conceptual module. One can think of the context as part of an environment for the verifier. If a call is made to a procedure, the specification of that procedure must be in the context so that the verifier knows what it does. Similarly, when the verifier checks a module realization for correctness, the specifications in the conceptual module for that realization must be available.
IV.2. Realization Module Declaration Rule

Before presenting the rule for Realization Module declaration is presented, some preparatory comments are in order. The rule takes up a lot of space and hence may give the impression of being difficult to understand. I ask that the reader remain open-minded and believe that long is not necessarily abstruse.

There are two reasons why this rule is lengthy. First, the rule must be written in such a way as to accommodate all the possible kinds of parameters, variable declarations, and constructs which may appear in any realization. As we saw in the stack module and will see in additional examples in subsequent chapters, normally only a few of these possibilities actually occur in any one module.

The second reason for the length of the rule is that all argument lists have been given in expanded form. Note that the abbreviated forms are really quite short. I include these complete expansions only to avoid any confusion as to what is allowed in any given list.

Before stating the rule, we exhibit a prototypical form of a realization module for the conceptual module discussed in the previous section. It includes one each of the possible kinds of parameters, variables and constructs allowed by our language:

```plaintext
Realization_Module R_M_N(val rv: RT2; Type RT5; Domain RD1;
def rf(val rx: RT13): RT6;
proc rq(var ry: RT14);
require pre_rq/ry,rf,rv\;
ensure post_rq(ry,#ry,rf,rv\ );
for C_M_N;
Uses Def rh. Type RT9, Proc rq2. Domain RD2,
Conceptual_Module CM2, Realization_Module RM2;
Require M_Req/rv,rf,cu,cg,ch\;
```
Module T_Inst is CM1(cu1,CT26, cg1, CD2)
    Realized_by RM1(rv1,RT27,rf1,rq2, RD2);
var rz: RT7;
adj var rw: RT8;

Conventions M_Conv_Cond/cu,rv,rw,rz,rf,cg,ch\end
Initialization M_I_Body end;
Type T = RT20;
    conventions Conv_Cond/t,rv,rw,rf,cu,cg,rz,ch\end
    correspondence Corr_Exp/conc.t,t,cu,rv,rw,rz,rf,cg,ch\end
    initialization I_Body end;
procedure p(var x: T; const y: T);
p_body
end;
end

In the rule we abbreviate the realization module as $R_M$. We consider each realization module to have three parts: (1) There is a heading which includes the name of the module, together with its parameter list and what requirements those parameters must meet. We will abbreviate the heading as $R_M_H$ in the rule. (2) There is a set of declarations needed in the module. This includes possibly global variables and declarations of module instances. We will abbreviate this set of declarations as $R_M_D$ in the rule. (3) The module has a body where the programmer implements a conceptual type by choosing a representation and writing appropriate code for the procedures which manipulate variables of that type.

Other abbreviations used in the rule include $R_Param_List$ to stand for the realization module parameters, $C_Param_List$ to represent the conceptual module parameter list, and $C_M$ to mean the conceptual module itself. With these abbreviations in mind, we can discuss what happens to the context in our new rule. Because lexically the conceptual module appears before the realization module, the context has already been enriched with the conceptual module. The rule emphasizes this fact by explicitly showing $\{C_M\}$ as part of the context on the bottom line of the rule.
On the upper lines of the rule a context $C'$ appears. This $C'$ represents
\[ C \cup \{C_M\} \cup \{C_{\text{Param List'}}\} \cup \{R_{\text{Param List'}}\} \cup \{R_{M_D}\} \]

where $R_{\text{Param List'}}$ is:
\[
R_{\text{Param List}} \text{ with } "Type RT5"
\]
\[
\text{replaced by}
\]
\[
\text{Type RT5} = \text{Domain}; \quad \text{exemplar } x;
\]
\[
\text{Constraints RT5.Cstmt}(x);
\]
\[
\text{Initially RT5.Init}(x);
\]

This makes available what the verifier needs to know about the imported type RT5.

A similar treatment of the conceptual parameter list yields:
\[
C_{\text{Param List'}} \text{ is } C_{\text{Param List}} \text{ with}
\]
\[
\text{"Type CT2" replaced by}
\]
\[
\text{Type CT2} = \text{Domain1}; \quad \text{exemplar } y;
\]
\[
\text{Constraints CT2.Cstmt}(y);
\]
\[
\text{Initially CT2.Init}(y);
\]

Some general observations about the conventions we have used in naming the variables which appear on parameter lists, on argument lists, and in declaration statements in our rules follow;

- Conceptual variables begin with C such as CT and realization variables begin with R, RT2. Similarly, at the third level, the instance declaration level, variables will begin with the letter "i."
- Repl_T1 and Repl_T2 suggest replicatable types, meaning types which have already been defined and can therefore be used.
- In general we consider the types of the parameters to be available in the context. By listing them explicitly, we emphasize that these particular types are to be used in this module.
- We consider computer types to be mathematical types as well.
- A definition is viewed as an association with some tuple an object of a known type.
- The parameter lists have one of each kind of parameter which has occurred in any of the module examples we have developed so far. These lists may need to be supplemented if new examples are discovered for which additional types must be imported. However, we do have a substantial set of examples to base our
representative lists on. [48]

With this preparation we are ready to look at the rule. It is presented here in abbreviated form with expansions and explanations following.

Realization Module Declaration Rule

\[
\begin{align*}
\text{Well\_Def\_Corr\_Hyp} \\
C' \setminus M_{\text{Init\_Hyp}} \\
C' \setminus T_{\text{Init\_Hyp}} \\
C' \setminus \text{Correct\_Op\_Hyp} \\
C \cup \{C_M\} \cup \{R_M,H\} \text{ aspt; code; Confirm Q} \\
\end{align*}
\]

Since most of the bottom line is already familiar, we will begin our discussion there. The abbreviation \textit{aspt} has been chosen to suggest assumptions and indicates that some assertions may appear before the realization module. As we know, the \( R_M \) stands for the realization module declaration. The rule indicates that the module may be followed by some code and a \texttt{Confirm} statement. This is the usual form of our rules.

We will now turn our attention to the top lines of the rule. The context has already been explained, so we will focus on the hypotheses in the order in which they appear. First is the \texttt{Well\_Def\_Corr\_Hyp} which is read "well defined correspondence hypothesis." This hypothesis states that any programming object which the implementation may create has a conceptual counterpart. This is the hypothesis that justifies allowing the programmer to do his thinking at the conceptual level and thereby to avoid the confusion between what the program is supposed to do and what the implementation details are.
For example, in the case of the stack, the user may think strictly in terms of strings and concatenations without even knowing about arrays and top pointers or, indeed, any other implementation details. This complete separation of concept from representation yields a real bonus in reusability. No matter how many stack instances occur, and no matter how many different implementations are written, the user needs to know only one conceptual view of stacks.

The expanded form of Well_Def_Corr_Hyp is:

Assume $C_M_{Req}/cu, cg, ch \land M_{Req}/cu, cg, ch, rv, rf \land M_{Conv\_Cond}/cu, ch, rv, rz, rf, cg$;

Confirm $\forall x: RT20$ if $Conv\_Cond[t \rightarrow x]/cu, ch, rv, rz, rf, cg$ then $\exists y: MathT$ such that

$(Corr\_Exp[conc.t \rightarrow y, t \rightarrow x]/cu, ch, rv, rz, rf, cg \land Cstmt\_Cond[t \rightarrow y]/cu, cg, ch)"

The Well_Def_Corres_Hyp assumes the conceptual module requirement $C_M_{Req}$, the realization module requirement $M_{Req}$ and the module conventions $M_{Conv\_Cond}$. It confirms that for any variable $x$ of the realization type $RT20$, if $x$ satisfies the type invariant $Conv\_Cond$, then there exists a conceptual variable $y$ of the appropriate type for which the correspondence is well defined and the conceptual type constraints $Cstmt\_Cond$ hold.

Remember that $t$ is the exemplar in our prototypical conceptual module, and that the conc.$t$ indicates the conceptual version of $t$, while $t$ itself stands for a prototypical variable as it would appear in the realization module.

The second hypothesis $M_{Init\_Hyp}$ establishes correctness of any code associated with newly declared variables that are global within the module. The stack example doesn't have any global variables, but in Chapter 4, the Nested List Facility does have some global variables, and the $M_{Init\_Hyp}$ will be applied to establish that these global variables meet the specifications of this particular module.
Third is the type initialization hypothesis $T_{Init\_Hyp}$ which establishes that the initialization procedure is done correctly. This differs from the module initialization hypothesis, $M_{Init\_Hyp}$ because it deals strictly with initialization of variables of the new type introduced. In REFORMS, all types must have initial values, and it is part of the compiler’s job to generate a call automatically to the appropriate initialization routine. The verifier checks the initialization code against the initially clause $Init\_Cond$ of the conceptual module.

$T_{Init\_Hyp}$ is:

\[
\begin{align*}
\text{Assume } & \quad C_{M\_Req}/cu,.cg, ch \land M_{Req}/cu, cg, ch, rv, rf \land \\
& \quad RT2.\text{Cstmt}(rv); \\
& \quad M_{I\_Body}; \\
& \quad \text{Confirm } M_{Conv\_Cond}/cu, rv, rw, rz, rf, cg, ch;
\end{align*}
\]

In the Assume clause are the conceptual module requirement, the realization module requirement, and the realization module conventions, together with the type constraints which apply to whatever variables have been declared at the realization module level. The declaration var t: RT20 causes a new variable of the realization type to be created. For example, in the stack initialization, the new variable is a record composed of an array and a top pointer.

$I\_Body$ represents the code which carries out the initialization. Finally, the Confirm clause includes the fact that the module conventions and the type
conventions hold, and that the correspondence together with the conceptual type constraints imply that the initial conditions set forth in the conceptual module are met.

The correct operation hypothesis Correct_Op_Hyp guarantees that any procedure introduced in this module meets its specifications. If a module has six procedures, this hypothesis is then applied six times. It is easy to see similarities with the procedure declaration rule presented and illustrated previously. What is new in the module version of procedure declaration is that there are additional assumptions to make and conclusions to check, which come automatically from other components of the module specifications.

These new specification components include module and type invariants whose syntax was illustrated and discussed in the previous chapter. However, although the correspondence was also explained at the abstract level, it is here that we truly see the significance of its contribution to reasoning for the first time.

The application of the correct operation hypothesis will show how the correspondence makes it possible for the proof rules to be applied to the realization code, while reasoning about the variables in the client programs can be kept at the conceptual level.

Correct_Op_Hyp is:

\[
\begin{align*}
\text{Assume } & \text{C}_M\text{-Req}/cu, cg, ch \land \text{M}_\text{Req}/cu, cg, ch, rv, rf \land \\
& \text{M}_\text{Conv}_\text{Cond}/cu, ch, rv, rz, rf, cg \land \text{RT2.Cstmt}(rv) \land \\
& \text{RT7.Cstmt}(rz) \land \text{RT8.Cstmt}(rw); \\
\text{Remember; } & \text{Assume Conv}_\text{Cond}[t \rightarrow x]/cu, ch, rv, rz, rf, cg \land \\
& \text{Conv}_\text{Cond}[t \rightarrow y]/cu, rv, rz, rf, ch, cg \land \\
& \text{Corr}_\text{Exp}[t \rightarrow \text{conc}.x]/cu, rv, rz, rf, cg \land \\
& \text{Cstmt}_\text{Cond}[t \rightarrow \text{conc}.x]/cu, cg, ch \land \\
& \text{Cstmt}_\text{Cond}[t \rightarrow \text{conc}.y]/cu, cg, ch \land \\
& \text{pre}[x \rightarrow \text{conc}.x, y \rightarrow \text{conc}.y]/cu, cg, ch; \\
\text{body; } & \text{Alter conc}.x; \\
\text{Confirm } & \text{M}_\text{Conv}_\text{Cond}/cu, ch, rv, rz, rf, cg \land \text{C}_M\text{-Req}/cu, cg, ch \land \\
& \text{Conv}_\text{Cond}[t \rightarrow x]/cu, rv, rz, rf, ch, cg \land \\
& \text{Corr}_\text{Exp}[t \rightarrow \text{conc}.x]/cu, rv, rz, rf, cg \land \\& \\
\end{align*}
\]
As in the initialization hypotheses preceding this one, the first three lines which comprise the Assume clause include both the conceptual and realization module requirements, the realization module conventions, and the constraints satisfied by variables declared within the given realization module. As in our procedure declaration rule formerly introduced, the Remember syntax alerts the verifier that we need to keep a copy of current values of whatever variables might change during execution of the procedure. This saved value of a variable will be referred to later as the "old" value and denoted by \( z \).

The second Assume clause states that the realization type conventions are satisfied when the appropriate parametric substitutions are made. Similarly, the actual parameters are substituted into the correspondence expression and the constraints clause of the conceptual module. Finally, the precondition for the given procedure with its parameters appropriately replaced is assumed.

These assumptions are followed by the procedure body. The Alter construct is applied to the conceptual form of the variable \( x \), \( \text{conc} \cdot x \) because the correspondence permits us to do our reasoning in terms of it.

In the Confirm clause we find both the realization and the conceptual module constraints, as well as the realization type conventions. Finally, we must show that the correspondence, together with the conceptual type constraints are sufficient to prove the post condition of the procedure with the appropriate parametric substitutions.

The last line of the top part of the rule states that in the given context, using the assumptions \( \text{aspt} \) the remaining code followed by the Confirm Q statement is correct.
IV.3. Module Instantiation Rule

Having presented rules which deal with declarations at the first two levels of our four layer approach, we are ready to see what happens when a module is instantiated. Before presenting the rule for module instantiation, we point out that in our layered approach to creating variables, there is an economy of issue allotment. Once a given issue has been handled in one layer, successive layers need not be concerned with that issue. For example, at the realization module level, all procedures are examined for correctness. At the instance level, all that must be done is to see if necessary conditions are met upon appropriate parametric substitutions, because, until instantiation takes place, the parameters remain unbound.

In order to instantiate a module and thereby to create the computational type exported by that module, the syntax requires that a name for the module instance be given, followed by the name of the conceptual module it instantiates and the name of the particular realization to be used. Delaying the binding of parameters until a module instance is declared is what permits reusability of the same conceptual type for multiple instances and a variety of realizations.

In order to create an instance of some type $T$, the user checks the library to see if a specification for $T$ exists and if so, what verified realizations are available. From the library the user picks the realization that meets his needs. If the library has no module to export $T$, then the user may write his own specifications and implementation following the syntax just presented. As in the rules for the first two layers, the third layer rule not only shows what is required for verification, but also illustrates the syntax of the instance declaration.
Module Instantiation Rule

\[
C_M_{\text{Req}}[cu \rightarrow iu, cg \rightarrow ig] \land M_{\text{Req}}[rv \rightarrow iv, rf \rightarrow if] \land
\text{pre}_p[cy \rightarrow iy, rf \rightarrow if, rv \rightarrow iv] \Rightarrow \text{pre}_q[iy, if, iv] \land
\text{post}_q[iy, #iy, if, iv] \Rightarrow \text{post}_p[cy \rightarrow iy, #cy \rightarrow #iy, rf \rightarrow if, rv \rightarrow iv]
\]

\[
C'' \equiv \text{code; Confirm Q;}
\]

\[
C \equiv \text{aspt; } I_M; \text{code; Confirm Q;}
\]

where \( I_M \) is:

\[
\text{Module Inst is } C_M N(iu, IT2, ig)
\]

\[
\text{Realized by } R_M(iv, IT5, if, iq)
\]

and where \( C' = C \cup \{C_M\} \cup \{R_M, H\} \cup \{\text{Proc iq(var iy: IT11); require pre}_q[iy, if, iv]; \text{ensure post}_q[iy, #iy, if, iv]\} \) and

\[
C'' = C' \cup \{\text{Type Inst.T = Math_T, T.Init_Cond, T.Cstmt}\} \cup \{\text{Proc Inst.p(var ix: T, const iy: T) require pre}_p[ix, iy]; \text{ensure post}_p[ix, #ix, iy]\}
\]

The first hypothesis of the rule checks to see that the conceptual module requirement clause \( C_M_{\text{Req}} \) is satisfied when its value parameter \( cu \) is replaced by the actual instance parameter \( iu \) and the definition parameter \( cg \) is replaced by the actual definition \( ig \). Similarly, the second hypothesis checks the realization requirements at the module level with the appropriate parameter substitutions. The value parameter \( rv \) is replaced by the actual parameter \( iv \) while the definition parameter \( rf \) is replaced by \( if \). The rule shows that the context must be enriched by associating with each procedure on the parameter list of the module correctly parameterized pre and post conditions. Since the correctness of the procedure code was already verified at the realization level, it remains to show that when the prescribed parametric substitution is made in \( \text{pre}_p \), the resulting statement is strong enough to imply the pre condition of \( iq \) and that the post condition of \( iq \) implies the
post condition of \( rp \) with the formal parameters replaced with their actual counterparts.

### IV.4. Variable Declaration Rule

The last of our declaration rules is the rule for variable declarations. Each new variable introduced into a program must be declared to be of some type exported by some previously declared module instance. The rule tells what the verifier must do upon encountering a variable declaration.

**Variable Declaration Rule**

\[
C \leftarrow \text{aspt; Assume}(\text{Inst.T.Cstrnt}(x) \land \text{Inst.T.Init}(x)) \\
\quad \text{code; Confirm Q;}
\]

\[
C' \leftarrow \text{aspt; var x: Inst.T; code; Confirm Q;}
\]

where \( C = C' \cup \{ \text{Type Inst.T} = \text{Math_T'}, T'.\text{Init_Cond}, T'.\text{Cstmt} \} \)

As one expects, the verifier, while processing declarations, upon coming to a variable declaration, must put into the (local) context the mathematical domain on which this type instance depends, together with the appropriate initialization clause and type constraints. In addition to enriching the context, the verifier must generate \text{Assume} clauses which state the initial conditions and the type constraints for the newly declared variable. We will see the significance of this in the next section in which these rules are applied.
IV.5. Using the Rules

An application illustrating the use of these rules emphasizes the mechanizable nature of the verification system by exhibiting for each of the required hypotheses an illustration of that hypothesis for the case of the stack example. Note that the Well_Def_Corr_Hyp is a mathematical one which involves no programming code, whereas both the T_Init_Hyp and Correct_Op_Hyp, because they involve programming code, make use of other proof rules in the system.

In order to focus on the main reason for presenting the proof, that of illustrating the module rules, we will assume that primitive module instances already exist for integers, booleans, arrays, records, characters, and strings. Hence, rather than declaring an instance of an integer facility and generating the integers zero and one which are needed in this stack module, we will assume those already exist. However, in the section on primitive modules, we will see how this can be accomplished using exactly the same methods as illustrated here for stacks.

The proof given here shows how all the intermediate assertions are mechanically generated by applying the proof rules in correct order. However, while care has been taken to simplify assertions only when appropriate, simplification rules are another issue altogether, as is the issue of automatic theorem proving.
Chapter 2 included both a conceptual module to specify stacks and a realization module which implemented that conceptual module. This section illustrates use of the realization module proof rule by applying it to this stack example.

The other three declaration rules require only brief commentary. The verifier, in progressing through the program containing the stack code, would first come to the conceptual Stack_Facility, which, in conformity with the conceptual module declaration rule, it would place in the context. The realization declaration rule, on the other hand, would generate a lot of work, and that is what this example will deal with. During the work done in processing the realization module, the verifier will use the instance declaration rule, because instances of arrays and records are declared in the stack realization. However, upon encountering those declarations, all the verifier must do is to enrich the context and to generate the constraint conditions for arrays and records. Finally, we will see an application of the variable declaration rule in the initialization procedure, because at the beginning of the initialization, a variable of type stack is declared.

It will probably be easier to follow the application of the module realization rule if we repeat the declaration of the stack realization from Chapter 2 here:

```
Realization Module Array_with_Top_Pointer for Bounded_Stack_Template;
    Uses Array_Template, Record_Template;

Module Item_Array_Fac is Array_Template(Item, max_depth)
    Realized_by Standard_Array_Realiz;

Module Stk_Record_Fac is
    Record_Template(Item_Array_Fac.Array, integer)
        renaming P1 as Contents,
        P2 as Top,
```
Swap1 as Contents_Swap,
Swap2 as Top_Swap;
Realized by Standard_Record_Realiz;

type Stack = Stk_Record_Fac.Record;

conventions
  S.Contents.l_bd = 1 ∧ S.Contents.u_bd = max_depth ∧
  0 ≤ S.Top ≤ max_depth ∧
  for all i: Integer, if S.Top < i ≤ max_depth, then
    Item.init (S.Contents.valu(i));

correspondence
  conc.S = \{\frac{s}{S}.Contents.valu(i)\}_{i=1}^{s.Top}

initialization

  duration O(max_depth * S.Item.initialization + 1);

with T do
  Set_Bdd(Contents, 1, max_depth);
end;

procedure Pop (var S : Stack; var res : Item);

  duration O(1+T.Item.initialization);

  var fresh_val: Item;
  with S do
    Swap_Entry(Contents, Top, fresh_val);
    res := fresh_val;
    Top := Top - 1;
  end;
end Pop;

procedure Push ( ...
  ...
  ....
end Array_with_Top_Pointer;

There are four hypotheses in the general form of the realization module declaration rule. For the stack, however, since there are no global variables, there is no module initialization hypothesis.
We will deal successively with the remaining three hypotheses in the order in which they appear in the rule.

Verification of Stack Realization Module

The first is the well defined correspondence hypothesis. We need to establish that any realization variable has a conceptual counterpart which meets the given constraints. Mechanically filling in what the rule calls for, we obtain the following statement as what is needed:

**Well_Def_Corr_Hyp**

**Need**: Assume max_depth > 0;

**Confirm**: \( \forall x: S_{\text{Record}.Record}, \text{if} \ (0 \leq x.\text{Top} \leq \text{max_depth} \land \forall i: \text{Integer}, \text{if} \ x.\text{Top} < i \leq \text{max_depth} \land \text{Item.Init}(x.\text{Contents}(i)) \) then \( \exists y: \text{Str(Item)}, \text{such that} \ |y| \leq \text{max_depth} \land \)

\[ y = \prod_{i=1}^{x.\text{Top}} \text{Contents}(i) \]

The proof requires us to use only the assume and confirm rules and a bit of string theory.

**Proof**: Assume \( x: \text{record} \). We must find \( y: \text{string} \) to satisfy the given conditions.

Assume \( 0 \leq x.\text{Top} \leq \text{max_depth} \). Then

\[ \prod_{i=1}^{x.\text{Top}} \text{Contents}(i) \]

is well-defined.

Let \( y = \prod_{i=1}^{x.\text{Top}} \text{Contents}(i) \)

By definition, \( |y| = x.\text{Top} \leq \text{max_depth} \).

It is now clear that the expanded form of the Well_Def_Corr_Hyp, while necessary in the writing of the rule, in reality for a particular module, will probably have only a few of the possible kinds of parameters and may not need every possible kind of assertion. In the stack, there is no module level require clause and there is no parameter list at all. Hence, the formulation of what is needed to prove the given hypothesis is reasonably short to write and the proof itself uncomplicated.
There is no module initialization M_Init for the stack and so next we consider Type initialization. Chapter 6, however, will present a proof illustrating module instantiation.

The Well_Def_Corr_Hyp required us to use only the assume and confirm rules, but the initialization and the Correct_Op_Hyp both require applications of rules for other language constructs. However, we have seen examples of applications of these rules before in Chapter 2, and so we proceed with the proofs, mechanically filling in the generated assertions.

As before, first is a statement of what is needed to prove the T_Init_Hyp. This is easily obtained by mechanically filling in what the hypothesis calls for. We know that the verifier will make one pass during which it processes the variable declaration and then will proceed from the end of the code, applying rules as needed.

\[ T\text{-Init\_Hyp} \]

Need: \( \text{C}' \text{ Assume max\_depth} > 0 \)
\[
\text{var T: S\_Record.Record;}
\text{with T}
\text{Set\_bdd\_and\_Init(Contents, 1, max\_depth);}
\text{end;
Confirm T.Contents.l\_bd = 1 \land}
\text{T.Contents.u\_bd = max\_depth \land}
\text{0 \le T.Top \le max\_depth \land}
\text{\forall i \text{ if} T.Top < i \le max\_depth}
\text{then Item\_Init(T.Contents.val(i)) \land}
\text{(conc.T) \le max\_depth \land}
\text{conc.T = \mathbb{1}\text{Top}}
\text{T.contents.val(i)_{i = 1} = 1 => conc.T = \Lambda)}
\]
\[
\text{where C' = \{Conceptual Module Stack\_Facility,}
\text{val max\_depth: integer; Type Item\}}
\]
Proof:
Variable declarations are carried out first, and upon applying the variable declaration rule to \text{var T: S\_Record.Record}, the verifier adds the following assumption to the above Assume statement:
Assume max_depth > 0 ∧ T.Contents.l_bd = 1 ∧ T.Contents.u_bd = 0 ∧ T.Top = 1;

The contents is enriched to C" and the code remains to be checked:

with T
    Set_Bdd_and_Init(Contents, 1, max_depth);
end;

where C" = C ∪ {Type S_Record.Record = Cart.Prod.,
                Item_Array.Array(Contents), Top = 0,
                constraints ... }

For brevity, I will refer to the new Assume clause as P and the Confirm clause as Q. Having taken care of the variable declaration, the verification now proceeds from the end of the assertive code to the beginning, applying the appropriate rules to each construct. The first rule needed is the with rule. Applying it yields the following:

Assume P; Assume Contents = T.Contents ∧ Top = T.Top;
    Set_bdd_and_Init(Contents, 1, max_depth); Alter T;
Assume T.Contents = Contents ∧ T.Top = Top; Confirm Q

Now applying the Assume rule:

Assume P; Assume Contents = T.Contents ∧ Top = T.Top;
    Set_bdd_and_Init(Contents, 1, max_depth); Alter T;
Confirm (T.Contents = Contents ∧ T.Top = Top) => Q

The Alter rule changes only the last line to be:

Confirm ∀ T, (T.Contents = Contents ∧ T.Top = Top) => Q

Now applying the call rule to Set_bdd_and_Init(Contents, 1, max_depth)
generates a change only to what must be confirmed:

Confirm Contents.l_bd = 1 ∧ Contents.u_bd = 0 ∧
    1 ≤ max_depth ∧
    ∀ ?Contents, ?Contents.l_bd = 1 ∧
    ?Contents.u_bd = max_depth ∧
    ∀ i, if 1 ≤ i ≤ max_depth, Item_Init(?Contents.val(i)) ∧
    T.Contents = ?Contents ∧ T.Top = Top
=> Q[Contents => ?Contents]
Now we have only Assume and Confirm statements over the underlying theories:

Assume \( \text{max\_depth} > 0 \land T.\text{Contents}.l\_bd = 1 \land T.\text{Contents}.u\_bd = 1 \land T.\text{Top} = 0; \)
Assume Contents = T.\text{Contents} \land \text{Top} = T.\text{Top};
Assume T.\text{Contents} = \text{Contents} \land T.\text{Top} = \text{Top};
Confirm Contents.l\_bd = 1 \land Contents.u\_bd = 0 \land 1 \leq \text{max\_depth} \land \\
\forall \ ?\text{Contents}, ?\text{Contents}.l\_bd = 1 \land 
?\text{Contents}.u\_bd = \text{max\_depth} \land 
\forall i, \text{if} \ 1 \leq i \leq \text{max\_depth}, \text{Item\_Init}(\ ?\text{Contents}.\text{val}(i)) \)

\( \Rightarrow Q[\text{Contents} \rightarrow ?\text{Contents}] \)

Before filling in \( Q \), we can take care of some of the statements to be confirmed:

Using the assumption that \( T.\text{Contents} = \text{Contents} \) and the fact that \( T.\text{Contents}.lb = 1 \) and \( T.\text{Contents}.ub = 0 \), we can get rid of the top line of the Confirm.

Now filling in \( Q[\text{Contents} \rightarrow ?\text{Contents}] \),

Confirm \( T.\text{Contents}.l\_bd = 1 \land 
T.\text{Contents}.u\_bd = \text{max\_depth} \land 
0 \leq T.\text{Top} \leq \text{max\_depth} \land 
\forall i, \text{if} \ T.\text{Top} < i \leq \text{max\_depth}, \text{then} 
\text{Item\_Init}(T.\text{Contents}.\text{val}(i)) \land \Pi^T \leq \text{max\_depth} \land 
\con T = \prod_{i=1}^{\text{top} T} ?\text{Contents}.\text{val}(i) \land 
\Rightarrow \con T = \Lambda \land 

Now by observing that \( ?\text{Contents}.l\_bd = 1 \) and \( ?\text{Contents}.u\_bd = \text{max\_depth} \), and using the assumptions of the with rule that \( T.\text{Contents} = ?\text{Contents} \), the first line of the Confirm clause is established.

Similarly for the \( \text{Item\_Init}(T.\text{Contents}.\text{val}(i)) \).

Finally, using the assumption that \( T.\text{Top} = 0 \), since

\( \con T = \prod_{i=1}^{\text{top} T} ?\text{Contents}.\text{val}(i) \land 
\) by definition of \( \Pi \), \( |\con T| = 0 \) and hence \( \con T = \Lambda \).
Since the Correct_Op_Hyp is the most complicated hypothesis of the Realization Module declaration rule, it will help us to consider some differences between traditional mathematical proofs and the syntax-driven (mechanical) proofs we are performing here. Mathematicians strive to avoid the inclusion of any assumptions which are redundant or irrelevant both in the statement of a theorem and in any proof to establish the correctness of that theorem. This practice makes proofs easier to read, because they are free of assertions which, although true, are unnecessary.

However, if automation is desired, and we believe that the tedium of proving program correctness necessitates a syntax-driven proof system, it is inevitable that redundant and even irrelevant clauses may be generated during the mechanical application of proof rules. Keeping in mind that some hypotheses in the following proofs may be generated, but not actually needed, we will label those particular hypotheses which are necessary to facilitate easier reading.

Because the Correct_Op_Hyp is lengthy, and because it requires us to amass assertions from both the Conceptual and Realization modules for our stack example, before actually showing the proof, we will write the assertions needed all in a single section so that following the rule mechanically will not require us to continually turn to other pages.

For applying the Correct_Op_Hyp to the Pop procedure, we will need the following assertions, listed here in terms of the exemplar for the module which exports the type about which the assertions are made. First are the clauses which apply to the stack module. After them are assertions we need to know about arrays, exemplar A, type item, and exemplar I.

Conv_Cond: \( T.\text{Contents}._\text{l_bd} = 1 \land T.\text{Contents}._\text{u_bd} = \text{max_depth} \land 0 \leq T.\text{Top} \leq \text{max_depth} \land \forall i: \text{integer}, \text{ if } T.\text{Top} < i \leq \text{max_depth} \text{ then } \text{Item.Init}(T.\text{Contents}.\text{val}(i)) \)

Corr_Exp: \( \text{conc}.T = \prod_{1}^{\text{Top}} T.\text{Contents}.\text{val}(i) \)
Cstmt_Cond: \(|T| \leq \text{max\_depth};

C_M_Req: \text{max\_depth} > 0;

pre-condition for the \text{Swap\_Entry} procedure for arrays: \(A.l\_bd \leq i \leq A.u\_bd\);

post condition for the \text{Swap\_Entry} procedure for arrays: \(A.l\_bd = \#A.l\_bd \land A.u\_bd = \#A.u\_bd \land A.val(i) = \#v \land v = \#A.val(i) \land \forall j: \text{integer}, \text{if } j \neq i, \text{then } A.val(j) = \#A.val(j)\);

Array.Cstmt(Constraints for Arrays): \(A.l\_bd \leq i \leq A.u\_bd \land A.u\_bd < A.l\_bd + \text{max\_size}\).

The variables used are those given in the Array module as it appears in Appendix B. The \text{Swap\_Entry} procedure begins as:

\[
\text{proc Swap\_Entry(var A: Array, const i: integer, var v: Item)}
\]

We will indicate the constraints on variables of type \text{Item} by \text{Item.Cstrnt}.

In applying the \text{Correct\_Op\_Hyp} line of the rule, we will use the above assertions with the appropriate variables substituted into the argument lists. The numbers enclosed between \(< >\) symbols are used as labels so that as we progress through the proof, we will be able to refer to the indicated hypotheses.

\text{Correct\_Op\_Hyp}

For procedure \text{Pop}:

\text{Need: Assume } <1> \text{max\_depth} > 0;

\text{Remember; Assume } <2> S.Contents.l\_bd = 1 \land S.Contents.u\_bd = \text{max\_depth }\land <3> 0 \leq S.Top \leq \text{max\_depth }\land <4> \forall i \text{ if } S.Top < i \leq \text{max\_depth }\land \text{then Item.Init(S.Contents.val(i)) }\land <5> \text{conc.S} = \prod_{i=1}^{\text{S.Top}} S.Contents.val(i) \land \text{conc.S} \leq \text{max\_depth }\land <6> \text{conc.S} \neq \Lambda;

\text{var fresh\_val: Item; with S}

\text{Swap\_Entry(Contents, Top, fresh\_val); fresh\_val := res; Top := Top - 1; end;}

\text{Alter conc.S; Confirm S.Contents.l\_bd = 1 }\land
\[
S.\text{Contents}.u_{bd} = \max_{depth} \land \\
0 \leq S.\text{Top} \leq \max_{depth} \land \\
\forall i \text{ if } S.\text{Top} < i \leq \max_{depth} \\
\text{then } \text{Item.Init}(S.\text{Contents}.\text{val}(i)) \land \\
\max_{depth} > 0 \land \\
\text{conc.S} = \prod_{i=1}^{\infty} S.\text{Contents}.\text{val}(i) \\
\Rightarrow |S| \leq \max_{depth} \land \#\text{conc.S} = \text{conc.S} \circ \text{res};
\]

Proof:

First, the variable declarations are processed, enriching the context and creating new assumptions. The new context includes what was already in \(C'\) together with

\[
\{ \text{Type Inst.Item} = \text{Item_Domain, Item_Domain.Init_Cond, Item_Domain.Cstmt} \}
\]

The variable declaration rule also causes the following clause to be generated:

\(<7>\) Assume \(\text{Item.Init(fresh_val)}\)

Having taken care of the declarations, we are now ready to apply our proof rules in the appropriate order, the first to apply being the with rule which generates some new assumptions both before and after the code of the with construct. The first of the assumptions follow <7>:

Assume <8> \(\text{Contents} = S.\text{Contents} \land \text{Top} = S.\text{Top};\)

Next is the code which is followed by:

\(\text{Alter S;}\)

\(<9>\) Assume \(S.\text{Contents} = \text{Contents} \land S.\text{Top} = \text{Top};\)

\(\text{Confirm Q; [Q represents the confirm clause given at the beginning].}\)

Next we apply the assignment and swap rules. The resulting clauses which come after the line of code calling \(\text{Swap_Entry}\) are:

\(\forall S, S.\text{Contents} = \text{Contents} \land <10> S.\text{Top} = \text{Top} - 1 \Rightarrow\)

\(\forall \text{conc.S} \)

\(S.\text{Contents}.l_{bd} = 1 \land S.\text{Contents}.u_{bd} = \max_{depth} \land \)

\(0 \leq \leq \max_{depth} \land \forall i, \text{if } S.\text{Top} < i \leq \max_{depth} \)

\(\text{then } \text{Item.Init}(S.\text{Contents}.\text{val}(i)) \land\)

\(\text{conc.S} = \prod_{i=1}^{\infty} S.\text{Contents}.\text{val}(i)\)

\(\Rightarrow\)
conc.S = Λ ∧ #conc.S = conc.S o fresh_val

Next, the call rule must be applied to Swap_Entry. This adds a new confirm clause:

(1) Confirm Contents.l_bd ≤ Top ≤ Contents.u_bd ∧
   ?Contents.u_bd = Contents.u_bd ∧
   <11> ?Contents.val(Top) = fresh_val ∧
   <12> ?fresh_val = Contents.val(Top) ∧
   <13> ∀ j: integer, if j ≠ Top, then
   ?Contents.val(j) = Contents.val(j) ) =>

∀ S, <14> S.Contents = ?Contents ∧ S.Top = Top - 1 =>
∀ conc.S, (2) S.Contents.l_bd = 1 ∧ S.Contents.u_bd = max_depth ∧
<15> 0 ≤ S.Top ≤ max_depth ∧
∀ (3) i: integer, if S.Top < i ≤ max_depth then
       Item.Init(S.Contents.val(i)) ∧
<16> conc.S = ∏_{i=1}^{max_depth} Contents.val(i)

(4) |conc.S| ≤ max_depth ∧ (5) #conc.S = conc.S o fresh_val

As mentioned previously, the <> symbols are used to number assumptions which will be used in proving results. The () symbols have been introduced to identify clauses which must be confirmed. In the final confirm statement, five clauses have been labeled and we now have a look at how each can be proven.

To establish clause (1), we can use <3> and <8>. Clauses <2>, <8>, and <9> are sufficient to confirm (2). <14>, <7>, and <4> prove that statement (3) is correct. Applying the Remember rule removes the # symbol in front of #conc.S. Then <8>, <9>, <10>, <11>, <12>, and <14> permit us to rewrite clause (5) as

conc.S = ∏_{i=1}^{max_depth} Contents.val(i)
       o fresh_val

which is equivalent to <16>.

Finally, using the above statement, we can derive (4) by using the definition of Π (string concatenation) together with <3>.
IV.6.1. Observations

Some observations concerning programming style may help to predict how complex or simple typical proofs will be. Our approach to designing and developing modules has been applied in writing the specifications for more than 20 data structures. In these examples, we have found that the procedures usually consist of three or four lines of code. Since our style dictates that we perform small operations on large structures, we believe that, although proof of program correctness is not a trivial exercise, it need not be an impossible dream.

Moreover, once a module has been verified, it can be used in other programs without any re-verification. This will be illustrated in Chapter 6 by the use of stacks in the Nested List Module presented there.

IV.7. Enhancements

We have seen how our four layer approach to specifying modules leads to reusability at each of the four layers. There is still another way in which REFORMS makes reusability natural. In addition to reusing mathematical theories and allowing multiple realizations for each conceptualization, REFORMS promotes reusability by distinguishing between primary and secondary operations. Into a conceptual module which exports one or more types, one puts operations which enable a user to manipulate variables of the given type(s) in ways appropriate for that type. For example, because a variable of type stack needs to permit additions and deletions at
one end, push and pop are provided. To allow defensive programming, Boolean functions are given which permit one to see if the stack is empty or full.

However, although some occasions may call for additional operations, such as copying the stack, clearing the stack, and others, because it is important to allow for a variety of implementations, the primary stack module includes only those operations which cannot be expressed in terms of others and which are necessary to manipulate variables of the given type as a programmer would expect. The burden on the implementor is then kept to a minimum.

In reality a programmer may want to perform a variety of operations on stacks - - operations which could be realized by the primary ones. For example, one may want to copy, reverse, compare two stacks for equality, or get the depth of a stack. It is easy to see how any of these operations can be carried out by using some combination of the primary operations.

These other operations are called secondary operations, and any or all of them may be added to a library of reusable modules by declaring an Enhancement Module. As an example to illustrate syntax, consider the following [48]:

Module Copyable_Int_Stack_Fac is Stack_Facility(100, integer)
   Enhanced by Copying_Capability((*: Int) := (*: Int))
   Realized by Array_with_Top_Pointer;
var S,T: Copyable_Int_Stack_Fac.Stack;
   .
   .
Copy_Stack(T,S);
   .
   .

The obvious advantages of separating these secondary operations from the primary module are that (1) Not every implementation of a stack must include these secondary operations, thereby allowing various machines and systems to include only those stack manipulations which are appropriate for them, (2) The verified primary
module can be reused in a variety of enhancements, (3) There is less redundancy in the library of software because the intersection of any set of enhancements need appear only once.

With regard to primary modules, however, the REFORMS point of view is not totally spartan. For example, even though integer multiplication can be performed by repeated addition, and division by repeated subtraction, the integer module has both multiplication and division as primary operations because a user would probably consider those operations as natural for integers.

IV.8. Primitive Modules

IV.8.1. General Comments

We have now seen how a well-known data type, a stack, can be created and verified for reuse under REFORMS. Clearly the stack module was dependent on other modules which were assumed to exist, including booleans, integers, arrays, records, each of which, in turn, may be dependent on others.

Indeed, the question of what types must be available in order for a user to program with ease, creating new types as he chooses, is a challenging one. Even more challenging is the achievement of uniform treatment of all types.

This chapter proposes one scenario for providing so-called primitive types. This approach adheres to the requirement that these primitives be treated just like any other type, that is, that they can be created and verified using the same techniques illustrated in Chapters 2 and 3.
I do not claim this to be the only way to achieve the goal of providing these primitives, but rather one possible way, useful because the modules built this way can be verified for reuse, and because these modules, built in the recommended order, can be used to bootstrap type building in our system.

IV.8.2. Bootstrapping

To start the bootstrapping of our system we need a module to export the type boolean. It uses the theory of two-valued Boolean Algebra and allows a user to generate the Boolean values true and false and the connectives 'and' and 'not.' An enhancement of this module would probably be included in the primitive library to provide other functions such as 'or' and 'nor.' Mathematics tells us that any other Boolean function can be built using 'and' and 'not.'

Once the boolean type is defined and implemented (probably in the machine hardware), an instance can be declared to be globally available for other modules. Chapters 2 and 3 have pointed out the importance of keeping specifications independent from implementations, and these primitive conceptual modules clearly possess the desired independence.

The following is a way to specify a module to export type boolean, followed by a declaration of an instantiation:
IV.8.3. A Boolean Module

Conceptual Module Boolean Facility
uses Two_Val_Boolean_Alg;

- type boolean £ {true, false}; exemplar B;
  - initially B = (true);
- proc gen_true(): boolean
  ensure gen_true = (true);
- proc gen_false(): boolean;
  ensure gen_false = (false);
- proc and(const x,y: boolean): boolean;
  ensure and = (x \& y);
- proc not(const x: boolean);
  ensure not = (¬ x);

end Boolean_Facility;

Before declaring a module instance, a few obvious but important points should be made about this primitive conceptual module. This module has no parameter list, but this is no surprise since we are assuming that no other types have been defined. The uses list makes it clear that this module is to be specified and verified with two-valued Boolean Algebra as the underlying mathematical theory. All four procedures are total functions, and hence, they need no require clauses to screen input.

As far as realizations go, most systems should implement booleans in the hardware using some agreed upon standard. For that reason, our instance declaration suggests the use of whatever that standard realization might be.

Module Std_Boolean is boolean_Facility Realized_by standard;

It should also be noted that the standard Boolean facility enjoys a singular position in our programming language because the basic control constructs, If and While, depend on this module.
IV.9. A Hierarchy of Primitives

IV.9.1. Integers

Once it is possible to create variables of type boolean, one might next consider
characters or integers, both being important to any programming language. Our
conceptual integer module exports type integer, making use only of the mathematical
type of integers and the type boolean already defined and instantiated. The integer
module permits the generation of a smallest integer and a greatest integer. These
values -- clearly machine dependent -- are defined in the specifications.

The integer module also provides for generating the integer zero, checking for
equality and less than. In addition, the standard operations of addition, subtraction,
multiplication, and division, an increment procedure and one to solve the congruence
\[ i \equiv r \pmod{m}, \]
given \( i \) and \( m \) are included.

A typical enhancement might include procedures to check whether or not two
integers can be added (or multiplied or subtracted or divided).

The following is our integer facility, followed by an enhancement and the
declaration of an instance:

```latex
Conceptual_Module Integer_Facility;
Uses Integer Theory, Std:boolean;

def min_int, max_int: Integer
Constraints min_int \leq 0 < max_int;

type Int \subseteq Integer; exemplar i: Int;
Constraints min_int \leq i \leq max_int;
initially i = 0;
```
proc gen_min(): Int;
    ensure gen_min = (min_int);

proc gen_max(): Int;
    ensure gen_max = (max_int);

proc gen_0(): Int;
    ensure gen_0 = (0);

proc Are_Equal(const x,y: Int): boolean;
    ensure Are_Equal = ( x = y );

proc Is_Less_Than(const x,y: Int): boolean;
    ensure Is_Less_Than = ( x ≤ y );

proc Add(const x,y, var res: Int);
    require min_int ≤ x + y ≤ max_int;
    ensure res = x + y;

proc Incr(const x: Int, var res: Int);
    require x < max_int;
    ensure res = x + 1;

proc Subtract(const x,y: Int, var res: Int);
    require min_int ≤ x - y ≤ max_int;
    ensure res = x - y;

proc Mult(const x,y: Int, var res: Int);
    require min_int ≤ x * y ≤ max_int;
    ensure res = x * y;

proc Div(const x,y: Int, var res: Int);
    require min_int ≤ x/y ≤ max_int ∧ y ≠ 0;
    ensure |y*res| ≤ |x| ∧
          |x - y*res| < |y|;

proc Mod(const i,m: Int, var r: Int);
    require m > 0;
    ensure 0 ≤ r < m ∧
           ∃ q: integer such that i = q*m + r;

end Integer_Facility;

Enhancement Range_Check of Integer_Facility;

proc In_Range_to_Add(const x,y: Int): boolean;
    ensure In_Range_to_Add = ( min_int ≤ x + y ≤ max_int );
end Enhancement;

Other operations may appear in this enhancement, such as checking to see whether or not subtraction, multiplication, or division are possible.

An instance declaration is needed so that programmers can create variables of type integer:
Module Default_Int_Fac is Integer_Facility
  Realized by standard;

One suggested realization for an integer module appears elsewhere [48].

Additional modules which export primitive types appear in Appendix C. A brief description for each is given in the next section of this chapter.

IV.9.2. Characters

Using the integer and boolean facilities previously instantiated, we can now build a character producing module in which an alphabet is specified, representation defined and appropriate operations described, including a conversion procedure which receives an integer and yields its character representation.

The alphabet will of course be realized by some collection of keystrokes. Even at this low level, we observe that reusability of specs is possible because the actual specifics of the alphabet are not given, rather the specs say that some alphabet must be defined. The convert procedure gives a connection to the instantiated integer.

An enhancement specifies a procedure to convert a digit represented in character form to some integer, possibly instantiated according to some unusual integer facility.
IV.9.3. Strings

Now that a way of handling characters has been introduced, the library of reusable modules can add a string facility. Our string module defines type string, declares a variable in which a string can be built, 'newstring,' and provides operations to extend the string, to determine the length of a string, and to get what is currently in New_Str, leaving New_Str empty.

A common problem arising when trying to concatenate strings is how to allot storage for the strings without making the program too slow. The specs for the string module avoid this problem by encouraging an implementation with a fixed place for constructing any string. Once the string is ready to be used, it is viewed as being taken away, leaving the vacated space for constructing the next required string. So here is a situation in which it is possible to keep the reasoning at the conceptual level, yet still design the specifications to encourage an efficient implementation.

IV.9.4. Records

Other modules which our library of reusable software will need are Record and Array facilities. Our record facility has a renaming capability. An example appeared in the stack facility, renaming P1 as Contents, P2 as Top. This facility provides the required projection operators needed to access each record field.
Finally, to support random access, we need an array module. Like the record facility this module has a type as a parameter, and hence the same array facility can be used regardless of what type of variables are to be stored in this structure.

The array specifications allow for a variety of implementations, including a 'lazy' implementation in which the array entries are initialized only when they are used for the first time; hence the Set_Bounds operation. This operation can be called only once for each array instantiation, thereby delaying allocation of space until the array is actually to be used.

The require clause of this procedure prevents capricious resetting of bounds by guaranteeing these procedures only when they are called with the initial conditions true, i.e. lower bound 1 and upper bound 0. Looking ahead to implementation, one notes that these specifications encourage a space and time saving approach by having the programmer explicitly call procedures to set bounds and initialize. When first created, the arrays are empty, and so no space must be allocated until the programmer needs it.

Other procedures allow the user to find out the bounds and, of course, to access elements of the array. The Swap_Entry procedure is consistent with the REFORMS philosophy concerning copy vs. assignment because it specifies the exchange of two values rather than copying either.
IV.10. Primitive Library

It is reasonable to suppose that any programming library would include all of the above modules, possibly multiple instantiations of some. Descriptions of these primitives are included in this thesis for two main reasons: first, to point out the homogeneity of treatment by REFORMS, that is, the specification of primitive modules is carried out using the same language and guiding principles as for any other modules, thereby making it possible to bootstrap. The second reason is to confirm that the proofs given in this thesis are not hiding some underlying problem when they mention that existence of certain primitives is assumed. For example, in the stack realization, the value of one is subtracted from that of Top. By including the integer facility I give evidence that the value of "1" makes sense, that there is a procedure for subtracting, and that verification based on this underlying module is possible.

One finds that in most programming systems, the concepts of integers, booleans, characters, strings and arrays are pervasive. What is important is that in REFORMS all these primitives can be formally specified so that programmers can do their reasoning in the appropriate mathematical setting and that verification of their implementations can be carried out using the proof rules illustrated.
CHAPTER V
Communal Modules

Chapter 3 introduced language mechanisms which permit users to define new types and to create variables of those types in such a way as to promote a high degree of reusability. Chapter 4 provided the logic necessary to establish the correctness of programs using those language constructs.

To illustrate both the syntax for writing modules and the rules for proving their correctness, the bounded stack example was presented. As the conceptual module specified, the depth of any bounded stack has a fixed limiting value $max_{depth}$. Therefore, any implementation of this module must meet the requirement that no stack can grow larger than $max_{depth}$.

With this limit in mind, the conceptual module provides, in addition to the standard push and pop operations for stacks, operations $is\_Empty$ and $is\_Full$ so that a programmer may write defensive code, i.e., he can check to see if a stack $S$ is full before performing a push or to see if $S$ is empty before popping it.

The bounded stack module and others written in the style of what we called Basic Modules are entirely appropriate for use in applications such as the primitive string module which appears in Appendix B. Moreover, the simplicity of the Bounded Stack makes it an ideal example to use for introducing the four layer system for creating variables.
However, in general, for efficient use of space, an implementor of a concept may want to permit all variables of a given type to share one large space allotment, thereby allowing one object to grow quite large while another remains small. For example, in the case of stacks, we may want to specify one overall limit to the total number of entries permitted in all stacks, rather than a fixed bound on each individual stack. Then some stacks may grow large while others may remain small, thus providing more flexibility and space efficiency than could be achieved by using the individual depth bound.

V.1. Specifying Communal Modules

Having presented a rationale for wanting to reason about the aggregate of all variables of a given type, we turn to the challenge of introducing adequate language mechanisms for writing assertions which describe this situation.

In order to talk about a setting in which all the stacks provided by a given stack module share a common storage area, for example, we will need to say that total length of all the stacks does not exceed some fixed limit.

To achieve the desired expressiveness, we turn to the familiar practice of introducing additional auxiliary variables. An auxiliary variable is one which is used to aid in program specification, but which is, in fact, ignored by the compiler. Another name sometimes used for such variables is *ghost variables*.

In our system, rather than placing on the programmer the burden of inventing unique auxiliary variables for each module, we will automatically associate with each type instance $F.T$ a collection of auxiliary variables which will allow programmers to make assertions which refer to all the variables of a given instance.
First we define a set \textit{Specimen\_Number} which consists of the positive natural numbers. Each time a new variable \(v\) of type \(F.T\) is declared, it will be assigned a unique element called "\(v\) number" and denoted \(v\#\) from the \textit{Specimen\_Number} set. These unique numbers, one for each variable, make it possible to distinguish among all variables of type \(F.T\), thereby giving programmers the expressiveness they need to write statements in their specifications about the aggregate of all such variables.

At the same time, to make these statements easy to write, we will need a function which will associate with any given specimen number the value of the variable which has been assigned to it.

Formally:

\textbf{F.T.Denotation}: \textit{Specimen\_Number} \rightarrow T.M\_T_F

The range \(T.M\_T_F\) is the mathematical type exported by the type instance \(F.T\).

Finally we associate with the type \(F.T\) an integer auxiliary variable \(F.T\text{.Last\_Specimen\_Num}\) which indicates the last integer from the set \textit{Specimen\_Number} which has been assigned to a variable of this type. The programmer will not write code to manipulate these auxiliary variables, but may assume that upon instantiation of a new type instance \(F.T\), \(F.T\text{.Last\_Specimen\_Num}\) is initialized to 0, and upon the declaration of each new variable \(v\) of type \(F.T\), \(F.T\text{.Last\_Specimen\_Num}\) is incremented, and the newly declared variable is assigned that specimen number as its unique conceptual identifier \(v\#\).
V.2. A Communal Stack Module

With these auxiliary variables available for writing specifications, we can now introduce a new stack concept -- one which places an upper bound on the total number of entries in all stacks of a given type instance, as opposed to placing a bound on the depth of each individual stack. We call the module introducing this new stack concept *Communal Stack Template* and place on its parameter list *Item* to indicate the type of the stack entries and *capacity* to set the limit on the total number of entries in all stacks of a given instance type.

In this Communal Stack module we will want the traditional push and pop operators, but now detecting whether or not we can perform a push in a given situation will depend not on the depth of one particular stack but rather on the sum of the number of entries in all the stacks. Therefore we will need an operator to provide us with this value. We call this operator *Remaining-Capacity*, a function which has no parameters, and which returns an integer which tells how many more items can be stored in the collection of stacks of type *F.T*.

We will also include an operator *Depth-of* which, given a stack *S* returns the length of *S*. This will enable the programmer to tell if a stack is empty by determining whether its length is 0.

The following is the specification of our new module:

```
Conceptual Module Communal_Stack_Template(var capacity: integer;
  Type Item;

  require capacity > 0;

  Type Stack \subseteq Str(Item);
  exemplar S;
```
definition
\[ \text{Aggregate\_Length} = \sum_{i=1}^{\text{Stack\_Last\_Specimen\_Num}} |\text{Stack\_Denotation(i)}| \]

constraints
\[ \text{Aggregate\_Length} \leq \text{capacity}; \]

initially
\[ S = \Lambda; \]

procedure Push(var Valu: Item, var S: Stack);
\hspace{1em} \text{require} \ \text{Aggregate\_Length} < \text{capacity};
\hspace{1em} \text{ensure} \ S = \#\text{Valu} \circ \#S \land \text{Item\_Init(Valu)};

procedure Pop(var S: Stack; var res: Item);
\hspace{1em} \text{require} \ S \neq \Lambda;
\hspace{1em} \text{ensure} \ #S = S \circ \text{res};

procedure Remaining\_Capacity(): Integer;
\hspace{1em} \text{ensure} \ \text{Remaining\_Capacity} = (\text{capacity} - \text{Aggregate\_Length});

procedure Depth\_of(var S: Stack): Integer;
\hspace{1em} \text{ensure} \ \text{Depth\_of} = (\ |S| );

end Communal\_Stack\_Template;
V.3. A Shared Storage Implementation

In this section we will present an implementation of the specifications for the Communal_Stack_Template, an implementation in which all variables of type stack share common storage in an array \( Stack\_Hldr \). Each individual stack \( S \) consists of a record with an index \( S\.Top \) into \( Stack\_Hldr \) and the length of the stack \( S\.Depth \). Initially, the elements of \( Stack\_Hldr \) are linked together as a single free storage list. The variable \( First\_Free \) is used to indicate the first index into this free storage list.

When a programmer wants to execute a push, the \textit{Push} operation gets space for the new item by removing it from the free storage list. Symmetrically, when a pop is performed, the index of the value popped from the stack is returned to the free storage. A module variable \( cap \) keeps track of how many indices are still available for use. \( Cap \) is decreased whenever a push is performed and increased for each pop.

The following is the initial part of the realization just described:

\begin{verbatim}
Realization_Module Shared_Array for Communal_Stack_Template,

Module Entry_Record_Fac is Record_Facility(Item, integer);
    realized_by Standard;
    renaming P1 as Valu,
        P2 as Next;

Module Entry_Array_Fac is Array_Facility(Entry_Record_Fac.Record, capacity)
    realized_by Standard;

var cap, First_Free: integer;
var Stack_Hldr: Entry_Array_Fac.Array;

Initialization
    uses cap, First_Free, Stack_Hldr;
    var Next_Free: integer;
    Set_Bounds(Stack_Hldr, 1, capacity);
    cap := capacity;
    First_Free := 0;
\end{verbatim}
While First_Free < capacity do
    Next_Free := First_Free;
    First_Free := First_Free + 1;
    Stack_Hldr(First_Free).Next := Next_Free;
end;
end;

Module Stk_Record_Fac is Record_Facility(integer, integer);
    realized_by Standard;
    renaming P1 as Top,
        P2 as Depth;

Type Stack = Stk_Record_Fac.Record;

V.4. Enhanced Proof Rules

We pause before completing our implementation to consider some issues concerning verification. Because one of our goals is to prove the correctness of whatever code we write, it is important that our proof system be powerful enough to do so.

Since we have introduced auxiliary variables and have used them to specify the Communal_Stack_Template, we must make sure that our module proof rules are correspondingly supplemented to deal with the new ideas these auxiliary variables permit programmers to reason about.

Because the array Stack_Hldr is implicitly shared by all the variables of a given stack facility instance, the question of whether or not a procedure call to which the stack S has been passed as a parameter effects changes to other stacks in Stack_Hldr is now of critical concern. Using the auxiliary variables introduced, we can strengthen our correct operation hypothesis in the following way:
For procedure $p(\text{var } L_1: \text{F.T, const } L_2: \text{F.T})$ add to the end of the last \text{Confirm} clause of \text{Correct_Op_Hyp}:

\[ \forall i: \text{integer, if } 1 \leq i \leq \text{F.T.Last Specimen Num}, \]

\[ \text{if } i \neq L_1\# \]

\[ \text{then } \#\text{F.T.Denotation}(i) = \text{F.T.Denotation}(i) \]

In the case of the bounded stack example, since there was no shared storage and there were no global variables, satisfying this additional clause is trivial. In any of the procedures in that stack realization, only the stack which is made available as a parameter to the procedure is accessible to the procedure and hence that procedure cannot either deliberately or inadvertently change any other stack.

In the Communal Stack Module, because there are global variables available to all the module procedures, it is possible that changes to those variables could take place even when they are not passed as parameters to a given procedure. The new clause added to the \text{Correct_Op_Hyp} takes care of this problem by requiring that only the variables on the procedure parameter list may change when the procedure executes.

Another concern which arises as a result of using shared storage is that of returning storage which is no longer needed to the available pool of space. We will want to add a finalization procedure to whatever realization we write, and the presence of this finalization procedure obviously requires an addition to the module rule, i.e. a hypothesis we will call \text{type finalization hypothesis $T_{Final_Hyp}$}, which is analogous to the $T_{Init_Hyp}$. We will need to supplement our \text{type initialization hypothesis} with auxilliary code to update the auxilliary variables, reflecting the fact that upon the declaration of $v$, the last specimen number has been incremented and the auxilliary variable $v\#$ has been assigned that incremented value.

Our modified $T_{Init_Hyp}$ is:

\[ \text{Assume } C_{M_{Req/\text{cu, cg, ch}}} \land M_{Req/\text{cu, cg, ch, rv, rf}} \land \]

\[ M_{Conv\_Cond/\text{cu, rv, rw, rz, rf, cg, ch}} \land RT2.Cstrnt(rv) \land \]

\[ RT7.Cstrnt(rz) \land RT8.Cstrnt(rw); \]
var t: RT20;
Last_Specimen_Num := Last_Specimen_Num + 1;
v# := Last_Specimen_Num;
I_Body;
Confirm Conv_Cond(t,cu,ch,rv,rv,rf,cg) ∧
M_Conv_Cond/cu,rv,rv,rf,cg,ch; ∧
(Corr_Exp/conc.t,t,cu,rv,rtw,rtw,rf,cg,ch) ∧

Cstmt_Cond[t -> conc.t]/cu,cg,ch\ =>
InitCond[t -> conc.t]/cu,cg,ch\;

Correspondingly, the $T_{Final\ Hyp}$ is:
Assume C_M_Req/cu,cg,ch ∧ M_Req/cu,cg,ch,rv,rf ∧
Conv_Cond(t,cu,ch,rv,rv,rf,cg) ∧
M_Conv_Cond/cu,rv,rv,rf,cg,ch ∧ RT2.Cstrnt(rv) ∧
RT7.Cstrnt(rz) ∧ RT8.Cstrnt(rw);
F_Body;
Confirm M_Conv_Cond/cu,rv,rv,rf,cg,ch;

This hypothesis requires that before the Finalization is called, both the module
conventions and the type conventions hold and that after the Finalization has executed,
the module conventions still hold. Note the symmetry with the initialization
hypothesis which requires that the module conventions hold before the initialization is
executed and that after the execution of the initialization, both the type conventions
and the module conventions hold.

The initialization and finalization routines interact with the system in the
following way:

When a new type is instantiated, there is an automatic call to the module
initialization code. For example, in the case of the Communal Stack, we will show in
the next section an implementation in which type instantiation will result in the setting
up of Stack_Hldr, an array which is shared by all variables of any given stack
instance.

When a variable of that type is declared, there is an implicit system call to the
type initialization procedure. In our Communal Stack implementation this means that
a record with two integer fields will be created, and those fields are automatically initialized to zero, according to the specifications of the primitive integer facility presented in Chapter 4. Also during the initialization, the auxilliary variable \textit{Last Specimen Num} is updated and the each newly declared variable is assigned a unique integer so that a programmer will be able to reason about all variables of the given type.

Correspondingly, at the end of the block in which a given variable is declared, the system calls the finalization procedure which can return to the shared pool of storage the space which has been occupied by the given variable.

V.5. The Role of Adjunct Variables in Correspondences

So far our implementation seems like a natural way to provide flexibility and efficiency in storage management for all the variables of any given stack module instance. Moreover, we have strengthened our module rules so that we can deal with proving assertions about any auxilliary variables as well as the programming variables.

However, we do not want to forget about writing a correspondence which relates our implementation representation to the conceptual one. In this case, we note that the implementation for any stack consists of a record containing just two integers, while the conceptual stack is a string -- two exceedingly different representations. The conceptual string provides a useful abstraction for reasoning about program variables, while the top pointer and depth integers enable us to achieve efficiency. However, we now face the challenge of relating these two representations.

When we look at \textit{Stack_Hldr} we see at each index only one entry of a given stack, and consequently, it will be necessary to follow pointers around to find all the
entries of a given stack.

One way to formalize the idea of extracting the conceptual string from the Stack_Hldr array is to introduce a recursive function. The function Str_Val defined next is such a function.

Str_Val: [0,capacity] -> Str(Item)

Str_Val(i) = A if i = 0
Stack_Hldr(i).Value o Str_Val(Stack_Hldr(i).Next) otherwise

Str_Val assigns to each integer between 1 and capacity the particular string which represents the conceptual stack whose top has the index i. Hence it is now easy to express the correspondence:

conc.S = Str_Val(S.Top)

Now it appears that formal verification can proceed quite simply. However, as is the case whenever a new function is described, in order to maintain mathematical integrity, we must prove that our function is well-defined - - a non-trivial exercise when the function happens to be recursive, as this one is.

In this particular situation we can tackle the problem by writing module conventions which require that the pointers in the Next field not overlap, i.e.,

\forall i,j: [1, capacity], Stack_Hldr(i).Next = Stack_Hldr(j).Next iff i = j or Stack_Hldr(i).Next = 0

Although in any individual case, addressing the total function question in an ad hoc manner may make the job possible, even easy, there is no way to know in general whether a given convention is adequate to guarantee that a particular recursive function is well-defined. Hence, using recursive functions as correspondences introduces a serious complication into our proof system, and we would prefer to avoid that. Since one of our goals is automated application of our proof rules, we do not welcome having to give up this automation because we must examine such correspondences in an ad hoc manner.
Experience with such situations has caused us to focus on another approach for writing correspondences. It is easy to see that our correspondence would be simpler to write if the entire string representing each conceptual stack were available to us in a single unit, rather than all spread out in Stack_Hldr. Of course, the whole purpose in setting up the implementation as we did was to avoid the overhead required for keeping each stack in its own unit, and so to store these stack strings would undercut our efforts at efficiency.

Fortunately, there is a way to have the best of both worlds. We can keep our Stack_Hldr as it is, introduce a new array of stack strings, and still maintain our efficiency. We do this by using what we call adjunct variables.

An adjunct variable is an auxiliary variable introduced into the module purely for the sake of conveniently expressing the relationship between the implementation of a type and its conceptual form. Useful adjunct variables frequently turn out to contain information which, although redundant as far as the implementation goes, is useful for setting up a fit between widely diverse objects in the conceptual and realization world views.

Since the adjunct variable will never be realized, that is, no actual computation will be performed on it, efficiency of manipulation is no longer an issue of importance; rather, redundancy of storage is now not only acceptable but desirable.

In this particular case, as a convenience, we will use as an adjunct variable an array Str_Keeper in which we will keep the strings which represent the stacks whose tops are at any given index into Str_Keeper. Then at any given time every stack which could possibly be manipulated or could occur as a result of popping an existing stack is available as a single entry in Str_Keeper. The declaration of Str_Keeper is denoted by the key word adj_var, a syntactic mechanism for distinguishing between executable program variables and adjunct variables.
Of course, this adjunct array containing all the strings we need to talk about remains useful only as long as it is kept up to date. As we perform operations on the different variables of whatever stack instance we are using, it is essential that the adjunct array `Str_Keeper` reflect these changes. Each time a push is completed, a new entry must be made in `Str_Keeper`, and when there is a pop, that change must also be reflected in the adjunct variable. One way to do this is to write code which theoretically manipulates the adjunct variable. This will have no effect on performance, because the adjunct code will not really execute. We will give more attention to this phenomenon later, but for now, we are ready to write the rest of our realization. Since some of the new parts to be added belong within the code already written, it will be easier to read if we simply rewrite what we have already seen, with the new parts containing adjunct variables incorporated:

Realization Module Shared_Array for Communal_Stack_Template;

Module Entry_Record_Fac is Record_Facility(Item, integer);
    Realized_by Standard;
    renaming P1 as Valu,
P2 as Next;

Module Entry_Array_Fac is Array_Facility(Entry_Record_Fac.Record, capacity)
    Realized_by Standard;

Adj_Module Str_Array is Array_Facility(Str(Item), capacity + 1);
    Realized_by Standard;

var cap, First_Free: integer;
var Stack_Hldr: Entry_Array_Fac.Array;
adj_var StrKeeper: Str_Array.Array

M_Conventions
∀ i: [1, capacity], 0 ≤ Stack_Hldr(i) ≤ capacity ∧
∀ i,j: [a, capacity] if Stack_Hldr(i).Next = Stack_Hldr(j).Next ≠ 0, then
i = j ∧ ∀ i: [1, capacity]
StrKeeper(i) = Stack_Hldr(i).Valu o StrKeeper(Stack_Hldr(i).Next) ∧
StrKeeper(0) = ∅ ∧
cap = |StrKeeper(First_Free)|;
Initialization

uses cap, First_Free, Stack_Hldr, StrKeeper;
var Next_Free: integer;
Set_Bounds(Stack_Hldr, 1, capacity);
Set_Bounds(StrKeeper, 1, capacity);
First_Free := 0;
cap := capacity;
Maintaining ∀ i,j ≤ Next_Free
if Stack_Hldr(i).Next = Stack_Hldr(j) then i = j;
while First_Free < capacity do
  Next_Free := First_Free;
  First_Free := First_Free + 1;
  Stack_Hldr(First_Free).Next := Next_Free;
  StrKeeper := A;
end;
end initialization;

Module Stk_Record_Fac is Record_Facility(integer, integer);
realized by Standard;
renaming PI as Top,
P2 as Depth;

Type Stack = Stk_Record_Fac.Record;

conventions
 ∀ n: [1, Stack.Last_Specimen_Num],
  Stack.Denotation(n).Depth = StrKeeper(Stack.Denotation(n).Top) \^ 
capacity = cap + Aggregate_Length \^ 
 ∀ m,n: [1, Stack.Last_Specimen_Num]
  if Stack.Denotation(m).Top = Stack.Denotation(n).Top ≠ 0,
  then m = n \^ ∀ i: [1, capacity],
  Stack_Hldr(i).Next ≠ Stack.Denotation(n).Top;

initialization
 /* The initialization of the integer fields to zero is 
    automatic. */

finalization
i := S.Top;
while i > 0 do
  var fresh_val: Item;
  Stack_Hldr(i).Valu := fresh_val;
  StrKeeper(i) := A;
  i := Stack_Hldr(i).Next;
end;
First_Free := S.Top;
S.Top := 0;
end finalization;

Correspondence conc.S = Str_Keeper(S.Top);

procedure Push(var Valu: Item; var S: Stack);
var fresh_val: Item;
Str_Keeper(First_Free) := Valu o Str_Keeper((S.Top).Next)
if S.Depth = 0 then
    S.Top := First_Free;
    Stack_Hldr(S.Top).Next := 0;
else
    Stack_Hldr(First_Free).Item := Valu;
    Stack_Hldr(First_Free).Next := S.Top;
    First_Free := Stack_Hldr(Stack_Hldr(S.Top).Next);
    S.Depth := S.Depth + 1;
    Valu := fresh_val;
    cap := cap - 1;
end;

end push;

procedure Pop(var res: Item, var S: Stack);
var fresh_val: Item;
temp := S.Top;
Stack_Hldr(First_Free).Next := S.Top;
Stack_Hldr(S.Top).Next := 0;
S.Top := Stack_Hldr(temp).Next;
Str_Keeper(temp) := A;
end pop;

procedure Capacity( );
    Capacity := cap;
end Capacity;
V.6. Formal Treatment of Adjunct Variables

As we have already mentioned, the use of adjunct variables for writing correspondences helps us to avoid the problem of dealing with the issue of deciding whether or not a given recursive function is total. However, we must still face the challenge of finding a way to keep our adjunct variables up to date.

In the case of our Communal Stack example, we have included code in various parts of the realization for doing this. In both the initialization and the finalization code, there are while loops which go through the global arrays to prepare them for later use.

Obviously then, it is necessary that these loops terminate so that we will be sure that our adjunct array has been properly filled in. Technically, our proofs with nonterminating adjunct calculations could actually be unsound. They could produce incorrect results when the adjunct calculations are removed, since the nonterminating calculations had been masking them.

Because of the need for checking termination of procedures which manipulate adjunct code, we cannot prove that the any implementation which uses adjunct variables meets its specifications until termination rules are available. In the next chapter total correctness rules, along with their proofs of soundness and completeness will be given.
CHAPTER VI
Adjunct Variables and Total Correctness

A chapter about total correctness in the middle of a thesis entitled "The Role of Verification in Software Reusability" may seem, at first glance, out of place. However, in the preceding chapter about Communal Stacks, the idea of using adjunct variables to aid in specification was introduced, and the accompanying explanation and discussion pointed out the reasons why the presence of adjunct variables in a program forces consideration of the issue of total correctness.

In fact, we can see two reasons for examining the issue of termination. The first is the usual concern about the possibility of developing a total correctness proof system. This is an old issue which has been addressed by a variety of people in the context of both sequential and concurrent programs. [2, 14, 22, 33, 28, 25, 37, 41]

The second reason makes the issue even more pressing because it is now clear that even for partial correctness, if a programmer uses adjunct variables which are updated during program execution, termination of this updating code must be established in order to preserve the soundness of the proof system.
VI.1. Total versus Partial Correctness

By definition, a partial correctness system, even if sound and relatively complete, still guarantees correctness of a given program only if the program terminates. Since a verified but non-terminating program is not a very useful thing, proof systems must ultimately confront the issue of termination. Perhaps it should be noted that the development of satisfactory partial correctness proof systems is a necessary precursor for a system which proves termination, because proofs of termination generally depend upon proving the correctness of assertions about various relationships between program variables.

A likely reason why partial correctness proof systems have been worked out and rules for total correctness left for later research is that there is a natural hesitation to attack a problem which has such obvious connections to the halting problem, since this is usually the first problem which computer science students learn to prove is unsolvable. However, relative completeness for total correctness rules does not imply the existence of a solution to the halting problem. Rather, the relative completeness of a total correctness rule guarantees that no incompleteness beyond that already present in the underlying mathematical thesis is introduced by the inclusion of the total correctness rules.

The basic strategy for proving termination is familiar in the literature and involves introducing a progress metric, that is, a mechanism for establishing that when the program is executing a loop, for example, progress toward termination is taking place.
Perhaps the most interesting question here is whether or not for every loop there exists an appropriate progress metric. This question of expressiveness is complicated in our case by the fact that REFORMS must handle complex objects, such as user-defined types which in turn are defined in terms of still other user-defined types. When such objects are introduced into the programming domain, statements in the corresponding theory must be included in our specifications.

By extending these new theories to include strings of new objects in addition to the objects themselves, the way is opened to solve the general termination problem by describing histories of computations using strings of objects. The lengths of these histories can then be used to provide an appropriate progress metric for any loop.

Hence, what is new in our treatment of termination is the discovery of a framework in which every terminating loop and recursive procedure has an appropriate progress metric. Within this improved specificationational framework it is then possible to prove the completeness of our proof rules for while loops and recursive procedures.

VI.2. Termination of While Loops

Most of the simple programming constructs introduce no problems as far as termination is concerned. However, while loops present a challenge. For most normal loops, finding an appropriate progress metric is relatively simple. However, the challenge lies in showing that every terminating loop has such a progress metric.

Before introducing the total correctness while rule, an extension of our programming syntax is necessary so that programmers will be able to include an appropriate progress metric with each loop. While loops will now have the syntactic form:
Maintaining Inv;
decreasing Metric_Name = \text{P}_\text{Exp};
while B do
body;
end;

As previous research has pointed out [38], the progress metric which is given by the $P_{Exp}$ takes values in any well founded set. However, for simplicity, we use the most familiar of these well founded sets, the Natural Numbers. So, evaluating the expression $P_{Exp}$ must yield a natural number whose value will be strictly smaller each time the loop body is executed. Since this value is a natural number, it is bounded below by zero, and hence the loop must terminate.

The following example illustrates this new concept by showing code which includes a while loop. The code reverses a given stack $S$. The progress metric chosen is the length of $S$.

Assume $S_{Reversed} = \Lambda \land |S| \leq \text{max\_depth}$;
Remember;
Maintaining $#S = (S_{Reversed})^R \circ S \land |S| \leq \text{max\_depth}$
decreasing Stack\_Size = $|S|$;
while $\neg \text{Is\_Empty}(S)$ do
    Pop(Next\_Item, S);
    Push(Next\_Item, S_{Reversed});
end;
$S := S_{Reversed}$;
Terminate\_with $S = (#S)^R \land S_{Reversed} = \Lambda$;

In this program segment, the programmer takes a given stack $S$ and reverses it by popping the items off $S$, pushing them onto another stack called $S_{Reversed}$ and then swapping the two stacks. Since our purpose here is to illustrate the new while rule, we will assume that $S$, $S_{Reversed}$, and Next\_Item have already been declared to be of appropriate types. Each time the stack $S$ is popped, the length of $S$ obviously decreases and so it becomes a natural candidate for a progress metric.
Now that we have a new syntactic slot for inserting progress metrics into programs with while loops, we are ready to formulate a proof rule to deal with this construct:

\[ C \text{ code; Terminate with Inv;} \]
\[ C \text{ Assume Metric\_Name } = P_{\text{Exp}} \land B \land \text{Inv;} \text{ body;} \]
\[ \text{ Terminate with Inv } \land P_{\text{Exp}} < \text{Metric\_Name; } \]
\[ \neg B \land \text{Inv } \Rightarrow Q; \]

C\text{ code; Maintaining Inv decreasing } P_{\text{Exp}} \text{ while } B \text{ do body end; Terminate with } Q; \]

To indicate the fact that the rule checks for total correctness, the key word Confirm is replaced by Terminate with. The second hypothesis now includes a check to see that the value of \( P_{\text{Exp}} \) strictly decreases on each pass through the loop body.

Before checking the rule for soundness and completeness, we will illustrate its application to the example code. One important observation is that the terminating while rule does not change the partial correctness rule for while loops, but simply supplements that rule, making it capable of checking termination.

In our example the first rule to apply is the swap rule, and it results in the Terminate with clause being changed to:

\[ \text{ Terminate with S\_Reversed } = (\#S)^R \land S = \Lambda; \]

In proceeding backward through the code, we next come to a while loop, and so we apply our new terminating while rule. As in the case for the partial correctness while rule, we have three hypotheses to check, but the second hypothesis has been augmented by the additional clauses needed for checking termination:

(1) code; Terminate with \#S = (S\_Reversed)^R \circ S \land |\#S| \leq \text{max\_depth;}

(2) Assume Stack\_Size = |\#S| \land \#S = (S\_Reversed)^R \circ S \land
|\#S| \leq \text{max\_depth } \land S \neq \Lambda; \text{ body;} \]
\[ \text{ Terminate with } \#S = (S\_Reversed)^R \circ S \land |\#S| \leq \text{max\_depth } \land |\#S| < \text{Stack\_Size; } \]
(3) $S = \Lambda \land \#S = (S_{Reversed})^R \circ S \land$
$\mid \#S \mid \leq \text{max\_depth} \Rightarrow$
$S_{Reversed} = (#S)^R \land S = \Lambda$;

We begin with (1). Filling in the code, we get:

Assume $S_{Reversed} = \Lambda \land \mid S \mid \leq \text{max\_depth}$;
Remember;
Terminate_with $\#S = (S_{Reversed})^R \circ S \land \mid \#S \mid \leq \text{max\_depth}$;

Applying the rule for Remember removes the $\#$ signs, changing the last clause to:

Terminate_with $S = (S_{Reversed}) \circ S \land \mid S \mid \leq \text{max\_depth}$;

Finally, upon applying the rules for Assume and Terminate_with, we get an implication:

$S_{Reversed} = \Lambda \land \mid S \mid \leq \text{max\_depth} \Rightarrow$
$\exists S = (S_{Reversed}) \circ S \land \mid S \mid \leq \text{max\_depth}$

Since $S_{Reversed} = \Lambda$, the first clause in the conclusion holds. The second clause is obviously true because it appears in the first part of the implication.

Next we prove the second hypothesis:

Assume $\text{Stack\_Size} = \mid S \mid \land S \neq \Lambda \land$
$\#S = (S_{Reversed})^R \circ S \land \mid \#S \mid \leq \text{max\_depth}$;
Pop(Next_Item, S);
Push(Next_Item, S_{Reversed});
Terminate_with $\mid S \mid < \text{Stack\_Size} \land$
$\#S = (S_{Reversed})^R \circ S \land \mid \#S \mid \leq \text{max\_depth}$;

Applying the procedure call rule to Push gives the following Terminate_with clause:

$\mid S_{Reversed} \mid < \text{max\_depth} \land \forall \exists \text{Next\_Item: Item},$
$\exists S_{Reversed}: \text{Stack}, \exists S_{Reversed} = \text{Next\_Item} \circ S_{Reversed} \land$
Item.init(Next_Item) $\Rightarrow \mid S \mid < \text{Stack\_Size} \land$
$\#S = (S_{Reversed})^R \circ S \land \mid \#S \mid \leq \text{max\_depth}$;

Next we must apply the call rule to Pop. This results in the following Terminate_with clause:

$S \neq \Lambda \land \forall \exists \text{Next\_Item: Item},$
S: Stack, S = ??Next_Item o ?S => |?S| < Stack_Size ∧
|S_Reversed| < max_depth ∧ |#S| = (??Next_Item o S_Reversed)^R o ?S ∧
|?S| ≤ max_depth;

Simplifying and applying the rules for Assume and Terminate_with, we get:

Stack_Size = |?S| ∧

#S = (S_Reversed)^R o S ∧ |#S| ≤ max_depth ∧

S = ??Next_Item o ?S => |?S| < Stack_Size ∧
|S_Reversed| < max_depth ∧ |#S| = (??Next_Item o S_Reversed)^R o ?S;


The last conclusion follows by substituting for S in the equation:

#S = (S_Reversed)^R o S
= (S_Reversed)^R o (??Next_Item) o ?S
= (S_Reversed)^R o ??Next_Item) o ?S
= (??Next_Item o S_Reversed)^R o ?S

The second conclusion follows by substituting for #S and S in the inequality:

|#S| ≤ max_depth

|(S_Reversed)^R o S| =

|??Next_Item o |#S| o (S_Reversed)| =

|S_Reversed|^R| + |??Next_Item o |S| | =

|S_Reversed| + |??Next_Item| + |?S| ≤

|S_Reversed| + 1 ≤

|S_Reversed| <

Finally, we examine the third hypothesis. We need to show:

Assume S = Λ ∧ #S = (S_Reversed)^R o S ∧
|#S| ≤ max_depth;
Terminate_with S_Reversed = (#S)^R ∧ S = Λ;

Since S is empty, this is obvious.

Just as in the case of applying the partial correctness rules, we find that in using the total correctness while rule, the mathematical expertise necessary for completing the proof is quite simple. In this case, since stacks are defined in terms of string theory, our proof relied on taking lengths of strings and manipulating those lengths as integers.
Having seen an application of the rule to an example, we turn now to checking the rule for soundness and relative completeness. In the chapter which describes REFORMS, it was pointed out that a program is considered valid if, having started in a state with assert status neutral, the execution of the program does not change the AS to CF. The assert status explained in Chapter 2 may now take on any of four values: N, VT, CF, and ⊥. The first three have been discussed. The ⊥ indicates non-termination.

The semantics of the usual programming constructs are given elsewhere [16], but the Terminate_with and the decreasing - while statements are new, so further discussion of their semantics is required.

The Terminate_with statement is almost like the Confirm statement; the only difference is that non-termination, ⊥, leads to categorically false. Formally, for any state S: State the semantic function, Sem, gives a state transition defined by:

\[
\text{Sem}(\text{Terminate_with } Q(S)) = S'
\]

where

\[
S = <\text{St},\text{AS}> \text{ and } S' = <\text{St},\text{AS}'> \land
\]

\[
\text{AS' = CF if } \text{AS = } \bot \text{ or }
\]

\[
\text{AS = N } \land \text{ Eval}(Q,\text{St}) = \text{ false}
\]

\[
\text{AS otherwise}
\]

As is customary for while statements, the semantics are defined in terms of the minimal fixed opint of a functional (WF. Specifically: Let S denote an arbitrary state and AS its assert status.

\[
\text{Sem(Maintaining Inv decreasing P_Exp while B do body end)}(S) = \text{Least_Fixed_Point}(WF),
\]

where

\[
WF(F)(S) = \text{Sem}(\text{Terminate_with Inv})(S)
\]

if \(\text{AS ≠ N} \text{ or } \text{Eval}(B,S) = \text{ false}
\]

\[
F(\text{Sem(body; Terminate_with Inv } \land
\]

\[
P_\text{Exp } < \text{Metric_Name})(S)) \text{ otherwise}
\]

The soundness of proof systems is usually easier to establish than their relative completeness, so I begin there. Proving soundness is simply a matter of showing that
every rule in a proof system preserves validity. The proofs of soundness for most of
the rules in REFORMS need only trivial modifications, when we go from partial to
total correctness. However, the while rule does require consideration.

**Theorem 1:** The Decreasing_While rule is sound.

**Proof:** Suppose the hypotheses of the rule are true. We need to show the validity
of the conclusion. In our system validity is established by showing that if the assert
status (AS) in not categorically false (CF) in state S (the state before a particular
construct is carried out), then the execution of that construct will not change AS to CF.
So we assume that AS ≠ CF upon completion of the code preceding Maintaining Inv ...
and consider what might cause the AS to become CF. Our semantics for
Terminate_with and Maintaining ... show clearly the circumstances in which having
begun at state S and proceeding to S’ by executing the loop AS’ = CF even though AS
≠ CF.

From the semantics for Maintaining ... we see that AS can become CF in case
Inv fails for Eval(B,S) = false or when Eval(B,S) = true, if P_Exp ≥ Metric_name or
Inv fails. But the hypotheses specifically provide that none of these possibilities can
happen.

From Terminate_with, AS’ = CF in case AS = ⊥ or eval(Q,St) = false but the
hypotheses include the requirement that ¬ B ∧ Inv => Q.

Hence, we have soundness. QED
VI.3. Completeness

Once we know that a system is sound (doesn't prove things it shouldn't), it is desirable to know that it is capable of proving everything it should (i.e., that it is relatively complete). Specifically, this involves showing that any valid assertive program can be proved correct in our system. Since REFORMS has exactly one proof rule for each programming construct, the portion of a proof involving the programming language constructs is unique. Being syntax driven, the system produces for any particular program, a fixed set of the verification conditions in the underlying mathematical theories. Following Cook, we can say that our proof system is complete relative to the underlying mathematical theories if all of verification conditions for a valid program are necessarily valid in the underlying theory -- which is to say that, if, contrary to the usual situation, there were a complete verification system for the underlying mathematical theory, then the system would be complete.

In a syntax driven verification system which is sound, the only way to establish relative completeness is to show that each proof rule preserves validity in the reverse direction; i.e., to show for each rule that, if the conclusion is valid, there is a way to choose the specification parameters (loop invariants, progress metrics, post conditions, etc.) so that all of the hypotheses are valid. Clearly, if we start with a valid program and generate only valid lemmas at each step of the proof discovery process, the verification conditions which are ultimately generated will be valid.
The proof that the proof rules for the elementary programming constructs preserve validity in the reverse direction is the same for total correctness as it is for partial correctness. The only interesting issue involves the while statement. The rule here requires the introduction of a suitable loop invariant and a satisfactory progress metric.

Cook and Oppen [50] observed that if the underlying mathematical theory is computationally expressive (e.g., number theory), then you can always find a strongest loop invariant which exactly describes what happens in the loop. This invariant will ensure that all states that come up for consideration in the hypotheses for the While rule can be shown to be harmless because they can potentially arise in the conclusion, and it is known to be valid. Cook also pointed out that certain theories (e.g., Presberger Arithmetic) are not computationally expressive, so that adequate loop invariants (and progress metrics) may not exist in these theories. This poses a serious problem, since abstract data structures may well introduce such computationally inexpressive theories into our specification languages. One solution lies in extending each of our underlying mathematical theories to include the theory of strings over that theory. The resulting theories are all computationally complete [16], and so we are guaranteed to be able to express the strongest loop invariants which we need in order to get relative completeness.

The new question raised by attempting to show the relative completeness of the terminating while rule is whether, for any terminating loop, there exists a natural number valued expression which could be used to prove that the loop terminates.

To show the existence of such an expression, I must provide some language in which to describe all valid programs and then to show how to use such descriptions to find progress metrics for while loops. For single type theories there is standard notation available, but because REFORMS supports user-defined types based on a variety of mathematical theories, there is a need to find a representation which permits
one to talk about this domain of many types. To do this I introduce an operational
description of what happens to objects of various types under the application of
programming constructs. This is achieved by using strings of configurations which
represent information at each step of the program.

In REFORMS a valid program terminates with a neutral assert status. Using the
string representation just mentioned, I will prove a theorem which establishes that
having a valid program is equivalent to having a string of "correct" configurations
representing that program with respect to any given starting state. To get a progress
metric for any loop, one can use the length operator on strings. Finally, to show
completeness of the total correctness version of REFORMS, it remains to show that
every valid assertive program has a proof of correctness. But since every valid
program has a "correct" string of configurations (a correct history of computation),
which can be used to specify the program, induction on the program structure achieves
the proof of completeness: one assumes a "correct history" for all but the last
statement of a given program and then checks each possibility to see that whatever the
final statement is, including a while loop which needs a progress metric, a final string
can be concatenated to the history so far to produce a correct history. The details
follow.

VI.4. Computational Histories

At this point we set up notation, make definitions and provide other mathematical
machinery to facilitate the statement and proof of our existence lemma, and,
ultimately, the relative completeness of our rule. Specifically, we need to talk about
an arbitrary assertive program ending with a while loop (i.e., a Code Preceded Loop).
We will also need to talk about the same code preceded loop when a progress metric
Exp has been added at the appropriate point.

Let CPL = "code; Maintaining Inv While B do Loop_Name: body end;"
Let CPL' = "code; Maintaining Inv Decreasing Exp While B do Loop_Name: body end;"

CPL' is just CPL with the the phrase Decreasing Exp added at the appropriate place.

Definition: An assertive program, P, is said to terminate naturally starting in state S, if when started in state S, P terminates with assert status Neutral.

Definition: A natural number valued expression, Exp, whose arguments are program variables is a proper progress expression for CPL, if, for any state S, whenever CPL terminates naturally, starting in S, CPL' terminates naturally, starting in S.

Loosely speaking, this definition says that the expression Exp strictly decreases each time we go around the loop in those cases for which the loop will terminate.

VI.4.1. Use of Strings

Our plan for using strings as a tool to create expressiveness over arbitrary theories includes performing the following:

- Elaborate our notion of assertive programs.
- Develop notation for describing a valid program using "histories of computation."
- Link these history strings to the semantics of the programs they are associated with.
- Show that every valid program can be specified using histories. (Specifying includes providing the progress metric).
VI.4.2. How Programs Look

In Chapters 2 and 3, there were several examples which illustrated the syntax of our language. This syntax, as we pointed out, includes keywords, such as Assume and Confirm, which provide positions in the programs for writing specification clauses. So far, as we have looked at various examples, our focus has been directed toward the language constructs, the assertions, and the proof rules which we use for establishing program correctness.

In this chapter our main goal is to establish the existence of progress metrics for terminating while loops and recursive procedures, and for this purpose, it will be useful to view our programs as sequences of labeled constructs, accompanied by appropriate assertions. (The assertions are not labeled). We will also emphasize here the fact that there will be particular states associated with different positions within the code:

- (initial state $S_0$)
- $L_0$ code
- ...
- (final state $S_{\text{final}}$)
- $L_n$ end
- postcondition

At the beginning is some initial state $S_0$ and between any two constructs there is an associated state.
VI.4.3. Notation and Terminology

In order to show completeness for the while rule, it is necessary to show that for every while loop there is a progress expression as called for in the rule. We do this by using strings to describe assertive programs. This use of strings gives us a symbolism in which to describe the behavior of objects of arbitrary types when acted upon by a program, hence introducing the use of the simple theory of strings to use for expressing and proving results involving complex objects.

First we select the symbols which are to be used as string components. We call them "descriptors" and explain when two descriptors may be correctly concatenated relative to some given programming language construct.

Since our strategy for dealing with termination is to apply the length operator to strings which represent computation histories in order to obtain an expression whose value decreases with each loop iteration, we need to define precisely what we mean by a computation history. We do this by giving what is in effect an operational specification of the execution of our language. Our language includes procedure calls, if then else, while, assignments and declarations.

Let AV represent the set of Active Variables.
Define: Descriptor = Cartesian Product
\[
\text{Lab: Label; Val}_\text{Rec}: \text{Cartesian Product} \\
\phantom{\text{Lab: Label; Val}_\text{Rec}: \text{Cartesian Product}} \Pi (\eta: \text{Val}); \eta \in \text{AV} \\
\phantom{\text{Lab: Label; Val}_\text{Rec}: \text{Cartesian Product}} \Pi (\#\eta: \text{Val}); \eta \in \text{AV} \\
\text{end; Val}_\text{Rec}: \text{Cartesian Product} \end{align*}

A descriptor is a pair, the first entry of which is a label and the second of which is the collection of variable values, both current and old, those preceded by #.
Define: Agreement Descriptor predicate AD(D).

Let D be a descriptor. \( AD(D) = \land D.\text{Val}_\text{Rec}.\eta = \eta \land \text{AV} \)

This predicate evaluates to true in any state on which that state and the active variables (AV) mentioned in the D.\text{Val}_\text{Rec} agree. Note that this requires us to view states as falling into equivalence classes according to agreement. \( S \equiv T \iff \exists C: \text{Descriptor such that} \quad \text{Eval}(AD(C),S) = \text{true} \land \text{Eval}(AD(C),T) = \text{true} \).

We use the notation Des_Str to represent the set of all descriptors. Intuitively, these strings of descriptors give an operational view of a given assertive program. We can think of them as histories of program behavior.

Intuitively, we develop an operational view of what it means to have a correct history to describe a piece of code. To do this we proceed inductively, first with a base case which tells what string is associated with a program consisting only of a label. Then we consider each construct of the language and describe its associated string.

We now define what it means to have such a history be correct relative to some code.

Definition: Correct History Formula (CHF(code)\(\delta\)) for \(\delta: \text{Des}_\text{Str}\) by induction on the structure of code.

(i) \( \text{CHF(L:)\(\delta\)} \iff \exists C: \text{Des} \text{ such that} \quad \delta = C \land C.\text{Label} = L. \)

(ii) \( \text{CHF(code;L:st;J:)(\delta)} \iff \exists \alpha, \beta: \text{Des}_\text{Str}, C,D: \text{Des} \text{ such that} \quad \delta = \alpha \circ C \circ \beta \circ D \land \text{CHF(code;L:)[}\delta \Rightarrow \alpha \circ C]\land \text{CSTF(st)(C \circ \beta \circ D)} \)

where CSTF represents Correct State Transition Formula which is defined case by case as follows:

Case: \( \text{st} = "x := \text{exp}" \)
\( \beta = \land \land D.\text{Label} = J \land \quad \text{D.\text{Val}_\text{Rec}.}\xi = C.\text{Val}_\text{Rec}.\xi \quad \text{if} \; x \neq \xi \)
\( \text{exp}[v \Rightarrow C.\text{Val}_\text{Rec}.v] \)
if $x = \xi$

Case: $st = "If \ B \ then \ code1 \ else \ code2"

$\forall v: AV$
if $B[v \rightarrow C.\text{Val}_{\text{Rec}}.v]$ then
$\text{CHF}(L: \text{code1}; J)[\delta \rightarrow C \circ \beta \circ D]$
Otherwise
$\text{CHF}(L: \text{code2}; J)[\delta \rightarrow C \circ \beta \circ D]$

Case: $st = \text{p}(a)$;

Case: $st = "\text{While } B \ \text{do } \text{body end;}"$

$\forall v: AV$
if $\neg B[v \rightarrow C.\text{Val}_{\text{Rec}}.v]$
then $D.\text{Label} = J \land D.\text{Val}_{\text{Rec}}.\xi = C.\text{Val}_{\text{Rec}}.\xi$
Otherwise,
$\forall \sigma: \text{Str}(\text{Des}_{\text{Str}}), \forall G, H: \text{Des}_{\text{Str}}$
$\rho, \tau: \text{Str of } \text{Des}_{\text{Str}}, \mu, \gamma: \text{Des}_{\text{Str}}$
If $\sigma = \rho \circ <\gamma \circ G> \circ <\mu \circ H> \circ \tau,$
then $\text{CHF}(L: \text{body}; J)[\delta \rightarrow G \circ \mu \circ H]$ and $C \circ \beta \circ D = \Pi \sigma$

What we now must do is show that this descriptor view of any given assertive program is compatible with the denotational semantics of that program.

Lemma 1: $\forall code \ \forall S, T: \text{State such that } S.\text{AS} = N \text{ and } T.\text{AS} = N$, $\text{Sem}(code)(S) 
\equiv T$ iff $\exists \delta: \text{Descriptor}_{\text{String}}$ such that $\delta = C \circ \delta'$ and $\text{Eval}(A_D(C), S) = \text{true and } \delta = \delta'' \circ D$ and $\text{Eval}(A_D(D), T) = \text{true and } \text{CHF}(code)|_\delta$.

Proof by induction on the structure of code:
<= Base: code = "L:"

Given $\text{CHF}(L:)\|_\delta$, by definition of CHF, $|\delta| = 1$. So $\delta$ consists of a single descriptor, say $C$, using the notation of the lemma where $C.\text{Label} = L$. Moreover, $C = D$ and $\delta' = \delta'' = \Lambda$. $\text{Eval}(A_D(C), S) = \text{true } \land \text{Eval}(A_D(D), T) = \text{true}$. Then $S = T'$, so $S = T$. Therefore $\text{Sem}(L:)(S) \equiv T$.

Induction step:

Assume that $\delta: \text{Des}_{\text{Str}}$ satisfies the conditions of the lemma for $L: \text{code1}; J:st; K:$ and that $\exists U: \text{State such that } \text{Sem}(L: \text{code1}; J:)(S) = U$ and that $S.\text{AS} = N = U.\text{AS}$. We need to show that $\text{Sem}(L: \text{code1}; J:st; K:)(S) \equiv T$ and that $T.\text{AS} = N$. Using the
definition of CHF, we know that \( \exists \alpha, \beta: \text{Des} \cdot \exists \phi: \text{Des} \cdot \alpha \circ \beta = \beta' \circ D \wedge \text{CHF(}L: \text{code}1; J:) \[ \delta \rightarrow \alpha \circ E \] \) Hence, we need to show \( \text{Sem}(J; st; K;)(U) \equiv T \). We now consider each possible statement, \( st \).

**Case: \( st = \text{"exp"} \)**

We know \( \text{Sem}(L: \text{code}1; J;)(S) = U' \equiv U \). We need \( \text{Sem}(J; st; K;)(U) \equiv T \). By definition of \( \text{CHF(}code) \[ \delta \], \( \delta = \alpha \circ E \circ \beta \circ D \) and \( \beta = \Lambda \). \( E.\text{Val}_\text{Rec} \[ \xi \] \) if \( x \neq \xi \) \( D.\text{Val}_\text{Rec} \[ \xi \] = \{ \text{exp}[v \rightarrow E.\text{Val}_\text{Rec} \[ v \] ] \} \) if \( x = \xi \).

Choose \( T^* \): State such that \( \text{Sem}(J; st; K;)(X) = T^* \). Then \( U'.\text{St} \[ \xi \] \) if \( x \neq \xi \) \( T'.\text{St} \[ \xi \] = \{ \text{Eval(exp,}U') \} \) if \( x = \xi \).

By hypothesis, \( D.\text{Val}_\text{Rec} \[ \xi \] = T.\text{St} \[ \xi \] \) on active variables. By definition, \( \text{Eval(exp,}U') = \text{exp}[v \rightarrow U'.\text{St} \[ v \] ] \) So \( T' \equiv T \).

**Case: \( st = \text{"If } B \text{ then code2 else code3;"} \)**

We already know \( \delta = C \circ \delta' = \delta' \circ D \) where \( \text{Eval(AD(C),}S) = \text{true} \wedge \text{Eval(AD(D),}T) = \text{true} \) By induction hypothesis, \( \exists U: \text{State such that } \text{Sem}(L: \text{code}1; J;)(S) = U \). Need: \( \text{Sem}(J; st; K;)(U) \equiv T \). By definition of \( \text{CHF} \), \( \exists \alpha, \beta: \text{Des} \cdot \exists \phi: \text{Des} \cdot \alpha \circ E \circ \beta \circ D \). \( \text{Eval(AD(E),}U) = \text{true} \). If \( \text{Eval(B,}U) = \text{true} \), then \( \text{CHF(code2)}[\delta \rightarrow \beta \circ D] \). Otherwise, \( \text{CHF(code3)}[\delta \rightarrow \beta \circ D] \). By induction hypothesis the required semantic condition holds for code2 and code3. Hence it holds for code.

**Case: \( st = \text{"While } B \text{ do body end;"} \)**

Because \( \text{CHF(}code\)[\delta] \) is given, it follows that \( \text{CHF(}J; st; K;)[\delta \rightarrow E \circ \beta \circ D] \). \( \text{So, } E \circ \beta \circ D = \Pi \sigma \) where \( \sigma = \rho \circ <\gamma \circ G> \circ <\mu \circ H> \circ \tau \) for \( \rho, \tau: \text{Str of Des} \cdot \text{Str}, G,H: \text{Des}, \gamma, \mu: \text{Des} \cdot \text{Str} \wedge \text{CHF(body)[} \delta \rightarrow G \circ \mu \circ H] \). Choose \( W,V: \text{State such that } \text{Eval(AD(H),}W) = \text{true} \wedge \text{Eval(AD(G),}V) = \text{true} \). Assume that \( \text{Sem(}J; st; K;)(W) \equiv T \). Consider \( \text{Sem(}J; st; K;)(V) \). If \( \text{Eval(B,}V) = \text{false} \), then \( \tau = \Lambda, H = D, \) and \( \mu = \Lambda \). So \( \text{Eval(AD(D),}V) = \text{true} \). Also, \( \text{Sem(While } B \text{ do body end;)}(U) = U \). But, since
Eval(AD(D),T) = true, U ≡ T.

Let Sem(While B do body end) = F If Eval(B,V) = true, then Sem(While B do body end;) = F(Sem(body)(V)) ≡ T.

=> Given Sem(code)(S) = T where S.AS = N and T.AS = N we must show that ∃ δ: Des_Str ∧ C,D: Descriptor such that δ = C o δ' and δ = δ'' o D such that CHF(code)\(\delta\). The proof is by induction on the structure of code.

Base: code = "L:" Sem(L:)(S) = S. So S = T. Choose C: Des so that C.Label = L. δ = C. Now δ = C o Λ = Λ o C. Eval(AD(C),S) = Eval(AD(D),S) = Eval(AD(D),T). Hence, CHF(L:)/\(\delta\).

Induction step: code = "L: code1; J: st; K"

Assume ∃ C,E: Desc and α: Des_Str, and U: State such that CHF(L: code1; J:)/\(\alpha\) and Sem(L: code1; J:)(S) = U. We need β: Des_Str and D : Des such that δ = C o α o E o β o D.

Case: st = "x := exp"

Let β = Λ. Choose D.Val_Rec so that Eval(AD(D),T') = true. Choose T' so that T.St.ξ = U.St.ξ if ξ ≠ x Eval(exp,U) if ξ = x Then T' ≡ T.

Case: st = "If B then code2 else code3;"

By induction hypothesis, ∃ μ: Des_Str for code2 satisfying the lemma conditions, including Sem(code2)(U) ≡ T for μ and Sem(code3)(U) ≡ T for ν. Choose β = μ for Eval(B,U) = true and β = ν for Eval(B,U) = false.

Case: st = "While B do body end;"

Because semantics for the program are defined and not ∅, the semantics for While B do body end; are defined. Let F = Sem(While B do body end) Then, ∃ n: N such that F = F_n where F_i represents the ith approximation to F.

By the induction hypothesis, ∀ V,W: State such that V.AS = N ∧ W.AS = N Sem(body)(V) = W iff ∃ G,H: Des, μ: Des_Str such that CHF(body)[δ → G o μ o H]
\( \land \) \( \text{Eval}(\text{AD}(G),V) = \text{true} \land \text{Eval}(\text{AD}(H),W) = \text{true} \).

If \( \text{Eval}(B,U) = \text{false} \), let \( \beta = \Lambda \). Let \( D.\text{Label} = K \land D.\text{Val}_\text{Rec.} \xi = E.\text{Val}_\text{Rec.} \xi \). In this case, \( \text{Sem}(\text{While } B \text{ do body end};)(U) = U \) and hence \( \text{Eval}(\text{AD}(D),T) = \text{true} \).

So, by definition of \( \text{CHF} \), \( \text{CHF}(\text{code}) \subseteq \Delta \).

If \( \text{Eval}(B,U) = \text{true} \), since semantics for the body exist, \( \exists V: \text{State such that} \text{Sem}(\text{While } B \text{ do body end};)(U) = V \) and hence, by induction hypothesis, \( \exists G,H: \text{Des,} \mu: \text{Des}_\text{Str} \) such that \( \text{CHF(body)}[\delta \rightarrow G \circ \mu \circ H] \).

In fact, \( \forall i < n \exists V_i: \text{State such that} \text{Sem}(\text{While } B \text{ do body end};)(U) = V_i \) and \( \exists G_i, H_i: \text{Des,} \mu_i: \text{Des}_\text{Str} \) satisfying the conditions of the lemma. Call \( \sigma = \Pi <G_i \circ \mu_i \circ H_i> \). Then \( \text{CHF(body)}[\delta \rightarrow <G_i \circ \mu_i \circ H_i>] \). Let \( \beta = \Pi <G_i \circ \mu_i \circ H_i> \). Let \( D.\text{Label} = K \). Let \( D.\text{Val}_\text{Rec.} \xi = H_{n-1}.\text{Val}_\text{Rec.} \xi \). Now, \( \delta = C \circ \alpha \circ E \circ \beta \circ D \) and \( \text{CHF}(\text{code}) \subseteq \Delta \). QED

There is now an established relationship between the denotational semantics for a given program and an operational view of that program given by means of descriptor strings.

What remains is to use this connection to show completeness of the termination rule for while loops, i.e. any valid assertive program has a proof. If a given assertive program, \( \text{code} \), is known to be valid, that means \( \forall S,T: \text{State such that} S.\text{AS} = N \text{ if} \text{Sem(\text{code})(S)} = T \text{ then} T.\text{AS} \neq \text{CF} \). What we must do is provide necessary assertions at appropriate points in the program so that, upon applying proof rules of our verification system, we can prove correctness of \( \text{code} \). We take advantage of our descriptor strings to write these assertions.

One additional piece of mathematical machinery is needed to make use of the descriptor strings now defined and linked to semantics. Since for any given starting state a valid program has a descriptor string, the question of choosing a string length requires us to take a maximal length over the possible descriptor strings which result
depending on various starting states. To do this we use an operator on predicates in the spirit of Kleene's minimization operator, $\mu$. However, this max operator produces total functions.

Suppose that $P(x,n)$ is a predicate defined for objects $x$ of an arbitrary type $T$ and for arbitrary natural numbers $n$. When applied to $P$, the operator $Y$ produces a function from objects $x$ of type $T$ to natural numbers: $Y n P(x,n) = f(x)$. The value of this function is zero if $P(x,n)$ holds for arbitrary large values of $n$; it is one if $P(x,n)$ holds for no values of $n$; and it represents two more than the maximum value of $n$ for which $P(x,n)$ holds, otherwise. Formally:

- If $Y n P(x,n) = 0$ then $\forall m: N \exists k > m$ such that $P(x,k)$.
- If $Y n P(x,n) > 0$ then $\forall m: N$
  - if $m + 2 = Y n P(x,m)$ then $P(x,m)$
  - if $m + 2 > Y n P(x,m)$ then $\neg P(x,m)$.

To find a progress metric for an arbitrary loop we set up a function which tells for any state $S$ how many times the loop body will be repeated, if the loop is started in $S$.

Using what is defined as a repeat state, $S$, and the descriptor string associated with a given loop having started in state $S$, we can formulate a predicate $P$ over $n$, the length of the remaining part of the descriptor string, $v$ and $CHF_{while\_code}(v)$.

Intuitively, the idea is very simple: One asserts that there exists a correct history for the given valid program and then uses the fact that semantics for the program exist, and hence, by the lemma relating semantics to descriptor strings, the required string exists.

To fill in the details for this proof it is necessary to define what we mean by an exact specification for a program.

Exact Precondition: $Epre$ is an exact precondition for "code" iff $\forall S$ such that $S.AS = N$, $Sem(\text{Assume } Epre<v>)(S) = S'$ and $S'.AS = N$ iff $Sem(\text{Code}; Post)(S) = S'$. 
Exact Postcondition: Epost is an exact postcondition for "code" iff ∀ S" such that S".AS = N, Sem(Confirm Epost<ν>)(S") = S' and S'.AS = N iff ∃ S such that Sem(code)(S) = S'.

Before defining exact loop invariant, I introduce the concept of "repeat state."

For a program

preloopcode;
loop_body;

∀ S₀, R: State, R is a "repeat" state iff ∃ n: N such that R = Sem(While B do loop_body)° o Sem(preloopcode)(S₀)

Exact Loop Invariant: E_loop is an exact loop invariant iff ∀ R: Repeat State, R': State if R.AS = N, then Sem(Maintaining E_loop)(R) = R' ∧ R'.AS = N iff ∃ S: State such that Sem(While B do loop_body)(R) = S ∧ S.AS = N.

Having defined exact pre and post conditions for any given valid program and having defined exact loop invariant, my strategy is to now present pre and post conditions and loop invariant in terms of Descriptor Strings, to show that these "descriptor" specs are exact specs, and to then establish that every valid program has a proof by using these "descriptor" specs.

The postcondition of any valid program stated in terms of descriptor strings is straight forward:

∀ code ∃ ν: Des_Str such that CHF(code)(ν).

The precondition is:

∀ code ∀ S: State such that S.AS = N 
∃ ν: Des_Str such that ν = D o ν ∧ Eval(AD(D), S) = true ∧ CHF(code)(ν).

We use the convenience of the repeat state definition to get a loop invariant for a while loop having "body" as the code of its body:

∀ S: Repeat_State ∃ γ: Des_Str such that 
γ = D o γ ∧ Eval(AD(D), S) = true ∧
The specifications written in terms of Descriptor Strings are exact propositions.

Proof: By using Lemma 1, we relate descriptors strings for any code to the semantics of that code. Having been given code with $v: \text{Des}_\text{Str}$ such that $\text{CHF}(\text{code})(\gamma)$ lemma 1 tells us that $\forall S''$: State such that $S''.\text{AS} = N$,

$$\text{Sem}(\text{Confirm CHF(code)}(\gamma)(S'')) = S'$$ and $S'.\text{AS} = N$ iff

$$\exists S \text{ such that Sem(code)}(S) = S'$$ and $S.\text{AS} = N$.

This is what it requires to have an exact spec.

The proofs for precondition and loop invariant are similar.

Lemma 2 has the immediate application of making it possible to write an exact spec for any valid program. However, there is another significant outcome of this lemma. Expressiveness is achieved without resorting to the use of representations which are completely different from the variables which need to be described. Frequently people have achieved expressiveness by using graphs, which have a very complicated theory and no obvious connection to the variables, conceptual or real.

However, the descriptor strings permit expressiveness in a natural and uncomplicated way. The descriptors actually include the variables in question and expressiveness is achieved by using the simple theory of strings over these variables.

The concept of exact specification is useful in several ways. Since "descriptor" specs are exact, they give a precise operational view of any valid program. But in addition to this operational view, since exact specs tell what and only what a program does, reasoning about the proof system can be done, when useful, in terms of exact specs without having to consider weakest pre or strongest post conditions.

If a program has an exact pre condition, then clearly any valid assertion is strong enough to imply that exact precondition. Symmetrically, any exact post condition is strong enough to imply any valid post condition.
With these observations in mind, one can approach the completeness issue with ease. To show that a set of proof rules is complete, one must establish that every valid program has a proof. An obvious strategy is to induct on the structure of the program. One assumes that there is a proof for all except the last statement. In REFORMS, since each construct corresponds to exactly one rule, it is sufficient to assume validity of the conclusion of a given rule and to establish validity of the hypotheses, since, if any proof exists, it must take the form of those hypotheses.

For all the constructs, except the while loop, establishing the necessary hypotheses follows directly from the fact that every valid program has an exact spec. Since there is an exact spec, any valid assertion about the program can be proven since the exact spec tells everything about the program. The while statement raises two questions of existence because the rule requires that the programmer supply both a loop invariant and a progress metric.

Because we have already shown expressiveness of Descriptor Strings as specs, the existence of loop invariants has been doubly established: one can also use Cook's results [50]. We have actually exhibited descriptor loop invariants. To get a progress metric, one takes advantage of the length operator on strings and the maximal operator on predicates, $Y$. Since for any starting state, $S$, each loop has a descriptor string which represents it and since the labels in the descriptors make it possible to identify sections of the string representing loop executions, one can apply the maximal function $Y$ to a predicate whose arguments include length of a particular string and the correct history formula to get a progress metric. This length exists for all valid programs since a valid program is one which terminates in a state with $AS = N$. Obviously, as the execution of the while loop progresses, the length of the string representing that execution decreases. Moreover, the length is a natural number bounded below. Let the predicate $P$ be:
Given a loop, while_code,
\[ \forall S_R: \text{Repeat State}, \forall \alpha, \beta, v: \text{Des}_\text{Str}, \]
D: Descriptor such that \( v = \alpha \circ \beta \wedge \beta = D \circ \beta' \wedge \text{Eval}(AD(D), S_R) = \text{true} \),
|\beta| = n \wedge \text{CHF}_{\text{while}_\text{code}}(\beta)

Choose as progress metric:

\[ Y_n \text{ such that } P(n, \text{CHF}_{\text{while}_\text{code}}(\beta)) \]

This gives a progress metric for every while loop and hence we have completeness for the termination while rule.

**VI.5. Application of the Termination While Rule**

At this stage REFORMS has the capability of providing proof rules for both partial and total correctness. In the case of the stack, since there were no iterations, no distinction needed to be made. In the next chapter I will present a proof in which total correctness is essential.

Before applying the total correctness while rule, in order to complete the discussion of total correctness rules, let us now examine recursive procedures.

**VI.6. Recursive Procedures**

Now that the details concerning what is needed to establish termination for while loops, the next reasonable step is to look at the question for recursive procedures. Because it is simpler to understand the general (mutually recursive) case by first examining the special case of self recursion, I present the termination rules first for self recursive procedures and then generalize to the mutually recursive case.
In REFORMS, one feature of the declare rule is that the hypothesis requiring the
correctness of the procedure body includes the heading of the procedure in its context
portion, thereby providing the information needed for verifying calls of the procedure
to itself.

To check for termination of the while loop construct, a proper progress
expression was introduced which decreased each time the loop body was executed.
Putting in an analogous progress expression for recursive procedures, an expression
which will be smaller each time the procedure calls itself seems a reasonable way to
approach recursive procedures.

In the cases requiring that a procedure terminate, our syntax takes the following
form:

\[
\text{Proc } p(\text{var } x: \text{T});
\text{ require pre;}
\text{ ensure post;}
\text{ decreasing } p\_\text{Exp;}
\text{ body}
\text{ end;}
\]

The declaration rule for termination is:

\[
C \cup \{p\_\text{heading}\}\ \text{Remember; Assume pre } \land p\_\text{bound} = P\_\text{Exp; body;}
\text{ Terminate with post}
C \cup \{p\_\text{heading}\}\ \text{Assume } p\_\text{bound} = \omega; \text{ code; Terminate with Q;}
\]

\[
C\ \text{proc } p(\text{var } x: \text{T}); \text{ require pre; ensure post;}
\text{ decreasing } P\_\text{Exp; body; end; code; Terminate with Q;}
\]

In our termination rule, we associate with procedure \( p \) a variable, \( p\_\text{bound} \) which
is used to hold a target value of the progress expression. As the rule indicates,
\( p\_\text{bound} \) is initially set to \( \omega \). This is done so that at each call, including the first, the
value of \( P\_\text{Exp} \) will be strictly less than \( p\_\text{bound} \). Note the use of \text{Terminate with},
rather than \text{Confirm} in this rule, indicating that the procedure body must terminate in
order to be correct. The other hypotheses of the rule are the same as those of the partial correctness rule previously introduced.

In our call rule we will need to check that \( p.\text{Exp} \), upon evaluation at call time, with the appropriate parameters, is strictly less than the value of \( p.\text{bound} \). Again we use the \texttt{Terminate\_with} as opposed to the \texttt{Confirm} in the rule to indicate total correctness.

The total correctness call rule is:

\[
\begin{align*}
\texttt{C\ code;} & \quad \texttt{Terminate\_with\ pre[x -> u]} \\
& \quad \land p.\text{Exp}[x -> u] < p.\text{bound}; \\
\texttt{C\ code;} & \quad \texttt{Terminate\_with\ \forall u',\ post[#x -> u, x -> u'] =>} \\
& \quad Q[u -> u'] \\
\end{align*}
\]

\[
\texttt{C\ code;} p(u); \texttt{Terminate\_with\ Q}
\]

The assertion about \( p.\text{bound} \) guarantees that when the actual parameter supplied by a particular call is substituted for the formal parameter, the resulting value is strictly less than the bound previously set. If this hypothesis fails, the assert status changes to categorically false, indicating that the program is incorrect. This has the effect of designating non-terminating programs and programs with improper bounding expressions as incorrect.

The following program illustrates our treatment of recursive procedures:

\[
\texttt{proc\ Factorial(const\ n: integer, \ var\ res: integer);} \\
\texttt{\ \ require\ n \geq 0;} \\
\texttt{\ \ ensure\ res = n!;} \\
\texttt{\ \ decreasing\ n;} \\
\texttt{\ \ if\ n = 0\ then\ res := 1} \\
\texttt{\ \ \ \ \ else\ Factorial(n - 1,\ temp);} \\
\texttt{\ \ \ \ \ res := n * temp;} \\
\texttt{\ \ \ \ \ endif;} \\
\texttt{\ end;}
\]
As we have already seen in our treatment of non-recursive procedures, the require clause is a programmer supplied assertion which tells under what conditions this procedure will guarantee that if it completes, the ensure clause holds. Together, these clauses form the specifications of the procedure. The heading of the procedure is the first line and the specifications:

```plaintext
proc Factorial(const n: integer, var res: integer);
  require n \geq 0;
  ensure res = n!;
  decreasing n;
```

Since, for any given block, the verifier first looks at all declarations and puts appropriate information into the context, i.e., saves information which will be needed to apply the rules, in this example, the verifier will have put the heading of Factorial into the correct context before applying the other rules necessary for proving correctness, hence making it possible for Factorial to call itself.

Since this entire program consists of a procedure, in order to verify it, we begin by applying the total correctness rule for procedure declarations. We get the following:

**Remember**: Assume \( n \geq 0 \land p.bound = n \); body;

- **Terminate_with** \( res = n! \);
- Assume \( p.bound = \omega \); code;

Since the body consists of an if then else, it is necessary to check both branches:

1. **Assume** \( n = 0 \land p.bound = n \); \( res := 1 \); **Terminate_with** \( res = n! \);
   Using the rule for assignment transforms this to
   
   **Assume** \( n = 0 \land p.bound = n \); **Terminate_with** \( 1 = n! \);
   Using the rules for Assume and **Terminate_with**, we get:
   
   \( n = 0 \land p.bound \Rightarrow 1 = n! \)

   This is true by a definition of number theory.

2. **Assume** \( n > 0 \land p.bound = n \); Factorial(\( n - 1 \), temp);
   **Terminate_with** \( n \times temp = n! \);
Now using the call rule (possible because the heading of the procedure, together with its name, are now in the context), first it is necessary to Terminate_with \( n - 1 \geq 0 \);
This follows because \( n > 0 \) is assumed.

We must also check that \( p.\text{Exp}[n \rightarrow n-1] < p.\text{bound} \).
But this is obvious since \( n - 1 < n \).

It remains to show:
\[
\forall ?\text{temp}, ?\text{temp} = (n - 1)! \Rightarrow n! = n \times ?\text{temp}.
\]
Since \( ?\text{temp} = (n - 1)! \), \( n \times ?\text{temp} = n \times (n - 1)! \)
But \( n \times (n - 1)! = n! \)

Seeing the simplicity of these rules for self recursion, we proceed to the general case, that of mutually recursive procedures. Although mutual recursion is not a commonly used tool in program development, we address the challenge of providing machinery for proving the correctness of such procedures for the sake of intellectual completeness. This general case for mutual recursion may involve any number of procedures which call each other and/or themselves.

When a collection of procedures \( p_1, p_2, \ldots, p_k \) interact in such a way as to create cycles of calls among those procedures, the situation is extensively more complex than that of a procedure calling itself. Indeed, because keeping track of all the paths which could occur in such a cycle of calls, leads us to a general graph problem, most proof systems have failed to address the problem of proof rules for general mutual recursion at all.

Our system allows us to generalize in a straight-forward way, enriching the declare and call rules for self recursion to create rules for mutual recursion. These rules provide termination checks and maintain the syntax-driven nature of the system (i.e., make automation possible).

One way to accomplish these goals is to add syntax for programmers to use to indicate that a collection of procedures belong together because they mutually recurse. For such mutually recursive collections of procedures, we introduce the key word
**Mutual_Recursion** to place at the beginning of such a collection and an end at the conclusion of their declarations.

Procedure headings are now more complex because in addition to supplying a progress metric for the procedures, we must also provide bounding expressions for the other procedures in the recursion. Intuitively, the progress expression for each procedure indicates the number of calls still to be made to that procedure, while the bounding expressions place limits on the numbers of calls out of the given procedure to other procedures in this mutually recursive collection.

We will write our rules based on the assumption that there are three mutually recursive procedures with no loss of generality for the cases having two or more mutually recursive procedures. Procedure headings now look like:

```
proc p(var x: T)
  require prep;
  ensure postp;
  decreasing p_Ex;
  bounding q by q_Ex;p;
  bounding r by r_Ex;p;
...
```

Each procedure in the mutually recursive collection will have a similar heading. Here q_Ex;p should be an upper bound on the value of q's progress metric for any direct or indirect call to procedure q from the body of p.

Our declaration rule is:

```
C \cup \{pr_heads\} \backslash Correct_Proc_Hyp
  Assume omega_bounds; code; Terminate_with Q;
```

```
C \backslash Mutual_Recursion mut_rec_proc end; code; Terminate_with Q;
```

where \(pr_heads\) is:

```
proc p(var x: T) require prep; ensure postp;
  decreasing p_Ex; bounding q_Ex;p; bounding r_Ex;p;
proc q(var x: T) require pre_q; ensure post_q;
  decreasing q_Ex; bounding p_Ex;q; bounding r_Ex;q;
proc r(var x: T) require pre_r; ensure post_r;
  decreasing r_Ex; bounding p_Ex;r; bounding q_Ex;r;
```
The context is enriched with all the headings from the mutually recursive procedures in this grouping.

The second abbreviation in the rule, Correct_Proc_Hyp, represents the following:

Remember
Assume p.bound = p.Exp \land pre_{p}; p_body;
Terminate_with post_{p};
Assume q.bound = q.Exp \land pre_{q}; q_body;
Terminate_with post_{q};
Assume r.bound = r.Exp \land pre_{r}; r_body;
Terminate_with post_{r};

The abbreviation omega_bounds stands for:
p.bound = q.bound = r.bound = \omega

The abbreviation mut_rec_proc on the bottom line of the rule stands for the entire block of code in which the headings and bodies of the given procedures are declared:

proc p(var x: T);
  require pre_{p};
  ensure post_{p};
  decreasing p.Exp;
  bounding q by qExp_{p};
  bounding r by rExp_{p};
  p_body;
end p;
proc q(var x: T);
  ...
end q;
proc r(var x: T);
  ...
end r;

Next we will formulate a call rule for mutually recursive procedures. The declaration rule has already checked the correctness of the procedure bodies and so we are left with the tasks of making sure that the precondition of the called procedure is met when the appropriate parameters are filled in and checking to see that progress toward completion of the recursive procedures is being made. The latter task is
accomplished by confirming that the progress expression for the called procedure has a value strictly less than its previously set bound and that none of the bounding expressions have increased in value. The call rule follows:

\[ \text{C code; } \text{Terminate\_with} \ pre_p \land p_{\text{Exp}}[x \rightarrow a] < p_{\text{bound}} \land q_{\text{Exp}}[x \rightarrow a] \leq q_{\text{bound}} \land r_{\text{Exp}}[x \rightarrow a] \leq r_{\text{bound}}; \]

\[ \text{C code; } \text{Terminate\_with} \ \forall a'(\text{post}_p[#x \rightarrow a, x \rightarrow a'] \Rightarrow Q[a \rightarrow a']; \]

\[ \text{C code; } p(a); \text{Terminate\_with} Q; \]

Before we can consider soundness and relative completeness of the above rules, we need to enrich our semantic space to support procedure declarations and calls. We retain the old state space \( S \times AS \) described previously but introduce a third component, \( PF \) describing the procedure functions. We rename the space \( S \times AS \times PF \) the environment \( Env \) to suggest its more complex nature. Structurally, \( PF \) maps each procedure name to a four-tuple of functions:

\[ PF \colon \text{Names} \rightarrow PE \times DP \times EP \times PM \]

1) PE: \( \text{Values} \rightarrow (\text{Values} \cup \{ \bot \}) \)
2) DP: \( \text{Values} \rightarrow \{ T,F \} \)
3) EP: \( \text{Values} \times \text{Values} \rightarrow \{ T,F \} \)
4) PM: \( \text{Values} \rightarrow \mathbb{N} \)

PE is the Procedure Effect Function and tells what the procedure does; DP is the Domain Predicate and records the precondition for the procedure; EP is the Effect Predicate and records the post condition; and PM is the Progress Metric and records the function defined by the progress expression.
Semantics for the call rule are given by:
\[ \text{Sem}(\text{code}; \ p(u))(E) = E'' \] where \( E : \text{Env} \),
\[ \text{Sem}(\text{code})(E) = E' \]

\[ \text{AS}'' = \text{CF} \text{ if } \text{AS}' = \text{N} \land \text{PF}'(p).\text{DP}(\text{eval}(u,E')) = \text{false} \lor \]
\[ \text{PF}'(p).\text{PM}(\text{eval}(u,E')) \geq \text{eval}(p.\text{bound},E') \]
\[ = \bot \text{ if } \text{AS}' = \text{N} \land \text{PF}'(p).\text{PE}(\text{eval}(u,E')) = \bot \]
\[ = \text{AS}' \text{ otherwise} \]
\[ \text{PF}'' = \text{PF}' \]
\[ S''(\tau) = S'(\tau) \text{ if } \tau \neq u \]
\[ \text{PF}'(p).\text{PE}(\text{eval}(u,E')) \text{ if } \tau = u \]

For the declaration rule, the values of all variables remain unchanged, the PF component of the environment is enriched by the addition of the newly declared procedure together with its heading, and the assert status becomes categorically false if and only if upon assuming the precondition given in the require clause and executing the procedure body, the post-condition given in the ensure clause fails or if the value of the progress expression fails to decrease or the values of the bounding expressions increase.

**Theorem 4** The declaration and call rules are sound.

Proof: To check for soundness of any rule, it is necessary to check that if the hypotheses of the rule do not allow the assert status to become CF, then the conclusion will not allow the assert status to become CF.

In the call rule, there are three ways that the final assert status may become CF, either \( \text{AS}' = \text{N} \) and \( \text{PF}'(p).\text{DP}(\text{eval}(u,E')) = \text{false} \) or \( \text{PF}'(p).\text{PM}(\text{eval}(u,E')) \geq \text{eval}(p.\text{bound},E') \) or \( \text{AS}'' = \bot \). But in the hypotheses, \text{Terminate}_\text{with} \ \text{exp} < p.\text{bound} and \text{Terminate}_\text{with} \ [x \rightarrow a] \) appear, eliminating these possibilities.

Similarly, in the declaration rule, the only way that the assert status may become CF is for the value of the progress expression to fail to decrease or for the ensure
clause to fail upon execution of the procedure body, even when the require clause was satisfied. The hypotheses of the declaration rule guarantee that the value of the progress expression decreases. Moreover, the hypotheses include assuming the precondition, and checking that the postcondition is met upon executing the body. Hence, both rules are sound.

Just as the existence of a proper progress expression was a vital component in proving relative completeness of our while rule, the existence of such an expression is obviously necessary for the proof of completeness of the termination rules for recursive procedures. To establish the existence of an expression whose value decreases with each iteration of a given while loop, we considered a string representing the histories of execution of the loop and observed that the length of that string was a decreasing positive natural number and hence provided a proper progress metric.

For recursive procedures we use the same strategy. We need an expression whose value decreases with each call, and so again we look at a string of states representing the history of execution of the given recursive procedure(s). However, because execution of recursive procedures is more complicated than that of iterative code, the description of this execution leads us to supplement the operational description given in the preceding section on loops. We handle self recursive and mutually recursive procedures the same way. We consider a collection of procedures p, q, r, ..., t in which any of the procedures may call any of the others or itself and in which cycles may occur, such as p calls q, q calls r, r calls t, and t calls p. For this discussion, we assume that within any procedure, p, each executable statement has been given a label.

In order to describe the computation of a collection of procedures, we will make assertions about Full_States, which include a stack of return states for all currently
active procedures. The concept of Local_State is the same as previously described.

\textbf{Full\_State} = \langle\text{Active\_State: Local\_State, Stack: String of Local\_State}\rangle.

When one procedure calls another, the label of the statement to which a return will be made, along with the values of local variables are pushed onto the stack. At return time, the stack is popped and execution continues.

It is now obvious that to provide technical details of descriptors for a language which supports recursive procedures, it is necessary to supplement the definition of CHF (Correct History Formula) by noting for each construct what happens to the stack. However, it is clear that for all constructs, except the procedure call, the stack introduced in the concept of Full\_State remains empty.

In the case of the call, the stack must retain the label of the statement to which control reverts upon completion of the procedure. With those modifications of what descriptors look like, it is now easy to see how one goes about establishing a lemma about valid programs and existence of descriptor strings for languages which include recursive procedures.

The termination results of this chapter, together with the module rules of Chapter 4 supply us with sufficient machinery to prove the correctness of the Nested List Realization.
CHAPTER VII
Nested Lists

VII.1. Motivation

The beginning chapters of this thesis have set forth a philosophy of programming which emphasizes precision in specification, efficiency in implementation, formalism in verification, and reusability in all of these activities. Along with these philosophical views, this document has introduced specific tools powerful enough to enable programmers to carry out the stated goals. Several short examples and a long one, the stack example, illustrating our language constructs and proof rules, were provided to show how these tools work.

In describing the primitive modules on which the stack example was built, we illustrated homogeneity of treatment within REFORMS. However, the stack is a well-known, relatively simple data type, which might be viewed as only one level above the primitive types supporting it. To lend credibility to our system as one which is capable of handling complex, real world structures, possibly built at hierarchical levels above user-defined types such as the stack, we would like to present here just such a structure, one which is built on top of user-defined, as well as primitive types, and one which is at the same time applicable in a natural, real-life setting.

With this goal in mind, this chapter introduces a new data structure, the Nested List Facility. Motivations for this particular structure can be found in a variety of
settings, including the popular class of computer programs known as editors. *Principles of Text Editing* by Tesky [55], for example, presents a survey of the various kinds of editors that exist and of the different sorts of features they provide, as well as other details about editors and formatters.

If one chooses any set of editing features and wants to write an editor to provide them, it is clear that the resulting editor should be correct and fast. The consistent achievement of these goals will only be possible when the programmer has at his disposal appropriate data structures in which to store the material to be edited — structures which allow quick access to various positions in the document and which permit powerful manipulations of the document components. Most frequently, editors have viewed the objects to be edited as strings of characters with markers at appropriate places to aid in locating particular positions. Other views have included linked lists or trees implemented as lists. Such conceptualizations force the programmer to think of a document as a single, frequently very long, string of characters. This conceptualization turns out not to be very conducive to easy or efficient implementation of the various editors.

The Nested List Facility described in this chapter suggests a new structure for storing objects to be edited — one whose conceptualization is flexible and permits a variety of conceptualizations, each of which is a natural way of describing a document. For example a document may be thought of as consisting of a table of contents, an introduction, a number of chapters, a conclusion, some appendices, and a bibliography, each of which may consist of additional subordinate parts. These subordinate parts, in turn, may be broken into subparts. For example, it may be the case that chapters are comprised of sections, sections are comprised of subsections, subsections are comprised of paragraphs, paragraphs are comprised of sentences, etc. By design, the Nested List will lend itself well to such applications.
The most important motivational point here is that this new structure is a natural response to an existing need, and this existing need issues both a challenge to and an opportunity for the REFORMS system. We will show how the tools this thesis has introduced facilitate the building of a new data structure which (1) is built hierarchically, (2) has practical uses, (3) can be implemented efficiently, and (4) is verifiable.

VII.2. Tree Theory

Building the Nested List Facility enables us to illustrate an additional feature of REFORMS: the provision for having new theories defined wherever appropriate. In the stack example and in all the primitive modules, the necessary mathematical theories for specifying the types exported by those modules already exist and are, in fact, mature parts of mathematics. In the substantial collection of data structures in Ogden’s work [48], for example, all but the structures dealing with trees are described in terms of well-known theories in mathematics, each with a standard notation. However, because the literature has not provided a standardized notation or a mature, commonly agreed upon theory of trees, it was necessary to develop a description for this theory in order to specify types based on it.

The theory we use for defining the Nested List Facility is one in which trees and strings of trees are defined by mutual induction. The formal description of this theory appears in Appendix C. Some examples will be given in a later section to illustrate the particular aspects of tree theory used in the Nested List Module. However, rest assured that the tree theory presented in Appendix C is not some exotic theory which confounds normal intuition about what trees are, but rather it shows a particularly simple way, we think a useful way, to formalize our intuitive notions about trees.
VII.3. Nested Lists

The fact that the new data structure we are about to introduce has been named Nested Lists and is specified in terms of tree theory will seem, I hope, both reasonable and natural. In conceptualizing a list which is composed of other lists also decomposable into subordinate lists, we turn naturally to trees. We picture each list with its component sublists as a tree, giving the overall view of a string of trees. Indeed, depending on an individual’s terminological preferences, one could name our structure either Nested Lists or Strings of Trees and still convey equally well the essence of the structure.

A great advantage in any conceptualization is a pictorial representation to aid our understanding of relationships among the parts of a complex object. Before writing the formal specifications (the conceptual module) for the Nested List Facility, we include some illustrations which provide us with a way to Precualize this new concept. The first example suggests a way that an implementor of a structured program editor might conceptualize the storing of program code using its BNF description.

Consider the code:

```
begin
    x := a + b;
end
```

Using italics for non-terminals and a standard font for terminal objects, we could represent the storage of this code as follows:
The large brackets indicate lists which include sublists [in this case the non-terminals] and the small brackets represent empty lists [in this case the terminals].
Structuring program text in this way will make it easy for the text editor to move expressions, statements, or blocks from one part of the text to another as whole units. The natural structure of the text will be maintained automatically, and if the Nested List is well implemented, the editing process will be more efficient than it is with a string based editor. The reason for developing a Nested List data structure should then be obvious.

In order to use this new structure, we see the need to refer to particular positions within the structure and the need to move from one position to another. For example, if one were to start at the beginning of the assignment statement with the intent of changing the `+' to a `*', he would need operations to advance horizontally and then to open a sublist of the list to get to the correct position. Moreover, it is obvious that some procedure to allow changing the labeling of a list is a necessity.

As soon as operations for moving around in the structure are available, other operations which check to see whether or not movement in a given direction can take place should be included. For example, it is important to know whether one is already at the end of a list before requesting to advance still further.

The types of operations provided for the nested list then fall into three categories, those which permit moving along in the list, those which permit checking to see if one is at a boundary of the list, and those which make changes to the list. First we will give a brief description of these operations and then some examples to illustrate them.

Included in the first category are Advance [move to the right at the same level of the list], Retreat [move to the left at the same level], Open_Sublist [go to the next deeper level], and Close_Sublist [move back up a level].

Three Boolean functions, At_Front, At_End, and At_Outside, check to see if one can go any further respectively to the left, to the right, and upward. At_End also checks to see if one can go further downward.
There are three operations which permit changes to be made to the list. The *Insert* procedure allows new entries to be made. This procedure is what makes it possible to create a new nested list as well as to insert new items into an existing one. *Swap_List* replaces the label on a list with another. There is also a command, *Swap_Rem*, which allows the user to interchange a designated part of one nested list with a similarly designated portion of another nested list.

The following figures illustrate the effects of the various operations on nested lists. The first figure shows a typical nested list position indicated by the arrow in front of the sublist labeled $f$. We will consider this the starting position for each of the operations illustrated in the subsequent figures.

---

Figure 2: Starting Position
The next figure illustrates what happens when we perform an *Advance* operation to the starting position. After the *Advance* is completed, we find the new position is at the right of the list labeled by $f$. If we called the *At_End* procedure here, we would get a response of *false*.

Figure 3: The Effect of Advance
Using Figure 2 as a starting position again, we next illustrate what happens when we perform an Open_Sublist operation, i.e. our current position moves down one level.

![Figure 4: Open_Sublist Example](image)

All of the figures so far have illustrated operations which permit movement within a nested list. The next one shows a modification of the starting position by showing what happens when we apply the Insert_List operation with $x$ as the label for the new list that is to be inserted.
Figure 5: Insert Example

These pictorial examples should be giving the reader a pretty good idea of how the nested list facility is going to operate. and that is why they were included. The code storage example in Figure 1 provides motivation for considering a new, two dimensional structure called nested list for use in editors and similar programs. The illustrations in Figures 2 through 5 have fostered an intuitive grasp of how to navigate in and operate on these nested lists.

In general, such pictorial illustrations are not only useful in helping people to become familiar with a new concept, but are, in fact, critical in providing a representation in terms of which people can reason about that concept. For example,
by using pictorial representations for points and lines in Euclidean geometry, one might be motivated to postulate that two lines which are perpendicular to a third line are parallel to each other. However, as any geometry student knows, the picture, although useful to strengthen his intuition, does not provide sufficient formality to enable one to establish that an intuitive idea is in fact correct. Similarly, the grade school experience of manipulating the decimal numerals we use to represent integers, might lead one to believe that multiplying two integers in either order will yield the same product. Yet, in order to prove the "commutativity of multiplication," one needs a formal axiomatization of number theory which is independent of decimal notation.

After considering these familiar theories of mathematics, we are not surprised to learn that we will need to select appropriate notation and axiomatization to formalize the ideas suggested by our drawings of nested lists. As earlier parts of this document have pointed out, formalization serves two important purposes. First, it provides the expressiveness needed to write precise specifications for our new structure; and secondly, it supplies the theory necessary for establishing correctness of programs which use nested lists.

To achieve the needed formalism, we have chosen the theory of trees and strings of trees. An axiomatization of this theory, together with definitions and theorems pertinent to our computer science applications appear in Appendix C. One particular property of this theory which makes it not only easy to apply, but also easy to understand, is its inductive nature, a property it shares with number theory.

The formal theory of labeled trees involves as a parameter, Γ, the desired set of labels, Tr(Γ), the set of trees with labels from Γ, and T_Str(Γ), the set of strings of trees. There are two primitive elements in the theory, Ω, the empty tree, and Λ, the empty string of trees. The basic functions of the theory are Join, the operation for
joining a string of trees together to form a new tree, and $\text{Ext}$, the operation for extending a string of trees by appending an additional tree.

The following example illustrates the intended effect of the $\text{Join}$ operation in our pictorial representation of trees:

In formal, type theoretic terms, $\text{Join}: \Gamma \times \text{T}_\text{Str}(\Gamma) \rightarrow \text{Tr}(\Gamma)$.

Similarly, we can illustrate pictorially the intended effect of the $\text{Ext}$ operation:
In type theoretic terms, \( \text{Ext}: T\_Str(\Gamma) \times T(\Gamma) \to T\_Str(\Gamma) \).

Just as in number theory where one can construct any given natural number by starting with 0 and applying the successor function an appropriate number of times, so in tree theory, one can construct any tree or string of trees by applying the extension and join functions appropriately to the \( \Lambda \) and \( \Omega \) elements.

In order to use the concepts of tree theory to describe our nested list structures and the effects of various operations we perform on them, we choose the same
notational conventions as those which appear in the axiomatization in Appendix C. Letting small Greek letters represent strings of trees and Roman letters stand for elements of the given alphabet $\Gamma$, we can express concisely and accurately the ideas we need.

For example, the first axiom of tree theory is written:

(a) $\forall \alpha: T_{\text{Str}}(\Gamma), x: \Gamma, \text{Join}(x, \alpha) \neq \Omega$.
(b) $\forall \alpha: T_{\text{Str}}(\Gamma), T: T_{\text{rf}}(\Gamma), \text{Ext}(\alpha, T) \neq \Lambda$.

This axiom illustrates not only the notation, but also the mutually inductive relationship between trees and strings of trees. We will use this notation and the theory of trees in describing nested lists formally, starting with the concepts necessary to talk formally about the notion of a position within a nested list.

In our pictures we indicated a current position in a nested list by means of an arrow pointing to that position. However, drawing pictures to illustrate relationships among the parts of some given structure may be a natural way to aid our reasoning about the structure, but it does not provide a basis for formal proofs about operations on that structure.

Just as the concepts of Euclidean geometry provide the formalisms for establishing relationships among points and lines, there are definitions in tree theory which supply the mathematically rigorous basis for describing and reasoning about nested list positions.

In an effort to keep our theory as simple as possible, we might try to describe a particular position in a nested list by use of its label. However, as Figure 1 easily illustrates, there may many positions all having the same label. This makes it clear that we will need more than a label to describe a position. We will need information which pinpoints a particular position relative to the rest of the nested list.
One way to provide this information is to record the path from the root list which leads to the given position. Such a path description should contain a record of the various sites which must be passed through in order to reach the current position. Just as a single label fails to uniquely identify a particular position, the string of labels in front of the sites on the path to the position will not provide sufficient information to uniquely identify it, since it is possible, for example, that all the positions in the nested list have the same label.

It is clear then that we must include more information than just a label or even a string of labels to identify our place in a nested list. At the same time, it is important for ease of understanding that we avoid "over-specification," i.e., redundancy in our description. Extensive consideration of a variety of ways to achieve a description of this unique location without redundancy has led us to the conclusion that the simplest solution involves using a complete description of the path that leads to the desired location. This path then consists of a sequence of complete descriptions of the sites that must be visited in order to get from the root list to the desired location. A site description indicates in which subtree the desired position is contained by recording the label of that subtree, all the trees that precede it, and all the trees that follow it.

If, for example, we were attempting to describe the position labeled \( j \) in Figure 2, the first site \( SI \) in the path to this site would be:
Figure 6: Site S1
The second site $S_2$ in the path to $j$ would be:

label $f$

left string

right string

![Diagram of Site S2]

Figure 7: Site $S_2$

The path to the appropriate sublist would then be $p = \langle S_1, S_2 \rangle$. To fully describe our location to be the tree labeled with $j$ within the sublist, we need two additional pieces of information.

The path shows what sites it was necessary to Precit in levels above the current level in order to arrive at the current position. To complete the identification of this unique position, we will use two strings which we call $Prec$ and $Rem$ to suggest what
has already been Precited within the current level and what remains Remited. For the position associated with the label $j$ we have the following associated strings:

$j.$ Prec

\[
\langle \ i \Delta \ j \rangle
\]

$j.$ Rem

\[
\langle \ j \rangle
\]

\[
\langle \ l \Lambda \ m \Omega \ n \rangle
\]

Figure 8: $j.$ Prec, $j.$ Rem
To understand how we use the Path as a way of factoring our list into disjoint parts which can be put back together, we will examine the third concept which is an operator called Suture, symbolized as \( \Xi \). This operator is defined in terms of the Join operator already described. \( \Xi \) is a function from the pair, Strings of Sites, Strings of Trees to Tree Strings and is defined inductively:

\[
\Xi : \text{Str}(\text{Site}) \times \text{Tr}_\text{Str} \rightarrow \text{Tr}_\text{Str}
\]

\( \forall S : \text{Site}, \rho : \text{Str}(\text{Site}), \alpha, \beta : \text{Tr}_\text{Str}, \)

(i) \( \forall \Xi \alpha = \alpha \)

(ii) \( \rho \circ S \Xi \alpha = \rho \Xi S.\text{LS} \circ \text{Join}(S.\text{Lbl}, \alpha) \circ S.\text{RS} \)

Using our example and applying the suture operator to the j_path and the exhibited pair of singleton tree strings, j.Prec and j.Rem, we must perform two steps, one step for each site in the given path. The first step joins the label of the second site in the path to the string formed by concatenating the pair \( j.\text{Prec} \) and \( j.\text{Rem} \). The final result of applying the suture operator is recovery of the entire nested list.
Step 1:

```
  e \Delta f
   \|
  i \Delta j
   \|
  l \Delta m
   \|
  n \Delta
```

Step 2: The entire list as it appears in Figure 2

Figure 9: Suture Example
Using the above examples to guide our intuition, we will define a new type, Nstd_Lst_Psn, to stand for nested list position, and explore a collection of operations which manipulate variables of the newly defined type. One reason for preceding the Nested List Module declarations by the motivation discussion and an example is the obvious pedagogical tactic of familiarizing the reader with a new idea before formalizing it. There is, however, a less obvious, yet nevertheless more compelling reason for the order of presentation, the fact that this is the same order in which REFORMS approaches real world problems.

First one notices some need which arises in the natural course of solving a programming problem. In the first part of this chapter, the need for a general storage structure to support editors was pointed out. Next one looks for possible solutions among existing software constructs. In the case of editors, one looks at existing storage structures such as linked lists and arrays as potential solutions. If available software does not achieve the identified goals, such as providing a natural and obvious match of the structure with the abstraction, then one thinks about alternatives, usually starting with informal examples.

Finally, after working out several examples, usually by means of some pictorial representation, including multiple representations and possibly some false starts, a programmer can articulate informally his new conceptualization upon which to build a solution to the original problem. At that point, the programmer is ready to formalize the new idea in the form of a conceptual module.

Before presenting the conceptual module for Nested Lists, we will consider a few preliminary remarks so that reading the module will be easy. The examples given so far have illustrated the concepts of path, site, position in a nested list, and the join and suture operators, all of which appear in the conceptual module.
One additional bit of tree notation is the predicate NFT_Str which stands for Null Free Tree Strings, and means strings of trees which do not include empty trees. There are both practical and technical reasons for excluding empty trees from our strings of trees. From the practical standpoint, there is no advantage to inserting an empty tree into a list. The only reason one would want to insert a tree is that the tree contains some nodes. Technically, it makes the specifications more complicated, simply because there are more possibilities to cover. Note that there can be empty strings, that is, strings having no trees at all. However, there cannot be any tree string in which an empty tree is one of the string components.

Also from the examples, it is clear that a nested list is a labeled list, each of whose members is a labeled list. The new module described in this chapter is called Nested_List_Facility, and it exports a type called Nstd_Lst_Psn to suggest nested list position. As we have seen, the nested list position is a cartesian product consisting of three parts: a path and two strings. The module puts constraints on variables of the type Nstd_Lst_Psn to restrict the total number of nodes. More will be said about this when the parameter list is explained. Initial conditions on the type Nstd_Lst_Psn require that each list be initialized so that each component in the cartesian product is empty.

On the parameter list of the conceptual module is max_cnt, an integer which sets the upper bound on how many nodes may appear in the collection of nested lists. Note that this maximum number will be divided up among all nested lists. The module requirement states that max_cnt > 0, thereby excluding the possibility of a useless module which exports a type of object which is always empty.

Also on the parameter list is a type Item. The type parameter indicates what type the variables in the nested list must belong to. Since the parameter is not bound until an instance is declared, the same conceptual module may be used for all nested lists,
regardless of the internal values stored in it. This is what we expect according to the work presented in Chapter 3.

Having seen several examples of this new kind of structure, together with illustrations of some operations associated with it, we are ready for a formal specification of the Nested List Facility. As usual, in keeping with the REFORMS philosophy, the specification of the nested list conceptual module provides for the programmer not only language which permits formal verification, but a convenient notation in terms of which the programmer can reason about the objects and operations associated with nested lists.

VII.3.1. Conceptual Module

Conceptual Module Nested_Lst_Facility(val max_cnt: Integer,
type: Item);

Require max_cnt > 0;
Uses Tree Theory including type Site; Definition Suture;

Type Nstd_Lst_Psn = Cart_Prod
    Path: Str(Site);
    Prec, Rem: T_Str
end;

exemplar P;
definition L.Pos(i: N) = Nstd_Lst_Psn.Denotation(i);
definition List_Node_Ct(i: N) = N_Cts(L.Pos(i).Path \in L.Pos(i).Prec
    o L.Pos(i).Rem);

definition Aggregate_Length = \sum_{i=1}^{Nstd_Lst_Psn} List_Node_Ct(i);  

definition Remaining_Capacity = max_cnt - Aggregate_Length;

consults Remaining_Capacity \geq 0;

initially P.Path = \Lambda \land
    P.Prec = \Lambda \land P.Rem = \Lambda;

Proc Open_Sublist(var P: Nstd_Lst_Psn)
    Require P.Rem \neq \Lambda;
Ensure \( \exists S: \text{Site such that } P.\text{Path} = P.\text{Path} \circ S \land P.\text{Prec} = \Lambda \land \)
\( S \subseteq P.\text{Rem} = P.\text{Prec} \circ P.\text{Rem} \land \)
\( S.\text{Lft}_\text{Siblings} = P.\text{Prec}; \)

Proc Close Sublist(var P: Nstd_Lst_Psn)
Require P.\text{Path} \neq \Lambda;
Ensure S: Site such that #P.\text{Path} = P.\text{Path} \circ S \land 
S \subseteq P.\text{Prec} \circ P.\text{Rem} = P.\text{Prec} \circ P.\text{Rem} \land 
P.\text{Prec} = S.\text{Lft}_\text{Siblings};

Proc Advance(var P: Nstd_Lst_Psn)
Require P.\text{Rem} \neq \Lambda;
Ensure P.\text{Prec} \circ P.\text{Rem} = #P.\text{Prec} \circ #P.\text{Rem} \land 
|P.\text{Rem}| = |#P.\text{Rem}| - 1;

Proc Retreat(var P: Nstd_Lst_Psn)
Require P.\text{Prec} \neq \Lambda;
Ensure P.\text{Prec} \circ P.\text{Rem} = #P.\text{Prec} \circ #P.\text{Rem} \land 
|P.\text{Prec}| = |#P.\text{Prec}| - 1;

Proc At End(const P: Nstd_Lst_Psn): boolean;
Ensure At End = (P.\text{Rem} = \Lambda);

Proc At Front(const P: Nstd_Lst_Psn): boolean;
Ensure At Front = (P.\text{Prec} = \Lambda);

Proc At Outside(const P: Nstd_Lst_Psn): boolean;
Ensure At Outside = (P.\text{Path} = \Lambda);

Proc Reset(var P: Nstd_Lst_Psn)
Ensure P.\text{Path} = \Lambda \land P.\text{Prec} = \Lambda \land 
P.\text{Rem} = #P.\text{Path} \circ #P.\text{Prec} \circ #P.\text{Rem};

Proc Swap Rem(var P, Q: Nstd_Lst_Psn)
Ensure P.\text{Path} = #P.\text{Path} \land Q.\text{Path} = #Q.\text{Path} \land 
P.\text{Prec} = #P.\text{Prec} \land Q.\text{Prec} = #Q.\text{Prec} \land 
P.\text{Rem} = #Q.\text{Rem} \land Q.\text{Rem} = #P.\text{Rem};

Proc Swap Label(var P: Nstd_Lst_Psn, var labl: Item);
Require P.\text{Rem} \neq \Lambda;
Ensure \( \forall \rho, \sigma : \text{Tr}\_\text{Str}, \text{clb}: \text{Item}, \text{if } \)
#P.\text{Rem} = \text{Join}\_\text{clb}(\rho) \circ \sigma \text{ then } 
P.\text{Rem} = \text{Join}\_\text{labl}(\rho) \circ \sigma \land \text{labl} = \text{clb} \land 
P.\text{Path} = #P.\text{Path} \land \text{Prec} = #P.\text{Prec};

Proc Insert List(var P: Nstd_Lst_Psn, var list lab: Item;
Require Remaining Capacity > 0;
Ensure $P.\text{Path} = #P.\text{Path} \land$
$P.\text{Prec} = #P.\text{Prec} \land$
$P.\text{Rem} = \text{Join}(\text{List}_\text{lab}, \Lambda \circ #P.\text{Rem};$

\textbf{Proc} \text{Delete}_\text{List}(\text{var} P: \text{Nstd}_\text{Lst}_\text{Psn}, \text{var} \text{lab}: \text{Item});$
\quad \text{Require} \exists a: \text{Item}, \alpha: \text{Tr}_\text{Str} \text{such that}$
\quad P.\text{Rem} = \text{Join}(a, \Lambda) \circ \alpha;$
\quad \text{Ensure} P.\text{Path} = #P.\text{Path} \land$
\quad P.\text{Prec} = #P.\text{Prec} \land #P.\text{Rem} = \text{Join}(\text{lab}, \Lambda) \circ P.\text{Rem};$

\textbf{Proc} \text{Clear}(\text{var} P: \text{Nstd}_\text{Lst}_\text{Psn});$
\quad \text{Ensure} P.\text{Path} = \Lambda \land P.\text{Prec} = \Lambda \land P.\text{Rem} = \Lambda;$
end Nested_List_Facility;

\textbf{VII.3.2. A Realization Module}

While the objectives of the conceptual module are to provide a language for formal verification and a representation for the programmer to use in thinking about the program variables, the primary goal of the realization module is efficiency. With time and space considerations in the forefront, the programmer tries to achieve the required operations in as little time as possible. To link this world of clever short cuts to the formal specification world, the programmer includes in the realization a correspondence showing how to recover a conceptual variable from any realization variable. In the case of the stack example in Chapter 3, the correspondence was expressed in terms of a simple concatenation of elements in an array. Before giving the correspondence for the nested list module, it is necessary to present the realization details.

In order to achieve an efficient realization for the nested list module, I have chosen to take advantage of the previously defined and verified stack module, as well as arrays and integers. At the realization level, the nested list position is represented by four components: $\text{Crnt}$, an index into three arrays, $\text{Lbl}$, $\text{First}_\text{Child}$, and $\text{Nxt}_\text{Sb}$,
Root, the index of the beginning of the list, Trace, a stack on which stacks of integers contain indices of portions already visited, and Loc_Trace, a stack of integers which are indices of positions already visited at the current level. Array Lbl keeps the items of the nested list. First_Child has the index of the first child of the current position. Nxt_Sb has the index of the next sibling to the right for the current position.

Another way the realization module achieves efficiency is by taking over the job of space management for all nested list positions. The stack F_S keeps unused indices into the three arrays, First_Child, Nxt_Sb, and Lbl so that each time a variable of type nested list position is declared, the initialization routine simply finds an available index into the arrays by popping the stack F_S and gives that index to the new list position as its current pointer. This makes it possible for all variables of type Nstd_Lst_Psn to share the same arrays.

This combination of structures permits the module to keep track of every necessary piece of data for carrying out the required procedures without redundancy. Each time a new portion of the list is Precited, the index of that location is placed on the Loc_Trace stack, thus making it possible to move backwards, if desired, by popping the stack to get the most recently Precited address.

The array, Lst_Hldr, is initialized so that many nested list positions can share the same storage, while each nested list position has its own Trace and Loc_Trace stacks, together with the Crnt and Root pointers.

The two modules together give both ease of logical manipulation and efficiency in implementation. Note that each realization procedure can be carried out in constant time. There are no loops or procedure calls except calls to an adjunct procedure which does not need to be executed. An explanation of this adjunct procedure will be given in the next section.
Having discussed how the realization works, we now turn to the job of relating the implementation to the conceptualization. Because there is such a wide separation between the conceptualization and this realization, it will be no surprise that we will introduce an adjunct variable in order to express the correspondence easily.

In particular we will use an adjunct array $Sbling_Hldr$ in which we store the entire remainder portions of nested lists for each label. Having all this information available makes it easy to describe each conceptual position because the adjunct entries are given in terms of the realization elements and then the conceptualization in terms of the adjunct.

The correspondence is a formal way to say that (1) The Remainder portion of any nested list position can be found in the array, $Sbling_Hldr$. (2) The $Loc_{Trace}$ has the indices of the Precited part of the nested list position. (3) By using the stack, $Trace$, one can recover the path of sites which are part of the nested list position and which permit one to know everything that belongs in the given nested list.

Upon examination of the realization module, one observes that there is a procedure, $Adjust$, in the realization which is not specified in the conceptual module. The specifications for $Adjust$ are written within the procedure as the require and ensure clauses. The purpose of this procedure is to keep the adjunct array up to date. Since the realization module introduced the adjunct variable as a convenience for verification, no conceptual notion of that variable needs to exist in the conceptual module for nested lists. However, in order for this adjunct variable to fulfill its designated purpose of aiding in the correspondence expression, the adjunct variable must reflect any changes which are made to the realization variables, hence maintaining the required correspondence.

The realization module conventions include the requirement that the $Sbling_Hldr$ array must maintain the correct correspondence upon execution of any procedure.
Some procedures do not change anything in the nested list, but rather only indicate that a move has been made (Advance, Retreat, Open, Close). Some, the Booleans, check a condition (At_End, At_Root, At_Front). However, the remaining procedures all change what is stored in a given nested list (Insert, Delete, Swap_List, Swap_Rem). These procedures which change the list, actually producing a new list, are unable to maintain the module conventions unless they make appropriate changes to the adjunct variable, Sbling_Hldr. Since all of these procedures need to do the same thing to Sbling_Hldr, one procedure can do the job. The name of this procedure is Adjust.

Adjust receives as parameters a nested list position, say $X$, as a constant. Copies of $X.Trace$ and $X.Loc_Trace$ are made as $Temp_Trace$ and $Temp_Loc_T$, respectively. The trace stack holds the stacks of integers which are indices into the Lst_Hldr array and represent which portions of the list must be updated if a change is made to the current position. All portions of the array whose indices are not in any of the stacks in the trace have not been Precited and hence need no updating.

The procedure Adjust pops off each stack of integers, one at a time from Temp_Trace, then one by one pops the integers from that stack of integers and assigns to the part of Sbling_Hldr with that index what should be there to keep the correspondence correct and hence maintain that part of the module conventions.

When Temp_Trace and all the stacks in it are empty, every portion of Sbling_Hldr is up to date. This is easy to see because the require clause states that all portions of the list whose indices are not in the stacks of Temp_Trace must already satisfy the module conventions. Hence, since each integer of every stack on Temp_Trace is removed and used as an index into Sbling_Hldr so as to set that part of Sbling_Hldr to the value which causes the correspondence to be correct and hence contribute toward preserving the module conventions.
It is important to note that the proof for Adjust, an adjunct procedure, is done exactly the same way as the proof for any real procedure. The fact that this procedure deals with adjunct variables does not require any unusual treatment. However, the theoretic implications of having adjunct variables in the program do require careful handling, as we have seen.

VII.3.3. Realization Module

This section gives part of one possible implementation for Nested Lists. This particular implementation takes advantage of our primitive modules, as well as a previously defined stack module. The part of the realization presented here declares instances of records, arrays, and stacks which are necessary to carry out the plan laid out in the pages immediately preceding this realization module.

Realization_Module Stack_of_Stack for Nstd_Lst_Facility

Module I_Stack is Stack_Facility(integer, max_cnt)
  Realized_by Array_with_Top_Pointer;
Module S_Stack is Stack_Facility(I_Stack.Stack, max_cnt)
  Realized_by Array_with_Top_Pointer;
Module L_Record is
  Record_Facility(integer, I_Stack.Stack, S_Stack.Stack)
  renaming P1 as Cnt,
    P2 as root,
    P3 as Trace,
    P4 as Loc_Trace,
    Swap1 as Cnt_Swap,
    Swap2 as root_Swap,
    Swap3 as Trace_Swap,
    Swap4 as Loc_Swap;
  Realized_by standard;

Module I_Array is Array_Facility(integer, max_cnt)
  Realized_by standard;
Module L_Array is Array_Facility(Item, max_cnt)
  Realized_by standard;
Adj_Module T_Array is Array_Facility(T_Str,max_cnt);

var nxt_free, root: integer;

var Lbl: L_Array.Array;
var First_Chld, Nxt_Sb: I_Array.Array;
Adj_var Sbling_Hldr: T_Array.Array;
Set_Bounds(Lbl, 1, max);
Set_Bounds(First_Chld, 1, max_cnt);
Set_Bounds(Nxt_Sb, 1, max_cnt);
Set_Bounds(Sbling_Hldr, 1, max_cnt);

Type Nstd_Lst_Psn = L_Record.Record;

This is enough to get an idea of what is involved in the realization, showing what is needed to set up the realization type. The remainder of the code will appear in the next chapter, along with proofs of correctness.

However, before looking at that work, in order to complete the discussion of the Nested List Facility as a useful module, we will examine some possible secondary operators for it. The enhancement given here is called Edit, suggesting that the operators are useful in a setting where editing is being done.

The first procedure, Append, allows one to append one nested list to another, obviously useful in any editing situation. The next operator allows one to delete the Remited portion of a list. The next allows the exchange of two lists, in essence renaming both.
VII.4. An Enhancement for the Nested List Facility

**Enhancement Edit of Nested List Module**

```plaintext
proc Append(L, M: Nstd_Lst_Psn)  
  require L.Path, L.Prec, M.Path, M.Prec = A;  
  ensure L.Rem = #L.Rem o #M.Rem;  
  var temp: LA_Record.Record;  
  While not At_End(L) do  
    Advance(L);  
  end;  
  with L  
    Swap(Lst_Hldr(cmt), temp);  
  end;  
  with M  
    Nxt_Sb := cmt;  
  end;  
  with L  
    Swap(temp, Lst_Hldr(cmt);  
  end;  
  While Not At_Front(L) do  
    Retreat(L);  
  end;  
end;
```

```plaintext
proc Delete_Rem(L: Nstd_Lst_Psn)  
  require  
    ensure L.Path = #L.Path ∧ L.Prec = #L.Prec ∨  
    L.Rem = A;  
  var M: Nstd_Lst_Psn;  
  Swap_Rem(L,M);  
end;
```

```plaintext
proc Exchange(L,M: Nstd_Lst_Psn)  
  require ∃t,u: NFT ∧ α, β: T_Str  
  such that #L.Rem = t o α ∧ #M.Rem = u o β;  
  ensure L.Rem = u o α ∧ M.Rem = t o β;  
  Swap_Rem(L,M);  
  Advance(L);  
  Advance(M);  
  Swap_Rem(L,M);  
end;
```
CHAPTER VIII
A Total Correctness Proof

The purpose of this chapter is to show that our proof system provides not only a mathematically rigorous approach to program verification, but a practical one as well. We achieve mathematical rigor through the use of formal specifications together with proof rules which are both sound and complete. In order to show that our system is also practical, we will apply our proof rules to a complex, hierarchically constructed module, thereby illustrating the fact that even though a module's implementation may be complicated, its correctness proof usually is not.

To accomplish this goal of establishing the practical nature of our proof system, we have chosen to prove that the realization of our nested list facility meets its specifications. This realization is complicated in two ways. In the first place, it involves an adjunct variable and therefore requires a total correctness proof. Secondly, this implementation relies on hierarchically built, user-defined types, namely stacks of stacks of integers. What we will see is that, despite these complications, our proof techniques remain unchanged and the mathematical assertions resulting from the application of our rules are quite simple.

But before this validation takes place, we are likely to benefit by considering briefly what has led us to this point. Chapter 3 introduced a new approach to writing modules with the purpose of achieving a large degree of reusability. Both conceptual and realization modules for stacks were written to illustrate this new approach.

183
Chapter 4 provided proof rules for module declarations and applied the realization declaration rule to the stack example. Chapter 5 specified and gave a realization for a new data structure called nested lists and pointed out that verification of the given realization required total correctness rules. Chapter 6 supplied the necessary rules so that now the tools we need to formally verify the nested list realization are available.

To begin our validation, we take advantage of the fact that our proof techniques need not change, that is the treatment of the nested list module is no different from that of either the primitive modules such as character and integer or the stack module which has already been presented. The same module rules apply. For this reason we will emphasize the part of the proof which illustrates what is different in this module as compared to the stack example, namely the use of an adjunct variable.

VIII.1. How the Adjunct Variable is Used

In Chapter 5, the implementation of the Communal Stack Module used an adjunct variable and so we know that the reason for introducing an adjunct variable is to simplify the expression of the correspondence between the realization and the conceptualization of a nested list position. The conceptual view of a nested list position allows one to know what has already been visited in the list in the context of the entire list. For efficiency, the realization includes only some indices into an array and two stacks. So, to express a relationship showing how to recover the conceptual view from the realization is complicated. The adjunct variable acts as an intermediate step in expressing this relationship. The adjunct array holds the unvisited portion of the nested list for any given labeled position.

With the use of the adjunct variable, expressing the correspondence can be done in terms of the adjunct variable, but the problem of keeping this adjunct array up to
date arises. When an insert or delete is performed, the adjunct array must reflect this change, and although the adjunct code need not be actually executed, it is necessary to check that this updating can be completed. I have chosen to do this by exhibiting an adjunct procedure, which happens to be in the form of a while loop to this updating. Hence, the termination rules are necessary.

To verify any of the procedures which include a call to an adjunct procedure, "Adjust" which guarantees that the adjunct variable satisfies the module conventions, it is necessary that the adjunct procedure be known to be totally correct.

Even though this procedure consists of pure adjunct code and hence need not be executed, it is critical that the procedure theoretically terminates so that the calling procedure can assum its post condition in order to prove the module conventions. I have chosen to handle this termination problem by applying the total correctness rules to the adjunct procedure.

Notice that the loop invariant for the while loop in this adjunct procedure states that the adjunct array is up to date for all indices except for those in the stack, Trace. Since, at the end of the procedure, the stack is empty, the entire array is required to be up to date. This is exactly one of the assertions in the module conventions and hence, the real procedure which called it will know that that module convention is true.

VIII.2. Shared Storage

The Nested List Module uses shared storage in the a way similar to that of the Communal Stack Realization of Chapter 5. However, because nested list positions are more complicated than stacks, it will be helpful to examine the details of the global variables involved in this sharing of storage among variables of type Nstd_Lst_Psn. In the Nested List Realization there are global (to the module) variables which all
variables of type Nstd_Lst_Psn share, namely the three arrays, Lbl, First_Chld, and Nxt_Sb. Each nested list position has two indices into these arrays, root which points to the outermost layer of the given list and crnt which points to the position ready to visit currently. The variable nxt_free is an index into the three arrays, available for use by any newly declared Nstd_Lst_Psn.

Another concern which arises as a result of using shared storage is that of returning storage which is no longer needed to the available pool of space. Since all the instances of nested list position have storage in the shared arrays, because the storage never overlaps (the conventions guarantee this), when an instance is no longer needed, its indices into the shared arrays can be returned to the free storage. To accommodate this need we put in a finalization procedure which pushes the root of the instance to be finalized onto the finalization stack, F_S.

Storage is shared in the following way:

• The Module Initialization procedure links together the First_Chld array and then pushes the index 1 onto F_S, the free storage stack.
• Whenever an insert is made into any nested list position instance, F_S is popped to get an available index.
• If the index taken from F_S has children or siblings, then those indices are returned to F_S for later use
• When an instance is no longer needed, a call is made to the finalization procedure which pushes onto F_S the root of the nested list position being returned. Hence the finalization is accomplished in constant time.
VIII.3. Notation

In order to facilitate both the presentation and the understanding of the proof that the Nested List Realization meets its specifications, I will point out what hypotheses must be checked according to the module rules and will make some appropriate abbreviations, noting that several formulas show up repeatedly and can hence be combined into some abbreviated form.

To begin let's match up the specific parts of the Nested List Realization with the module rule:

The Realization Module Heading together with the Realization Module Declarations:

Realization_Module for Nested_List_Facility;
Module I_Stack is Stack_Facility(integer)
  Realized by Array_with_Top_Pointer;
Module S_Stack is Stack_Facility(stack)
  Realized by Array_with_Top_Pointer;
Module L_Record is
  Record_Facility(integer,I_Stack.Stack,S_Stack.Stack)
  renaming P1 as Cnt,
  P2 as root,
  P3 as Trace,
  P4 as Loc_Trace,
  Swap1 as Cnt_Swap,
  Swap2 as root_Swap,
  Swap3 as Trace_Swap,
  Swap4 as Loc_Swap;
  Realized by standard;

Module I_Array is Array_Facility(integer)
  Realized by standard;
Module L_Array is Array_Facility(Item)
  Realized by standard;
Conceptual Module Tree_String_Facility(Type Alphabet,
    const size: integer);

Uses *Tree Theory*;
Type Tre = Tree; exemplar t
  Constraints ∀ t: Tre
    \(N_Cts(t) \leq size\);
initially \(t = \emptyset\);
Type T_Str = String of Tree; exemplar s
  Constraints ∀ s: T_Str,
    \(M_Ct(s) \leq size\);
Initially \(s = \emptyset\);
Proc Join(const a: Alphabet, const x: T_Str,
    var y: Tre)
  Require \(N_Cts(Join(a,x)) \leq size\);
  Ensure \(y = Join(a,x)\);
end;
Module T_Array is Array_Facility(T_Str);
var nxt_free, cap: Integer;
var Lbl: L_Array.Array;
var First_Chld, Nxt_Sb: I_Array.Array;
Adj_var Sbling_Hldr: T_Array.Array;

The only module parameter is the constant max, an integer which tells how large
the number of items may be. This is the total for all instances of Nstd_Lst_Psn as the
assertions state.

As was pointed out in the chapter which introduced Nested Lists, the
implementation strategy includes sharing storage for all variables of type Nested List
spread over three arrays, \textit{Lbl}, in which the items are stored, \textit{First_Chld}, in which
indices of first children are stored, and \textit{Nxt_Sb}, in which indices of the next sibling are
kept. For any given Nested List, \(P\), one index, \(P.crnt\) serves to find one's place in all
three arrays.

Several module instances are needed for the realization. The \textit{L_Record} module
exports a type which keeps sufficient information for finding one's place in a given
nested list. \textit{L.Root} is the index of the starting place of a given Nstd_Lst_Psn. \textit{L. L.Crnt} is the index of the current position. \textit{L.Trace} is a stack of stacks of integers, as
explained before. *L.Loc_Trace* is a stack of integers. As advances or retreats are performed, *L.Loc_Trace* is pushed or popped respectively.

The *I_Array* exports the type which is an array of integers. Both the *Nxt_Sb* and *First_Child* arrays must be of this type. The entries in these arrays serve as indices to the next sibling and the first child respectively of a given node. Another way to think of these integer entries is as pointers to the next value within the same level and to the first value at the next level of the given nested list.

*Lbl* is of type array of items to permit storage of the particular values which belong at the nodes. These values are of type "item" which is a parameter to the module. *I_Stack* and *S_Stack* provide stacks of integers and stacks of stacks of integers respectively needed for the Loc_Trace and Trace whose use is described previously.

The Conceptual Module *Tree_String_Facility* is declared because it is needed for the adjunct variable *Sibling_Hldr* described in chapter 3. Using the module is the adjunct instance *I_Array* which exports the type, array of tree strings.

The remaining declarations are obvious from the above explanations. The code following the last variable declaration is initialization code for the arrays. The next part of the code is the body of the realization. I have chosen to put all but the individual procedures next and will show the code for the procedures at the place where the verification is presented.

**Definition** $\forall T: \text{Tre}, a: \text{Item}, \rho: T_{\text{Str}},$

if $T = \text{Join}(a, \rho)$, then any suffix of $\rho$ is called a Sibling String;

**Definition** $\forall \rho, \sigma, \tau: \text{Str}(N), i: N$

if $\rho = \sigma \circ \tau$, then $\rho$ is a Sibling Integer String

iff *Sibling_Hldr*(i) is a Sibling String.

**Definition** $\forall S: \text{Sibling Integer String}, i,j: N,$

$\rho, \sigma: \text{Str}(N)$, if $S = j \circ \sigma = \rho \circ i$,

if $S'$ is the site such that $S'.LBS = LBSl(i)$. 

S'.Rt_Sblings = Sling_Hldr(i) and
S'.Lft_Sblings o S'.Rt_Sblings = Sbling_Hldr(j), then
S' is called the corresponding site of S.

Lemma F: Sibling Integer Strings -> Sites
defined by F(S) = S' is a function.

M_Conditions
∀ i,j,k: integer if 1 ≤ i,j,k ≤ max_cnt,
then ∀ First_Chld(i), First_Chld(j) if 1 ≤ First_Chld(i),
First_Chld(j) ≤ max_cnt,
then First_Chld(i) = First_Chld(j) iff i = j ∧
Nxt_Sb(k) ≠ First_Chld(i) ∧
∀ i: integer, if 1 ≤ i ≤ cap - Aggregate_Length, then
Sbling_Hldr(i) = Join(LBS(i),
Sbring_Hldr(First_Chld(i)) o Sbling_Hldr (Nxt_Sb(i))) ∧
cap = |First_Chld(First_Free)|;

Initialization
uses cap, Sbling_Hldr, First_Free, First_Chld, Nxt_Sb, Lbl;
var F_S: I_Stack.Stack;
var index: Integer;
Set_bdds_and_Init(Lbl, 1, max_cnt);
Set_bdds_and_Init(Nxt_Sb, 1, max_cnt);
Set_bdds_and_Init(First_Chld, 1, max_cnt);
Set_bdds_and_Init(Sbling_Hldr, 1, max_cnt);
index := 1;
Maintaining ∀ i,j: N if 1 ≤ i,j ≤ index
First_Chld(i) = First_Chld(j) iff i = j;
While index < max_cnt do
    First_Chld(index) := index + 1;
end;
index := index + 1;
end
First_Free := 1;
Push(First_Free, F_S);
end;

Type Nstd_Lst_Psn = L_Record.Record;

Conventions 1 ≤ cmt ≤ max_cnt ∧
1 ≤ root ≤ max_cnt ∧
max_cnt = cap + Aggregate_Length ∧
∀ m,n: [1, Nstd_Lst_Psn.Last_Specimen_Num]
Nstd_Lst_Psn(m).root = Nstd_Lst_Psn.Denotation.root ≠ 0
then \( m = n \land \forall i: [1, \max \_ cnt], \)

\[ \text{First\_Chld}(i).\text{Next} \neq \text{Nstd\_Lst\_Psn.Denotation}(n).\text{root}; \]

**Initialization**

*/ Each nested list position is automatically initialized according to the initialization specified for records /*

**Correspondence**

\( \forall P\#: \text{Nstd\_Lst\_Psn} \)

\[ \text{ConcP\#.Rem} = \text{Sbling\_Hldr}(P\#.\text{cmt}) \]

\( \forall i: \text{Integer}, \rho: \text{Str}(\text{Integer}) \)

such that \( \text{Loc\_trace} = i \circ \rho \)

\[ \text{ConcP\#.Prec} \circ \text{ConcP\#.Rem} = \text{Sbling\_Hldr}(i) \]

\( \forall \rho, \sigma: \text{Str}(\text{Stack}), S: \text{Stack}, \)

if \( P\#.\text{Trace} = \rho \circ S \circ \sigma \) and

\( \forall i, j: \text{integer}, \tau, \xi: \text{Str}(\text{integer}), \)

if \( S = \tau \circ i \) and \( S = j \circ \xi \) then

\[ \text{ConcP\#.Path} = \rho^' \circ S^' \circ \sigma^' \]

where \( S^' \) is the site corresponding to \( S \) and \( \rho^', \sigma^' \): \text{Str}(\text{Site})

On the first line of the body is the \( M\_\text{Conv\_Cond} \). In this case the module wide invariant is that the max value is not bigger than the max_count of the conceptual module. In other words the realization cannot store more items than the specs have indicated. The module conventions also require that the adjunct variable be kept up to date and that the storage for different nested lists be kept disjoint, that is, each nested list has its own unshared indices into the global arrays of the realization module and its own stacks for keeping track of movement within the list.

Next is the type definition, simple in form because of the \text{L\_Record} module declaration in the heading. The components of this module instance have already been described.

The definitions are given strictly to make later descriptions easier. These definitions set up terminology used in the lemma which establishes a functional relationship between integers on the stack called \text{Loc\_Trace} and strings of trees in the adjunct variable. This relationship helps in writing both the type conventions and the
correspondence.

The correspondence is long, although it is not complicated, because the conceptual variables of type Nstd_Lst_Psn form a cartesian product consisting of three components and hence each of the three must be described. For any P of type Nstd_Lst_Psn, the unvisited part of the list is exactly what appears in the adjunct array at the current index, Sbling_Hlr(P.cmt). Using the deepest entry in the Loc_Trace as index and concatenating the visited and unvisited portions of the list yield the Sbling_Hldr entry indicated. Finally, to get the path, we consider each stack on Trace. Associated with each stack (integer string) is a corresponding site. The definitions introduced into the module permit expression of this concept. The string of these sites corresponding to the Sibling Integer Strings which comprise the stacks entered on Trace make up the path.

Having examined in detail the realiztion module for Nested Lists (except for the procedures whose code will be shown elsewhere), our next logical step is to apply the realization module declaration rule. However, because some of the assertions, such as the correspondence, are lengthy, it will be to our advantage to choose some abbreviations. The goal in representing certain clauses in condensed form is not to save space, but rather to direct our thinking toward using the specifications to verify program correctness and not at the specifications themselves. This is entirely appropriate in view of the fact that we have just scrutinized the specifications both formally as mathematical clauses and informally as ideas expressed in natural language.

With the goal of verification in mind, we choose abbreviations based on observations we have made in considering what assertions are needed in the various hypotheses of the rule in question.
We note that the conceptual module constraint, \( C_{M_{\text{Req}}} \), and the realization convention, \( M_{Conv_{\text{Cond}}} \) together require that the structures meet a size limit. Let's call this pair of assertions "In_bdd." The module convention also requires that the adjunct variable be kept up to date and that nested list representations be kept disjoint, that is no two Traces or Loc_Traces overlap and no two nested list instances share any indices into the array, Lst_Hldr. This is suggestive of the abbreviation "adj_update \& DsJoint."

The role of the correspondence is the same for every module, there being no special syntactic distinction peculiar to this particular module, and so I choose "corres" as the abbreviation for the correspondence.

VIII.4. Correctness Proofs

With these abbreviations in mind, we turn to the Realization Module Declaration Rule presented in Chapter 4. The rule has already been described in detail, and it has been applied to the stack example. What is new here are our notational conveniences in the form of abbreviations to use in applying this rule to the Nested List Realization.

Our strategy is to look at an easy proof first to get used to the notation, rather than proceed in the order in which the code of the conceptual module and the realization module has been written.

With this plan in mind, we begin with the Boolean functions since they are the simplest procedures. Their code is short and in all cases, no changes are made to any nested lists, each being passed as a constant parameter a Nstd_Lst_Psn, \( L \).
Using the Correct_Op_Hyp which appears in Chapter 4, we can abbreviate the first Assume clause as:

\[
\text{Assume } \text{In}_{\text{bdd}} \land \text{Array} \cdot \text{Cstrnt} \land \text{adj}_\text{update} \land \text{DsJoint};
\]

This is followed by Remember. The next Assume clauses replace the formal parameters by the actual parameters. In the case of the Boolean procedures, there is only one parameter called L. There is no precondition since these Boolean procedures are functional. So we have

\[
\text{Assume Corres;}
\]

Next is the body. In the case of At_Outside we have:

\[
\begin{align*}
\text{with } L \\
\quad \text{if Is}_\text{Empty}(L, \text{Trace}) \\
\quad \text{then } \text{At} _\text{Out} := (\text{true}); \\
\quad \text{else } \text{At} _\text{Out} := (\text{false}); \\
\end{align*}
\]

Finally comes the Confirm clause:

\[
\text{Confirm } \text{In}_{\text{bdd}} \land \text{Up}_\text{date} \land \text{DsJoint} \land \text{post}
\]

Now we apply the with and if then else rules getting two branches to check, the true branch and the false branch, but applying the assignment rule in each case leaves us with true = true in one case and false = false in the other. The final clause generated is

\[
\begin{align*}
\text{In}_{\text{bdd}} \land \text{Up}_\text{date} \land \text{DsJoint} \Rightarrow \\
\text{In}_{\text{bdd}} \land \text{Up}_\text{date} \land \text{DsJoint}
\end{align*}
\]

The other Boolean procedures are handled similarly, the only difference in their proofs being the specific code of the procedure body.

Proofs, including code, for all but two of the other procedures appear in Appendix D. The rules in use are exactly those already illustrated in the stack proof. The complete list of rules appears in Appendix A.

I have singled out two particular procedures whose proofs illustrate what is new in nested lists and did not show up as an issue in stacks, and proofs for these procedures will be given next.
VIII.4.1. The Adjust Procedure

The first procedure is Adjust, which consists entirely of adjunct code having the purpose of updating the adjunct array, Sbling_Hldr. Adjust(L) receives as constant parameter a Nstd_Lst_Psn, L for which copies are made of x.Trace and x.Loc_Trace. Although our usual programming practices do not include making copies if at all avoidable, in this case, because the code is adjunct, complexity no longer matters since the code will not really be executed. However, it is important that it could execute and that if it were executed, it would terminate.

Since any positions of L which have been visited have indices on one of the stacks on Trace, the adjunct array Sbling_Hldr can be updated if all the values for indices on those stacks are adjusted to reflect whatever change has been made.

The code achieves this updating by using a nested while loop in which each stack is popped from Temp_Trace (a copy of Trace) and then the resulting stack is popped to get each index into Sbling_Hldr which needs adjusting. The adjustment is made and this process continues until Temp_Trace is empty. The loop invariant requires that for all indices not in Temp_Trace Sbling_Hldr has the correct values. Hence, when Temp_Trace is empty, all the values in Sbling_Hldr are up to date. This fact contributes toward what must be true in order for module conventions to hold.

Adjust is a procedure which exists only at the realization level and so its specifications appear along with the code. The precondition for Adjust is that for all indices between one and max Sbling_Hldr values are correct unless those indices are on a stack of x.Trace.
At the beginning of the procedure some variables of appropriate types are declared in which to make copies of \textit{x.Trace} and \textit{x.Loc_Trace}. These are called \textit{Temp_Trace} and \textit{Temp_Loc_T} respectively. \textit{Res_I} is used to hold whatever index is popped from \textit{Temp_Loc_T}.

The rest of the procedure is a nested while loop, the outer loop popping stacks one at a time from \textit{Temp_Trace} and the inner loop popping integers one at a time from \textit{Temp_Loc_T} to get the necessary indices of the portions of \textit{Sbling_Hldr} which must be updated. Because termination is necessary here, the specifications include not only loop invariants, but also progress metrics for both inner and outer loops.

For the outer loop we have chosen the progress metric to be the length of \textit{Temp_Trace}. Since the stack is popped during each iteration, the value of this expression, a natural number, decreases as the loop executes. These requirements allow us to show one of the hypotheses of the termination while rule. For the inner loop, we make the progress metric the length of \textit{Temp_Loc_T} for the same reasons as above.

For the outer loop the invariant states that the values in \textit{Sbling_Hldr} are up to date for all indices which do not appear in the Temp_Trace stack. The inner loop invariant requires that the entries in \textit{Sbling_Hldr} are correct for all indices not in either \textit{Temp_Trace} or \textit{Temp_Loc_T}.

Here is the code:

```pascal
proc Adjust(const x: Nstd_Lst_Psn);

require \(\forall i, \text{if } 1 \leq i \leq \text{max} \land \forall \alpha, \beta: \text{Str I Stack.Stack}, L: \text{I Stack.Stack},\)
\(\text{if } x = \alpha \circ L \circ \beta \land \forall \gamma, \delta: \text{Str integer},\)
\(j: \text{integer, if } L = \gamma \circ j \circ \delta \land \exists p, \sigma: \text{Str(integer)}\)
\(\text{such that } x.Loc\_Trace = p \circ j \circ \sigma \text{ and } i \neq j\) then
\(\text{Sbling\_Hldr}(i) = \text{Join}(\text{LBSl}(i), \text{Sbling\_Hldr(First\_Chld}(i)) \circ\)
\(\text{Sbling\_Hldr}(\text{Nxt\_Sb}(i)));
```
terminate_with ∃ i, if 1 ≤ i ≤ max,
then Sbling_Hldr(i) = Join(LBSl(i), Sbling_Hldr(First_Chld(i)) o
Sbling_Hldr(Nxt_Sb(i));

var Temp_Trace: S_Stack.Stack;
var Temp_Loc_T: I_Stack.Stack;
var Temp_Res: integer;

with x
Temp_Loc_T := Loc_Trace;
Temp_Trace := Trace;
end;
Push(Temp_Trace, Temp_Loc_T);

maintaining ∀ i, if 1 ≤ i ≤ max ∧ 
∀ α, β: Str(I_Stack.Stack), L: I_Stack.Stack,
if Temp_Trace = α o L o β ∧ ∀ γ, δ: Str integer,
j: integer, if L = γ o j o δ and i ≠ j then
Sbling_Hldr(i) = Join(LBSl(i), Sbling_Hldr(First_Chld(i)) o
Sbling_Hldr(Nxt_Sb(i));

decreasing Trace_Size = |Temp_Trace|;

While Temp_Trace ≠ ∧ do
  Pop(Temp_Trace, Temp_Loc_T);
  maintaining ∀ i, if 1 ≤ i ≤ max ∧
∀ α, β: Str_I_Stack.Stack, L: I_Stack.Stack,
if Temp_Trace = α o L o β ∧ ∀ γ, δ: Str integer,
j: integer, if L = γ o j o δ and i ≠ j ∧
∀ μ, ν: Str integer, k: integer, if
Temp_Loc_T = μ o k o ν and i ≠ k then
Sbling_Hldr(i) = Join(LBSl(i), Sbling_Hldr(First_Chld(i)) o
Sbling_Hldr(Nxt_Sb(i));

decreasing Loc_Trace_Size = |Temp_Loc_T|;
While Temp_Loc_T ≠ ∧ do
  Pop(Temp_Loc_T, Temp_Res);
  Sbling_Hldr(temp_res) := Join(LBSl(Temp_Res),
Sbling_Hldr(First_Chld(Temp_Res)) o
Sbling_Hldr(Nxt_Sb(Temp_Res));
end;
end;

Note the use of assignment statements wherever quick copy operations are
available, such as for stacks.
In verifying Adjust, I will emphasize what is new, the termination rule. The other constructs have been illustrated before in detail.

In order to verify the Adjust procedure, it is necessary to apply the procedure declaration rule. In this case the only parameter to our procedure is the constant \( x \) of type Nstd_Lst_Psn. As the rule indicates, the new specificatons are put into the context. To verify correctness the following hypotheses must be shown:

\[
\text{Remember; Assume} \; \text{pre}_{\text{Adjust}}(x); \; \text{body;}
\]

\[
\text{Confirm} \; \text{post}_{\text{Adjust}}(x);
\]

In examining the body, the verifier first generates assertions which occur as a result of processing the declarations, in this case stack and integer constraints for the appropriate variables. The new assume clause generated is:

\[
\text{Assume Stack.Cstrnt(Temp_Trace) \wedge Stack.Cstrnt(Temp_Loc_T) \wedge Int.Cstrnt(Res_I)};
\]

Now the verifier proceeds backwards over the body. The code consists of four main constructs, two assignments, a call to the Push procedure and a while loop. The first rule applicable is hence the while rule and in this case it is the termination while rule which is necessary. That rule generates the following hypotheses to check:

1. \( \text{code1;} \; \text{Confirm O.Inv} \)
2. \( \text{Assume Trace.Size} = |Temp_Trace| \wedge \\
   \text{Temp.Trace} \neq \Lambda; \wedge \\
   \text{O.Inv}; \; \text{body; Confirm O.Inv} \wedge \\
   |Temp_Trace| < \text{Trace.Size}; \\
3. \( \text{Temp.Trace} = \Lambda \wedge \text{O.Inv} \Rightarrow \text{post}; \\

(1) In the same spirit as we chose abbreviations previously, we will use O.Inv to suggest the outer loop invariant. Moreover, we will name the outer loop ‘Out_Loop’ so that a variable by that name can be used to save the evaluation of the progress metric expression as called for in the rule.

\text{Code1} consists of an Assume clause, two assignments and a call to Push. O.Inv states that for all indices not on Temp_Trace or on Temp_Loc_T the Sibling_Hldr values are correct. Backing through code1, since the Push operation placed all the
indices of \( Temp\_Loc\_T \) of the given nested list onto \( Temp\_Trace \) and the two assignments made exact copies of \( x\_Trace \) and \( x\_Loc\_Trace \), the precondition guarantees the invariant since it requires that any integers between 1 and \( max \) not on \( x\_Trace \) or \( x\_Loc\_Trace \) have their Sbling\_Hldr entries correct.

Formally:

\[
\text{Remember; Assume pre}[L \rightarrow x] \land \text{Stack.Cstrn}(x, \text{Trace}) \land \\
\text{Stack.Cstrn}(x, \text{Loc}_T) \land \text{Int.Cstrnt}(\text{Res}_I); \text{code1}; \\
\text{Confirm \text{O.Inv}}
\]

Note that the \text{O.Inv} is given in terms of \( Temp\_Trace \). Applying the call rule to \( \text{Push} \) results in getting \text{O.Inv} in terms of \( Temp\_Trace \circ Temp\_Loc\_T \). The assignments then give the \text{O.Inv} in terms of \( x\_Trace \circ x\_Loc\_Trace \). The invariant, now written with its arguments \( x\_Trace \) and \( x\_Loc\_Trace \) is exactly the precondition with the appropriately substituted arguments.

(2) The second hypothesis requires a look at the body of the outer loop. This body consists of a call to the \( \text{Pop} \) procedure followed by another while loop. Proceeding backwards through this body the verifier first applies the termination while rule which yields as before 3 hypotheses:

(4) \text{code2}; \text{Terminate with} \ I\_Inv \land |\text{Temp}\_Loc\_T| + 2 > 0;
(5) \text{Assume} \ Loc\_Trace\_Size = |\text{Temp}\_Loc\_T| \land \\
\text{Temp}\_Loc\_T \neq \Lambda \land \text{I.Inv}; \text{body}; \\
\text{Terminate with} \ I\_Inv \land |\text{Temp}\_Loc\_T| < \text{Loc\_Trace}\_Size;
(6) \text{Temp}\_Loc\_T = \Lambda \land I\_Inv \Rightarrow \text{O.Inv}

(4) \text{code2} consists of the statements preceding the inner while loop. This includes the assumptions of (2). The inner invariant requires that for all indices of the Sbling\_Hldr the corresponding values are correct except for those on either \( Temp\_Trace \) or on \( Temp\_Loc\_T \). This follows from the precondition and the outer invariant.
(5) This hypothesis requires a look at the body of the inner loop which includes a call to the Pop operation and an assignment. Backing over the assignment and the call statements results in having the progress metric decrease because now the Temp_Loc_T has lost an element and hence the length has decreased and so the Terminate_with clause holds. The loop invariant holds because the element which has been removed from Temp_Loc_T has been used as an index into Sbling_Hldr to assign the correct value to that part of the array.

(6) Once Temp_Loc_T is empty then the O_Inv holds because it requires that for all indices not on Temp_Trace the Sbling_Hldr is correct, but the I_Inv guarantees this.

Finally, (3) Once Temp_Trace is empty, since the loop invariant requires that Sbling_Hldr is up to date for all indices not on x.Trace the entire array must be up to date. That is exactly the postcondition of Adjust.

VIII.4.2. The Swap_Label Procedure

Knowing that the Adjust procedure works correctly, it is interesting to check correctness for a procedure which calls it, showing that we can take advantage of the fact that the Adjust procedure has been verified. This means that all we need to use about that procedure are its specifications.

Here is the code:

```pascal
Swap_Label(var L: Nstd_Lst_Psn, var labl: Item)
    with L
        Swap(LBSl(cmt), labl);
    end;
    Adjust(L);
end;
```

Since the Swap_Label is a procedure specified in the conceptual module for Nested Lists, the module rule must be applied. In particular, the Correc_Op_Hyp is
needed. Applying the rule generates:

\[
\text{Assume } \text{In}_b\text{dd} \land \text{adj}_\text{update} \land \text{DsJoint} \land \\
\text{int}_\text{Cstrnt}(\text{next}_\text{free}) \land \text{Array}_\text{Cstrnt}(\text{LBSl}) \land \text{Array}_\text{Cstrnt}(\text{First}_\text{Chld}) \land \\
\text{Array}_\text{Cstrnt}(\text{Next}_\text{Sb}) \land \text{Array}_\text{Cstrnt}(\text{Sbling}_\text{Hdr}); \\
\text{Remember: } \\
\text{Assume } \text{Conv}_\text{Cond}[L] \land \text{Corres}[L] \land \\
\text{L} \neq \text{\#}; \text{body}; \text{Alter } \text{conc}_\text{L}, \text{labl}; \\
\text{Confirm } \text{Conv}_\text{Cond}[L] \land \text{In}_\text{bdd} \land \text{Up}_\text{date} \land \\
\text{DsJoint} \land \text{Corres} \Rightarrow \exists \rho, \sigma: \text{NFT}_\text{Str}, \text{l: Item such that} \\
\#\text{conc}_\text{L}.\text{Unv} = \text{Join}(\text{l}, \rho) \circ \sigma \land \\
\text{conc}_\text{L}.\text{Unv} = \text{Join}(\text{labl}, \rho) \circ \sigma.
\]

To make the proof easy to read, let's call the last confirm statement "Q." Applying the Alter rule puts a \( \forall \) symbol in front of the variables which may change. Next, backing through the body, we apply the procedure call rule to Adjust(L). This generates:

\[
\text{Assume } \text{Nstd}_\text{Lst}_\text{Cstrnt}; \\
\text{code1; Confirm } \text{pre}_\text{Adjust}[L] \land \\
\forall ?L \text{ post}_\text{Adjust}[\#x \rightarrow L, x \rightarrow ?L] \\
\Rightarrow Q[L \rightarrow ?L]
\]

code1 refers to the statements preceding the call, in this case a with statement.

Applying the with rule yields:

\[
\text{code2; Assume } \text{Path} = L.\text{Path} \land \text{Vis} = L.\text{Prec} \land \\
\text{Unvis} = L; \text{Swap}(\text{LBSl, labl}); \text{Alter } L; \\
\text{Assume } L.\text{Path} = \text{Path} \land L.\text{Prec} = \text{Vis} \land \\
L = \text{Unvis}; \text{Confirm } Q'
\]

Here code 2 refers to the assertions preceding the with statement. \( Q' \) stands for \( Q[L \rightarrow ?L] \) after the Alter rule has put \( \forall \) in front of L. Although the swap procedure has not been explicitly written here, this is a procedure which exchanges the values of the variables indicated. The precondition is true. Hence the ultimate effect of applying the call here is to interchange the values of its two variable parameters, say \( x \) and \( y \).

Formally:

\[
\text{Confirm } \text{true;} \land \forall ?L, ?\text{labl} \\
\text{post}_\text{swap}[\#x \rightarrow L, x \rightarrow ?L, \#y \rightarrow \text{labl}, y \rightarrow ?\text{labl}] \\
\Rightarrow Q'[L \rightarrow ?L, \text{labl} \rightarrow ?\text{labl}]
\]

Call this Q'.
The remember rule removes the # symbols. Now everything is in terms of Assume and Confirm clauses. Applying the assume and confirm rules yields a single implication:

\[
\text{In\_bdd} \land \text{Update} \land \text{DsJoint} \land \\
\text{int\_Cstrnt}(\text{nxt\_free}) \land \text{Array\_Cstrnt}(\text{LBSl}) \land \\
\text{Array\_Cstrnt}(\text{First\_Child}) \land \\
\text{Array\_Cstrnt}(\text{Nxt\_Sb}) \land \text{Array\_Cstrnt}(\text{Sbling\_Hldr}) \land \\
\text{Conv\_Cond}[L] \land \text{Corres}[L] \land \\
L \neq \Lambda \implies Q''
\]

The post condition of the Adjust procedure establishes the Update predicate of Q'', as well as the correspondence. The other parts of Q'' follow directly from the original assumptions.

Proofs and code for the remaining procedures of the Nested List Module are in Appendix D.

The two proofs presented in this chapter have illustrated some significant facts about our proof system. First, the addition of progress metrics to address total correctness did not, in any way, change either our mechanical approach to verification or our modular approach to writing programs.

A second important point is that writing loop invariants and progress metrics, even for a complicated structure, turns out to be straightforward. Expressing these important parts of the specification depended not on obscure mathematical theorems, but rather on an understanding of what the program needed to do in order to carry out the desired functionality.

Finally, we have had a chance to see the beauty of reusing even previously completed proofs. In Chapter 4, the stack realization module was verified and in this chapter, we were able to take advantage of that fact when we used stacks in the realization of our Nested List Module. Additionally, we saw that once the Adjust procedure had been verified, any calling procedures needed to know only the
specifications of Adjust, relying on the fact that Adjust had already been verified.

These facts promote the idea that verification, even of complex structures, is practical. They also remind us that reusability goes well beyond copying of templates and macro expansion of code.
CHAPTER IX
Summary and Conclusions

IX.1. Specification and Verification

This thesis has presented results of research performed under the general heading of software engineering and the particular area of formal verification of program correctness.

Although initially, the goal was to specify and verify some useful, non-trivial code, during the course of pursuing that goal, a wide variety of issues emerged. Each of these issues was intimately related to the initial concern and therefore required resolution, so that while the goal was eventually met with the definition and verification of the Nested List Facility, it was necessary to complete a substantial collection of tasks before achieving the original goal.

In developing some understanding of just what we mean by useful, non-trivial code, we had to address issues of efficiency, language support, and clarity of specifications both from the formal point of view and from that of a user. Because it became clear that the formal specification and verification of programs is a costly activity, the question of reusability emerged as being of paramount importance.
IX.2. Reusability

The title of this thesis contains two key words: verification and reusability. Both have appeared frequently throughout the document in a variety of settings. Several examples of verification (proving correctness) have been given, and in every chapter, illustrations have appeared to indicate how some portion of a system could be reused. However, there are some subtle points about reusability which merit additional attention.

One topic frequently addressed by both academia and the world of industrial applications is "reusable code." Examples of this can be found in almost any commercial setting. For example, a programmer has written a procedure to compute solutions to a certain class of differential equations and has used it in a program for predicting economic growth. At a later time, the programmer has to write code for determining some trajectory angles which can be found by solving differential equations from the same class. The programmer hopes he can use the code he has already written and used and has established some confidence in. This is one view -- the most commonly held view of what is meant by reusability.

The view of reusability promoted in this thesis includes not only the commonly known practice of using code from one program in another which needs to perform the same computation, but much more. In fact, we have seen a multiplicity of ways to reuse all parts of a system. At the specification level, we reuse mathematical theories. For example, stacks, queues, and linked lists can all be described in terms of string theory. At the realization level, there may be several implementations for the same conceptual module, each yielding efficiency in a different setting. A realization may
lead to several instances, each with a distinct set of parameters. There can be stacks of integers, stacks of reals, stacks of stacks of queues of integers, each using the same realization code and the same conceptual specifications.

With all these levels of reusability, a programmer will be able to put together a program using a library of verified modules, doing his reasoning at the conceptual level. The programmer needs to know nothing about the implementation details of the verified modules, but only what the modules do (the specifications). Moreover, checking correctness of the newly composed program can be carried out without reverifying the modules on which it is based. Hence, even the proof of correctness can be reused.

IX.3. Verification and Total Correctness

In our view, verification means a formal mathematical treatment of correctness, and this requirement for rigor has directed us to consider some previously unexplored issues. In particular, we found it necessary in describing correspondences, which tell how to recover a conceptual object from its realization counterpart and to address the problem of termination, even for partial correctness.

The terminating while rule and the termination rules for mutually recursive procedures, together with proofs of their soundness and relative completeness in a general abstract object setting, are new contributions to the literature. The availability of these rules opens two areas with respect to proving correctness. First is the obvious advantage of being able to provide total correctness proofs for programs - - even those with complex user defined types.

A more subtle, but equally powerful use of the termination rules involves the use of adjunct variables in the specifying of correspondences between conceptual and real
objects of a given type. The termination rules allow us to update adjunct variables with adjunct code, thereby taking advantage of the usual proof techniques for programs to establish that certain useful intermediate objects exist, rather than resorting to different existence proofs in mathematics.

IX.4. Productivity

So far, my summary has suggested a purely theoretical thesis, but in fact, a primary goal of this work, identified in the introduction, is that of increasing the productivity of software engineers. All of the tools introduced in this thesis contribute toward that goal.

The separation of the conceptual module (specifications) from the realization module (implementation) allows the programmer to think purely in terms of the conceptual objects, completely free from implementation details. Hence, reasoning about programming variables can be done at an appropriate level of abstraction, thereby encouraging emphasis on the question of what the program is supposed to do, rather than on how to do it.

This separation facilitates the writing of correct programs on the one hand because of this capability of reasoning at the conceptual level, but at the same time, it leaves open the choice of implementation so as to promote efficiency. As we have pointed out, for each conceptual module there may be multiple realizations, each yielding special efficiencies in some particular setting.

Tools which promote correctness and efficiency make obvious contributions toward productivity, but the REFORMS system does not stop there. The layering of the modular structure permits and encourages a high degree of reusability as already described. Enhancements of modules allow reuse of the basic functionality while
adding additional operations for use in particular circumstances. While other systems may be reusing sections of code, in REFORMS we are reusing multiple components of our systems, and more importantly, we have formal specifications to tell us what those components do and rules to verify that they are correct.

Another contributor to productivity is ease of use, and the uniformity of treatment in REFORMS makes it an easy system to use. Primitive types and user defined types are specified and implemented using the same techniques. REFORMS makes it possible to build a library of verified modules ready for reuse in a variety of environments by selecting the parameters appropriately. Because even the primitive modules have parameters, such as size, one achieves hardware independence. For example, the integer module can be used on a large machine or a small one by proper choice of size parameters. Moreover, issues related to hardware concerns, such as arithmetic overflow, are taken care of by the specifications.

In fact the specifications serve many purposes. Not only do they make verification possible, they also give clear, unambiguous statements of functionality for the user so that he can make proper use and reuse of system components. This includes using the components only within their proper domains. Experienced programmers know that it is frequently the seemingly trivial concerns which cause delays, thereby decreasing productivity. For example, a programmer writes code to compute a function such as the natural log for real numbers. The code works correctly on the domain for which it is intended. But if the code is inserted into some environment which permits values whose logs are undefined, the program may fail and those who are trying to figure out what's wrong cause a serious slow down in production. Such a slow down need not occur when the specifications explicitly restrict the domain.
One may say that a certain domain is implied by the expression of the function but that is little help in identifying errors and their causes. In REFORMS the specifications for every procedure includes explicit, precise statement of its domain so that there is no guess work about "implied" domains. As a result, software productivity increases because every module built within the framework of REFORMS has precise specifications so that there is no guess work in deciding on whether or not it fits a particular need. Every module is mathematically verifiable so that users can rely on program correctness, and is reusable for these reasons.

Finally, in this section on productivity, it is appropriate to focus attention on the communal modules as introduced in Chapter 5. These modules permit implementations which permit all variables of a given type to share storage, thereby promoting efficient use of space.

IX.5. Future Research

The first part of this chapter has summarized what issues this thesis has addressed and resolved. However, the research leading to these results has introduced some additional questions to consider.

At present a syntactic slot for complexity, "duration," exists. The efficiency goal of REFORMS requires a mechanism to formally specify and verify the computational complexity of a program. The goal here is to design such a mechanism so that its use can be modularized in the same way that the rest of the proof system works, i.e. once the computational complexity for one program component has been specified and verified, it should be possible to verify the complexity claims about any component containing it without additional work on the previously verified component.
Another topic of interest which has arisen as a result of the research leading to this thesis concerns arrays. In defining arrays so that, regardless of length or content, all arrays share the same conceptual module, we raise questions concerning multi-dimensional arrays. These structures are obviously different from the traditional multi-dimensional arrays common in tensor or vector space theory. The point is that new names are necessary to distinguish between these different structures and conceptual modules written for them.

The realization for nested lists was written with the possibility of doing an enhancement in which a copy procedure appears. In setting up the shared storage, I was careful to keep indices only for moving in one direction. One reason for doing so is that such an implementation is more likely to yield a "quick" copy method for nested lists. With such a copy procedure, additional operations useful in editing such as replacing strings of items and searching for occurrences of a particular item, become highly efficient.

In REFORMS each type is defined in terms of some mathematical domain. Associated with each mathematical domain is a theory. So far we have been placing the domain name on the module Uses list and assuming that its theory comes with the name. At the stage of automatically generating intermediate assertions by the verifier, approach is fine. However, when the assertion generator interfaces with a theorem prover, specific syntax for describing the mathematical theory will be needed.

One last topic I want to mention for future research is concurrency, a particularly active area of research. Of particular interest are the following questions: Is it possible to specify concurrency adequately? If so, how does one formally do so? Will the conceptual notions used in REFORMS play an important role in a satisfactory concurrent programming environment?
IX.6. Conclusions

By developing syntax and verification rules for module structures and applying these tools to a real example by specifying, implementing, and proving the correctness of the Nested List Module, this thesis has shown that it is possible to achieve mathematical rigor without sacrifice of efficiency.

By exhibiting primitive module specifications as well as hierarchically built modules, this work has demonstrated that uniformity of treatment with regard to structures is possible in REFORMS.

By formulating sound and relatively complete termination rules for while loops and recursive procedures, this thesis has provided a total correctness proof system.

By designing and implementing a module in the programming style of REFORMS, this dissertation has illustrated how a programmer can achieve efficiency and formality while enforcing the principles of information hiding, separation of concerns, clarity, and modularity.
APPENDIX A
Verification Rules

Assignment Rule:

\[ \text{C \ code; Confirm } Q[x \rightarrow \text{Spec Exp(expr)}] \]

\[ \text{C \ code}; \ x := \text{expr}; \text{ Confirm } Q; \]

where Spec Exp is defined by:
\[ \text{Spec Exp}(x) = x \]
\[ \text{Spec Exp}(c) = c \text{ for any constant } c \]
\[ \text{Spec Exp}(f(e_1,e_2)) = M_E[x \rightarrow \text{Spec Exp}(e_1), y \rightarrow \text{Spec Exp}(e_2)] \]

where proc f(val x,y: T1): T2
\[ \text{ensure } f = M_E(x,y) \]

Swap Rule:

\[ \text{C \ code; Confirm } Q[x \rightarrow y, y \rightarrow x] \]

\[ \text{C \ code; } x := y; \text{ Confirm } Q; \]

Confirm Rule:

\[ \text{Math Ext(C)}} \ \ Q \]

\[ \text{C \ Confirm } Q; \]
Assume Rule:

\( C \text{\ code; Confirm } P \Rightarrow Q; \)

-------------------------------------------

\( C \text{\ code: Assume } P; \text{ Confirm } Q; \)

Alter Rule:

\( C \text{\ code; Confirm } \forall x: T, Q; \)

-------------------------------------------

\( C \text{\ code: Alter } x; \text{ Confirm } Q; \)

where \( C = C' \cup \{ x: T \} \)

Remember Rule:

\( C \text{\ code: Confirm } \text{Shift}(Q); \)

-------------------------------------------

\( C \text{\ code; Remember; Confirm } Q; \)

where \( \text{Shift}(Q) \) is defined by:

- \( \text{Shift}(x) = x \)
- \( \text{Shift}(c) = c \) for any constant \( c \)
- \( \text{Shift}(#) = x \)
- \( \text{Shift}(f(e1,e2)) = f(\text{Shift}(e1),\text{Shift}(e2)) \)

If Then Else Rule:

\( C \text{\ code; Assume } B; \text{ code1; Confirm } Q; \)

\( C \text{\ code; Assume } \neg B; \text{ code2; Confirm } Q; \)

-------------------------------------------

\( C \text{\ code; If } B \text{ then code1 else code2; endif; Confirm } Q; \)

While Rule:

\( C \text{\ code; Confirm Inv;} \)

\( C \text{\ Assume } B \wedge \text{Inv; body; Confirm Inv;} \)

\( C \text{\ Confirm } \neg B \wedge \text{Inv } \Rightarrow Q; \)

-------------------------------------------

\( C \text{\ code; Maintaining Inv While } B \text{ do Loop_Name: body end; Confirm } Q; \)
Definition Rule:

\[ C \cup \{ \text{def} \} \backslash \text{Aspt} ; \text{Assume } f = \exp_{x,y,g,T2}; \]

\[ C \backslash \text{Aspt} ; \text{def } f(\text{val } x: T); T1 = \exp_{x,y,g,T2}; \]

Procedure Call Rule:

\[ \text{Assume } T.C\text{stmt} \]
\[ C\backslash \text{code}; \text{Confirm } \text{pre}[x \rightarrow a] \land \]
\[ \forall ?a \text{ post}[\#x \rightarrow a, x \rightarrow ?a], \Rightarrow \]
\[ Q[a \rightarrow ?a]; \]

\[ C\backslash \text{code}; p(a: T); \text{Confirm } Q \]

Procedure Declaration Rule:

\[ C \cup \{ \text{p_heading} \} \backslash \text{Remember}; \text{Assume } \text{pre}; \text{p_body}; \text{Confirm } \text{post}; \]
\[ \text{code; Confirm } Q; \]

\[ C \backslash \text{proc } p(\text{var } x, \text{const } y); \]
\[ \text{require } \text{pre}; \text{ensure } \text{post}; \]
\[ \text{p_body}; \text{code; Confirm } Q; \]
\[ \text{code; Confirm } Q; \]

With rule for records:

\[ C\backslash \text{code}; \text{Assume } x = R.x \land y = R.y; \text{code1}; \text{Alter } R; \]
\[ \text{Assume } R.x = x \land R.y = y; \text{Confirm } Q; \]

\[ C\backslash \text{code}; \text{with } R \text{ code1}; \text{end; Confirm } Q; \]

Conceptual Module Declaration Rule

\[ C \cup \{ C.M \} \backslash \text{code}; \text{Confirm } Q; \]

\[ C\backslash C.M; \text{code; Confirm } Q; \]
where \( C_M \) is:

\[
\text{Conceptual Module } C_M(N(C_Param_List))
\]

\[
\text{Require } C_M.\text{Req/cu, cg} \\
\text{Uses } C_Use_List
\]

\[
\text{Type } T = \text{Math}_T; \text{ exemplar } t : T; \\
\text{Constraints } \text{Stmt}_\text{Cond/t, cu, cg, ch} \\
\text{Initially } \text{Init}_\text{Cond/t, cu, cg, ch} \\
\text{Proc p(var } cx : T, \text{ const cy : T);} \\
\text{Require } \text{pre/cx, cy, cu, cg, ch} \\
\text{Ensure } \text{post/cx, cy, #cx, cu, cg, ch}
\]

and \( C_Param_List \) is:

\[
(\text{val cu : CT1, Type CT2, Domain CD1,} \\
\text{ def cg(val z : CT3) : CT4})
\]

and \( C_Use_List \) is:

\[
\text{def ch, Type CT25; (*math theories*)}
\]

Realization Module Declaration Rule

\[
\text{Well Def Corr Hyp} \\
C'M_Init Hyp \\
C'T_Init Hyp \\
C'C_Correct Op Hyp \\
C \cup \{C_M\} \cup \{R_M_H\} \text{ aspt; code; Confirm Q;}
\]

\[
\text{R}_M \text{ is the Realization Module:}
\]

where \( R_M \) is:

\[
R_M.H; \\
R_M.D; \\
R_M.B
\]

and \( R_M.H \) is the Realization Module Heading:

\[
\text{Realization Module } R_M.N(R_Param_List); \\
\text{for } C_M.N; \\
\text{Uses } R_Use_List; \\
\text{Require } M_Req/\text{rv, rf, cu, cg, ch}
\]

and where \( C' = C \cup \{C_M\} \cup \{C_Param_List\} \cup \{R_Param_List\} \cup \{R_M.D\} \)
and where $R\_Param\_List$ is:

```
val rv: RT2; Type RT5; Domain RD1;
def rf(val rx: RT13): RT6;
proc rq(var ry: RT14);
   Require pre\rq_{\text{ry},\text{rf},\text{rv}}\end
   Ensure post\rq_{\text{ry},\text{#ry},\text{rf},\text{rv}}\end
```

and where $R\_Param\_List$ is $R\_Param\_List$ with "Type RT5"
replaced by
Type RT5 = Domain; exemplar x;
Constraints RT5.Cstmt(x);
Initially RT5.Init(x);

and where $C\_Param\_List$ is $C\_Param\_List$ with
"Type CT2" replaced by
Type CT2 = Domain1; exemplar y;
Constraints CT2.Cstmt(y);
Initially CT2.Init(y);

and where $R\_Use\_List$ is:
Def rh, Type RT9, Proc rq2, Domain RD2;
Conceptual_Module CM2, Realization_Module RM2;

and where $R\_M\_D$ is Realization Module Declarations:
```
Module T_Inst is CM1(cu1,CT26, cg1, CD2)
   Realized_by RM1(rv1,RT27,rf1,rq2, RD2);
```

```
var rz: RT7;
adj_var rw: RT8;
```

and $R\_M\_B$ is Realization Module Body:
```
Conventions M_Conv_Cond\cu,\text{rv},\text{rw},\text{rz},\text{rf},\text{cg},\text{ch}\end
Initialization M_I_Body end;
Type T = RT20;
Conventions Conv_Cond\t,\text{rv},\text{rw},\text{rf},\text{cu},\text{cg},\text{rz},\text{ch}\end
Correspondence Corr_Exp\conc,t,\text{cu},\text{rv},\text{rw},\text{rz},\text{rf},\text{cg},\text{ch}\end
Initialization I_Body end;
proc p(var x: T; const y: T)
p_body
end;
end
```
and where \textit{Well\_Def\_Corr\_Hyp} is:
\begin{align*}
\text{Assume } & C_{M\_Req/\text{cu},\text{cg},\text{ch}} \land M_{\text{Req/\text{cu},\text{cg},\text{ch},rv,rf}} \land M_{\text{Conv\_Cond/\text{cu},\text{ch},rv,\text{rw},\text{rz},\text{rf},\text{cg}}} ; \\
\text{Confirm } & \forall x:\ RT20 \text{ if } Conv\_Cond[t \rightarrow x]/\text{cu,\text{ch},rv,\text{rw},\text{rz},\text{rf},\text{cg}] \\
\text{then } & \exists y: \text{Math}_T \text{ such that} \\
\text{Corr\_Exp/} & \text{conc.} t \rightarrow x]/\text{cu,\text{ch},rv,\text{rw},\text{rz},\text{rf},\text{cg}] \land \\
\text{Cstmt\_Cond/} & t \rightarrow y]/\text{cu,\text{cg},\text{ch}} .
\end{align*}

and where \textit{M\_Init\_Hyp} is:
\begin{align*}
\text{Assume } & C_{M\_Req/\text{cu},\text{cg},\text{ch}} \land M_{\text{Req/\text{cu},\text{cg},\text{ch},rv,rf}} \land \\
RT2. & \text{Cstmt}(rv) ; \\
M\_I\_Body ; \\
\text{Confirm } & M_{\text{Conv\_Cond/\text{cu},rv,\text{rw},\text{rz},\text{rf},\text{cg,\text{ch}}} ; \\
\text{and where } & T\_Init\_Hyp \text{ is:} \\
\text{Assume } & C_{M\_Req/\text{cu},\text{cg},\text{ch}} \land M_{\text{Req/\text{cu},\text{cg},\text{ch},rv,rf}} \land \\
M\_\text{Conv\_Cond/} & \text{cu,rv,\text{rw},\text{rz},\text{rf},\text{cg,\text{ch}}} \land RT2.\text{Cstmt}(rv) \land \\
RT7. & \text{Cstmt}(rz) \land RT8.\text{Cstmt}(rw) ; \\
\text{var } & t: RT20 ; \\
I\_Body ; \\
\text{Confirm } & \text{Conv\_Cond/} t,\text{cu,\text{ch},\text{rv,\text{rw,\text{rz,rf,cg}}}] \land \\
M\_\text{Conv\_Cond/} & \text{cu,rv,\text{rw},\text{rz},\text{rf,\text{cg,\text{ch}}} ; \\
( & \text{Corr\_Exp/} \text{conc.} t,\text{cu,rv,\text{rtw,\text{rz,rf,cg,\text{ch}}} } \land \\
\text{Cstmt\_Cond/} & t \rightarrow \text{conc.} t]/\text{cu,\text{cg,\text{ch}}} = > \\
\text{Init\_Cond/} & t \rightarrow \text{conc.} t]/\text{cu,\text{cg,\text{ch}}}
\end{align*}

and where \textit{Correct\_Op\_Hyp} is:
\begin{align*}
\text{Correct\_Op\_Hyp} & \text{ is:} \\
\text{Assume } & C_{M\_Req/\text{cu},\text{cg,\text{ch}}} \land M_{\text{Req/\text{cu},\text{cg,\text{ch},rv,rf}} \land \\
M\_\text{Conv\_Cond/} & \text{cu,\text{ch,rv,\text{rw},\text{rz},\text{rf},\text{cg}}} \land RT2.\text{Cstmt}(rv) \land \\
RT7. & \text{Cstmt}(rz) \land RT8.\text{Cstmt}(rw) ; \\
\text{Remember;} \\
\text{Assume } & \text{Conv\_Cond/} t \rightarrow x]/\text{cu,\text{ch,rv,\text{rw,\text{rz,rf,cg}}} \land \\
\text{Conv\_Cond/} & t \rightarrow y]/\text{cu,rv,\text{rz,rf,\text{ch,\text{cg}}} \land \\
\text{Corr\_Exp/} & t \rightarrow \text{conc.} x]/\text{cu,rv,\text{rz,rf,\text{cg}}} \land \\
\text{Cstmt\_Cond/} & t \rightarrow \text{conc.} x]/\text{cu,\text{cg,\text{ch}}} \land \\
\text{Cstmt\_Cond/} & t \rightarrow \text{conc.} y]/\text{cu,\text{cg,\text{ch}}} \land \\
\text{pre/} x \rightarrow \text{conc.} x, \ y \rightarrow \text{conc.} y]/\text{cu,\text{cg,\text{ch}}} ; \\
\text{body;} \text{Alter conc.} x ; \\
\text{Confirm } & M_{\text{Conv\_Cond/\text{cu,\text{ch,rv,\text{rw,\text{rz,rf,cg}}} \land C_{M\_Req/\text{cu,\text{cg,\text{ch}}} \land } \\
\text{Conv\_Cond/} & t \rightarrow x]/\text{cu,rv,\text{rx,rf,\text{ch,\text{cg}}} \land \\
\text{Corr\_Exp/} & t \rightarrow \text{conc.} x]/\text{cu,rv,\text{rv,\text{rz,rf,\text{cg}}} \land \\
\text{Cstmt\_Cond/} & t \rightarrow \text{conc.} x]/\text{cu,\text{cg,\text{ch}}} => \\
\text{post/} x \rightarrow \text{conc.} x, \ #x \rightarrow \text{conc.} x, \ y \rightarrow \text{conc.} y]/\text{cu,\text{cg,\text{ch}}}
Module Instantiation Rule

\[ C_{M} \text{_Req} \{cu \rightarrow iu, cg \rightarrow ig\} \land \]
\[ M_{Req} \{rv \rightarrow iv, rf \rightarrow if\} \land \]
\[ \text{pre}_{\text{req}} \{cy \rightarrow iy, rf \rightarrow if, rv \rightarrow iv\} \Rightarrow \text{pre}_{\text{req}} \{iy, if, iv\} \land \]
\[ \text{post}_{\text{req}} \{iy, #iy, if, iv\} \Rightarrow \]
\[ \text{post}_{\text{req}} \{cy \rightarrow iy, #cy \rightarrow #iy, rf \rightarrow if, rv \rightarrow iv\} \]
\[ C'' \text{\_ code; Confirm Q; } \]

\[ C' \text{/ aspt; } \text/_M; \text{ code; Confirm Q; } \]

where \(_M is:

\[ \text{Module Inst is } C_{M\_N} \{(iu, IT2, ig) } \]
\[ \text{Realized by } R_{M} \{(iv, IT5, if, iq) \} \]

and where \( C' = C \cup \{C_{M}\} \cup \{R_{M\_H}\} \cup \)
\[ \{\text{proc } \text{iq}(\text{var } iy: IT11) \}; \]
\[ \text{Require } \text{pre}_{\text{iq}} \{iy, if, iv\}; \]
\[ \text{Ensure } \text{post}_{\text{iq}} \{iy, #iy, if, iv\} \} \] and

\[ C'' = C' \cup \{\text{Type Inst.T } = \text{Math.T, } \}
\[ \text{T.Init_Cond, T.Cstmt} \cup \]
\[ \{\text{Proc Inst.p(\text{var } ix: T, const iy: T) } \}
\[ \text{Require } \text{pre}_{\text{Inst.p}} \{ix, iy\}; \]
\[ \text{Ensure } \text{post}_{\text{Inst.p}} \{ix, #ix, iy\} \} \]

Variable Declaration Rule

\[ C' \text{/ aspt; Assume(Inst.T.Cstrnt(x) } \land \text{Inst.T.Init(x)) } \]
\[ \text{code; Confirm Q; } \]

\[ C' \text{/ aspt; var x: Inst.T; code; Confirm Q; } \]

where \( C' = C \cup \{\text{Type Inst.T } = \text{Math.T, } T\text{.Init_Cond, } \}
\[ T\text{.Cstmt} \} \)
APPENDIX B
Primitive Modules

B.1. Characters

Conceptual Module Character_Facility;
uses Default_Int_Fac;
dom Alph;
    constraints {"0", "1", ..., "9"} ⊆ Alph;
def Init_Char: Alph;
def max_rep, min_rep: Default_Int;
    constraint: min_rep < max_rep;
def rep: Alph -> [min_rep, max_rep];
    constraints ∀ x,y: Alph, if rep(x) = rep(y) then
       x = y ∧ ∀ n: Default_Int if min_rep ≤ n ≤ max_rep
       then ∃ x: Alph such that rep(x) = n;
type Char ≤ Alph;    exemplar C
initially C = Init_Char;

proc Are_Equal(const x,y: Char): boolean;
    ensure Are_Equal = (x = y);

proc Replicate(const x: Char): Char;
    ensure Replicate = x;

proc Min_Rep(): Default_Int;
    ensure Min_Rep = (min_rep);

proc Max_Rep(): Default_Int;
    ensure Max_Rep = (max_rep);

proc Int_Rep(const x: Char): Default_Int;
    ensure Int_Rep = (rep(x));

proc Char_Convert(const n: Default_Int, var res: Char);
require min_rep \leq n \leq max_rep;
ensure n = rep(res);

end Character_Facility;

Enhancement Int_Convert_Capability
for Character_Facility(type other_int);

proc Cnvrt_Dig(const d: Char, var res: other_int);
  require d \in \{"0", "1", ..., "9"\};
  ensure res = 0 if d = "0"
    1 if d = "1"
    ...
    9 if d = "9";

proc gen_0_ch(): Char;
proc gen_1_ch(): Char;
...

end Enhancement;

B.2. Strings

Conceptual_Module Primitive_String_Facility(val max_length;
  type Item);
  require max_length \geq 0;
  type P_Str \subseteq Str_of Item;
  exemplar S;
  constraints |S| \leq max_length;
  initially S = \Lambda;
  var New_Str: P_Str:

proc Extend_New_Str(var x: Item);
  uses New_Str;
  require |New_Str| < max_length;
  ensure New_Str = #New_Str o #x \land Item.Init(x);

proc New_Str_Length(): Integer;
  uses unch New_Str;
  ensure New_Str_Length = |New_Str|;

proc Take_New_Str(var S: P_Str);
uses New_Str;
ensure S = #New_Str ∧ New_Str = A;

proc Swap_Entry(var S: P_Str; unch p: integer;
    var val: Item);
    require 1 ≤ p ≤ |S|;
    ensure ∀ α, β: Str_of Item, ∀ χ: Item,
    if #S = α · χ · β ∧ |α| = p - 1, then
    S = α · #val · β ∧ val = χ;

proc Length_of(unch S: P_Str): Integer;
    ensure Length_of = |S|;
end Primitive_String_Facility;

B.3. Records

Conceptual_Module Record_Facility(type T1,T2)

    Type Record = Cart_Prod;
        P1: T1,
        P2: T2
    end;
    example R: Record;
    initially
        T1.init(P1(R)) and
        T2.init(P2(R));

procedure Swap1(var x: T1; var R: Record);
    ensure x = P1(#R) and P1(R) = #x and
        P2(R) = P2(#R);

procedure Swap2(var y: T2; var R: Record);
    ensure y = P2(#R) and P2(R) = #y and
        P1(R) = P1(#R);
end Record_Facility
B.4. Arrays

Conceptual Module Array_Facility (const max_size: integer;
  type Item);
  require max_size > 0;

  type Array = Cart Prod
    valu: integer -> Item,
    l_bd, u_bd: integer;
  end;
  example a: Array;

  constraints a.l_bd <= a.u_bd + 1 ∧
    a.u_bd < a.l_bd + max_size;

  initially
    a.l_bd = 1 ∧ a.u_bd = 0;

  procedure Set_Bdd(var A: Array, const low, hi: integer);
    require A.l_bd = 1 ∧ A.u_bd = 0 ∧
    low ≤ hi;
    ensure A.l_bd = low ∧ A.u_bd = hi ∧
      ∀ i, if low ≤ hi then Item.Init(A.valu(i));

  procedure Swap_Entry (var A: Array; const i: integer;
    var v: Item);
    require A.l_bd ≤ i ≤ A.u_bd;
    ensure A.l_bd = #A.l_bd and A.u_bd = #A.u_bd and
    A.valu(i) = #v and v = #A.val(i) and
    for all j: integer if j ≠ i then A.valu(j) = #A.val(j);

  procedure Lower_Bound (const A: Array): integer;
    ensure Lower_Bound = A.l_bd;

  procedure Upper_Bound (const A: Array): integer;
    ensure
APPENDIX C
Tree Theory

Trees labeled with alphabet $\Gamma$

Signature: $\langle \text{Tr}(\Gamma), \text{T_Str}, \Omega, \Lambda, \text{Join}, \text{Ext} \rangle$

$\Omega: \text{Tr}(\Gamma), \Lambda: \text{T_Str}, \text{Join}: \Gamma \times \text{T_Str} \rightarrow \text{Tr}(\Gamma)$
$\text{Ext}: \text{T_Str} \times \text{Tr}(\Gamma) \rightarrow \text{T_Str}$

Axioms:

I. (a) For all $\alpha: \text{T_Str}$, $x: \Gamma$, $\text{Join}(x, \alpha) \neq \Omega$.
(b) For all $\alpha: \text{T_Str}$, $T: \text{Tr}(\Gamma)$, $\text{Ext}(\alpha, T) \neq \Lambda$.

II. (a) For all $\alpha, \beta: \text{T_Str}$, $x, y: \Gamma$, if $\text{Join}(x, \alpha) = \text{Join}(y, \beta)$
then $x = y$ and $\alpha = \beta$.
(b) For all $\alpha, \beta: \text{T_Str}$, $T, U: \text{Tr}(\Gamma)$, if $\text{Ext}(\alpha, T) = \text{Ext}(\beta, U)$ then $\alpha = \beta$ and $T = U$.

III. For all $A \subseteq \text{Tr}(\Gamma)$ and $B \subseteq \text{T_Str}$,
if (i) (a) $\Omega \in A$ and
(b) $\Lambda \notin B$ and
(ii) if $T \in A$ and $\alpha \in B$ then
(a) $\text{Join}(x, \alpha) \in A$ and
(b) $\text{Ext}(\alpha, T) \notin B$
then $A = \text{Tr}(\Gamma)$ and $B = \text{T_Str}$

Definition (by induction): $\alpha \circ \beta$

$\circ: \text{T_Str} \times \text{T_Str} \rightarrow \text{T_Str}$

(i) $\alpha \circ \Lambda = \alpha$
(ii) $\alpha \circ \text{Ext}(\beta, T) = \text{Ext}(\alpha \circ \beta, T)$

Theorem 19: $\alpha \circ (\beta \circ \gamma) = (\alpha \circ \beta) \circ \gamma$

Definition of height (by induction): $h(T)$ and $M_H(\alpha)$

$h: \text{Tr}(\Gamma) \rightarrow \mathbb{N}$, $M_H: \text{T_Str} \rightarrow \mathbb{N}$

(i) (a) $h(\Omega) = 0$
(b) $M_H(\Lambda) = 0$
(ii) (a) $h(\text{Join}(x, \alpha)) = \text{suc}(M_H(\alpha))$
(b) $M_H(\text{Ext}(\alpha, T)) = \text{max}(M_H(\alpha), h(T))$

Definition (node_count): $N_C(T), N_Cts(\alpha)$

223
\[ N_C : \text{Tr}(\Gamma) \rightarrow \mathbb{N}, N_{\text{Cts}} : \text{T}_\text{Str} \rightarrow \mathbb{N} \]

(i) (a) \( N_C(\Omega) = 0 \)
    (b) \( N_{\text{Cts}}(\Lambda) = 0 \)

(ii) (a) \( N_C(\text{Join}(x, \alpha)) = \text{suc}(N_{\text{Cts}}(\alpha)) \)
    (b) \( N_{\text{Cts}}(\text{Ext}(\alpha, T)) = N_{\text{Cts}}(\alpha) + N_C(T) \)

**Definition (leaf\_count):** \( L_C(T), L_{\text{Cts}}(\alpha) \)

- \( L_C : \text{Tr}(\Gamma) \rightarrow \mathbb{N}, L_{\text{Cts}} : \text{T}_\text{Str} \rightarrow \mathbb{N} \)
- (i) (a) \( L_C(\Omega) = 0 \)
  (b) \( L_{\text{Cts}}(\Lambda) = 0 \)
- (ii) (a) \( L_C(\text{Join}(x, \alpha)) = 1 \) if \( L_{\text{Cts}}(\alpha) = 0 \)
  \( L_{\text{Cts}}(\alpha) \) if \( L_{\text{Cts}}(\alpha) \neq 0 \)
  (b) \( L_{\text{Cts}}(\text{Ext}(\alpha, T)) = L_{\text{Cts}}(\alpha) + L_C(T) \)

**Theorem 20:** \( N_C(T) \geq h(T) \)

**Theorem 21:** \( L_C(T) \leq N_C(T) \)

**Definition:** Site = Cartesian Product

- \( \text{Labl}: \Gamma; \)
- \( \text{LBS}, \text{RBS}: \text{T}_\text{Str} \)

**end**

**Definition:** \( \text{T}_\text{Pos} = \text{Cartesian Product} \)

- \( \text{path}: \text{Str}(\text{Site}) \)
- \( \text{Rem}: \text{Tr}(\Gamma) \)

**end**

**Definition:** \( \text{T}_\text{Str}_\text{Pos} = \text{Cartesian Product} \)

- \( \text{path}: \text{Str}(\text{Site}) \)
- \( \text{Prec}, \text{Rem}: \text{T}_\text{Str} \)

**end**

**Definition (by induction) (Zip operation):** \( \rho \Psi T \)

- \( \Psi : \text{Str}(\text{Site}) \times \text{Tr}(\Gamma) \rightarrow \text{Tr}(\Gamma) \)

  (i) \( \Lambda \Psi T = T \)

  (ii) \( \text{Ext}\rho, S) \Psi T = \rho \Psi \text{Join}(S.\text{val}, S.\text{LBS} \circ T \circ S.\text{RBS}) \)

**Definition (positions in same Tree):** \( P \equiv_T Q \)

- \( \equiv_T : \text{T}_\text{Pos} \times \text{T}_\text{Pos} \rightarrow \text{Boolean} \)

- \( P \equiv_T Q \) iff \( P.\text{path} \Psi P.\text{Rem} = Q.\text{path} \Psi Q.\text{Rem} \)

**Definition (k\_way\_tree):** \( K_\text{Tr}(\Gamma) \_ \text{Tr}(\Gamma) \)

- \( T \in K_\text{Tr}(\Gamma) \) iff for all \( P: \text{T}_\text{Pos} \) such that...
P.path $\Psi$ P.Rem = $T$, either P.Rem = $\Omega$ or
there is $\alpha$: $T$ Str and $x$: $\Gamma$ such that P.Rem = $\text{Join}(x, \alpha)$ and $\text{loc} = k$.

Theorem 22: ($k$-Tree induction): For all $A \in k$ Tr($\Gamma$),
if (i) $\Omega \in A$ and
(ii) if for $1 \leq i \leq k$ $T_i \in A$ then $\text{Join}(x_i, \prod_{i=1}^k T_i) \in A$
then $A = k$ Tr($\Gamma$)

Theorem 23: if $T \in k$ Tr($\Gamma$) then $L_C(T) \leq \frac{k^{h(T)} - 1}{k - 1}$

Theorem 24: if $T \in k$ Tr($\Gamma$) then $N_C(T) \leq \frac{k^{h(T)} - 1}{k - 1}$

Definition: Suture
$\Xi$: Str(Site) $\times$ Tr.Str $\rightarrow$ Tr.Str
$\forall S$: Site, $\rho$: Str(Site), $\alpha, \beta$: Tr.Str,
$\Lambda \Xi \alpha = \alpha$
$\rho \circ S \Xi \alpha = \rho \Xi S.LS \circ \text{Join}(S.Lbl, \alpha) \circ S.RS$
APPENDIX D
Nested List Proofs

Proof of Module Initialization:

Applying the M_Init_Hyp:

Assume $\text{max}_\text{cnt} > 0 \land \text{integer}\.cstmt(\text{max}) \land \text{max} \leq \text{max}_\text{cnt}$;

Initialization

uses $\text{cap, Sbling_Hldr, First_Free, First_Chld, Nxt_Sb, Lbl}$;

var $F_S: \text{I_Stack.Stack}$;

var $\text{index}: \text{Integer}$;

Set_bdds_and_Init($\text{Lbl}$, 1, $\text{max}_\text{cnt}$);

Set_bdds_and_Init($\text{Nxt_Sb}$, 1, $\text{max}_\text{cnt}$);

Set_bdds_and_Init($\text{First_Chld}$, 1, $\text{max}_\text{cnt}$);

Set_bdds_and_Init($\text{Sbling_Hldr}$, 1, $\text{max}_\text{cnt}$);

$\text{index} := 1$;

Maintaining $\forall i, j: \mathbb{N} \text{ if } 1 \leq i, j \leq \text{index}$

First_Chld(i) = First_Chld(j) iff $i = j$;

While $\text{index} < \text{max}_\text{cnt}$ do

First_Chld($\text{index}$) := $\text{index} + 1$;

end;

$\text{index} := \text{index} + 1$;

end

First_Free := 1;

Push(First_Free, $F_S$);

end;

Confirm In_bdd $\land$ $\text{adj_update} \land$ $\text{Dsnt}$

Next is the type finalization:

Assume $\text{max}_\text{cnt} > 0 \land \text{max} \leq \text{max}_\text{cnt}$ $\land$

In_bdd $\land$ $\text{adj_update} \land$ $\text{Dsnt} \land$ $\text{Cstmts}$;
with L
    Push(root, F_S);
end;
end;

Confirm In_bdd ∨ adj_update ∨ Dsnt;

For the remaining procedures I use "Common_Pre" and "Common_Post" to stand for conditions to assume and confirm respectively for all the procedures.

Common_Pre is:
Assume max_cnt > 0 ∧ max ≤ max_cnt ∧
In_bdd ∧ adj_update ∧ Cstmts; Remember;

Common_Post is:
Confirm In_bdd ∧ adj_update ∧ Dsnt ∧
0 ≤ L.cmt, L.root ≤ max

To prove Proc Insert_List_Labl(L: Nstd_Lst_Psn, Lb: Item):
Confirm Corres \Rightarrow \text{conc.L.Path} = \#\text{conc.L.Path} \land \\
\text{conc.L.Prec} = \#\text{conc.L.Prec} \land \text{conc.L.Rem} = 1 \odot \#\text{conc.L.Rem};

Next is the procedure Delete_List(L: Nstd_Lst_Psn, Lb: Item)

\text{Common_Pre} \land \text{Assume } 0 \leq L\text{.cmt}, L\text{.root} \leq \text{max} \land \\
\text{Corres} \land \exists a: \text{Item}, \alpha: \text{NFT.Str} \text{ such that} \\
L\text{.Rem} = \text{Jn}(a, \Lambda \circ \alpha);

\text{var Res_S: I.Stack.Stack;}
\text{var Res_I: integer;}
\text{with L}
\quad \text{nxt_free} := \text{cmt};
\quad \text{Push}(\text{nxt_free}, \text{F_S});
\text{end;}
\text{If Is_Empty(Trace) \land Is_Empty(Loc_Trace) then cm t := 0;}
\text{If Not Is_Empty(Trace) and Is_Empty(Loc_Trace) then Pop(Trace, Res_S);}
\quad \text{Pop}(\text{Res_S}, \text{Res_I});
\quad \text{Copy(First_Chld(Res_I), Nxt_Sb(cmt));}
\quad \text{cmt} := \text{Nxt_Sb(nxt_free)};
\text{If Not Is_Empty(Loc.Trace) then Pop(Loc_Trace, Res_I);}
\quad \text{Copy(Nxt_Sb(Res_I), Nxt_Sb(cmt));}
\quad \text{cmt} := \text{Nxt_Sb(Res_I)};
\text{Push(Loc_Trace, Res_I);}
\text{end;}
\text{First_Chld(nxt_free) := 0;}
\text{Nxt_Sb(nxt_free) := 0;}
\text{Adjust(L);}
\text{end;}

\text{Common_Post} \land \text{Corres} \Rightarrow \text{conc.L.Path} = \#\text{conc.L.Path} \land \\
1 = a \land \\
\text{conc.L.Prec} = \#\text{conc.L.Prec} \land \text{conc.L.Rem} = \alpha;

\text{Proc Open_Sublist(L: Nstd_Lst_Psn)}
\text{Common_Pre} \land \text{Assume } 0 \leq L\text{.cmt}, L\text{.root} \leq \text{max} \land \\
\text{Corres} \land \text{L.Rem} \neq \Lambda;
\text{var St: I.Stack.Stack;}
\text{with L}
\quad \text{Push(} \text{Loc_Trace, cmt);}
\quad \text{Push(} \text{Trace, Loc_Trace);}
Swap(Loc_Trace, St);
Copy(cmt, First_Chld(cmt));
end;
end;
Common_pre ∧ Corres ⇒ ∃ S: Site such that conc.L.Path = #conc.L.Path o S ∧
conc.L.Rem = #conc.L.Prec o #conc.L.Rem ∧
S.Lft_Sblings = #conc.L.Prec;

Proc Close_Sublist(L: Nstd_Lst_Psn)
Common_Pre ∧ Assume 0 ≤ L.cmt, L.root ≤ max ∧
Corres ∧ L.Path ≠ ∅;
   var res_S: I_Stack.Stack;
   var res_I: Integer;
   with L
      Pop(Trace, res_S);
      Swap(Loc_Trace, res_S);
      Pop(Loc_Trace, res_I);
      Copy(cmt, Nxt_Sb(Res_I));
end;
end;
Common_Post ∧ Corres ⇒ ∃ S: Site such that #conc.L.Path = conc.L.Path o S ∧
S.Y #conc.L.Prec o #conc.L.Rem = conc.L.Rem ∧
conc.L.Prec = #conc.L.Prec;

Proc Advance(L: Nstd_Lst_Psn)
Common_Pre ∧ Assume 0 ≤ L.cmt, L.root ≤ max ∧
Corres ∧ L.Rem ≠ ∅;
   var Res_S: Stack;
   with L
      Push(Loc_Trace, cmt);
      Copy(cmt, Nxt_Sb(cmt));
end;
end;
Common_Post ∧ Corres ⇒
conc.L.Prec o conc.L.Rem = #conc.L.Prec o #conc.L.Rem ∧
|conc.L.Rem| = |#conc.L.Rem| - 1;

Proc Retreat(L: Nstd_Lst_Psn);
Common_Pre ∧ Assume 0 ≤ L.cmt, L.root ≤ max ∧
L.Prec ≠
var Res_I: Integer;
with L
    Pop(Loc_Trace, Res_I);
    Copy(cmt, Res_I);
end;
end;
Common_Post \& Corres =>
conc.L.Prec o conc.L.Rem = #conc.L.Prec o #conc.L.Rem \&
iconc.L.Prec = i#conc.L.Prec - 1;

Proc Loc_Reset(L: Nstd_Lst_Psn);
Common_Pre \& Assume 0 < L.cmt, L.root < max;
    var Res_I: integer;
    with L
        While Not Is.Empty(Loc_Trace) do
            Pop(Loc_Trace, Res_I);
            crnt := root;
        end;
end;
Common_Post \& Corres => conc.L.Path = #conc.L.Path
conc.L.Prec = #conc.L.Prec o #conc.L.Rem;

Proc At_Front(L: Nstd_Lst_Psn, At_L: Boolean)
Common_Pre \& Assume 0 < L.cmt, L.root < max;
    with L
        If Is.Empty(Loc_Trace)
            Then At_L = true;
            Else At_L = false;
        end;
end;
Common_Post \& Corres => At_L_end is true iff conc.L.Prec = #;

Proc At_Outside(L: Nstd_Lst_Psn, At_Out: Boolean)
Common_Pre \& Assume 0 < L.cmt, L.root < max;
    with L
        If Is.Empty(Trace);
            Then At_Out = true;
            Else At_Out = false;
        end;
end;
Common_Post \& Corres => At_Outside = true iff conc.L.Path = #;
Proc At_End(L: Nstd_Lst_Psn, At_E: Boolean)
Common_Pre \& Assume 0 \leq L.cmt, L.root \leq max;
  with L
    If Lst_Hldr(cmt).Nxt_Sb = 0
      Then At_E = true;
    Else At_E = false;
  end;
end
Common_Post \& Corres => At_E = true iff conc.L.Rem = \Lambda;

Proc Swap_Rem(L, M: Nstd_Lst_Psn)
Common_Pre \& Assume 0 \leq L.cmt, L.root \leq max \&
  0 < M.cmt, M.root \leq max \& Corres \&
  \exists T, S: NFT(\Sigma \rho, \sigma; NFT_Str) such that
  L.Rem = T \circ \rho \& M.Rem = \sigma \circ S;

  var Res_S: I_Stack.Stack;
  var Res_I: integer;
  with L
    Swap(cmt, tempn);
  end;
  with M
    Swap(cmt, tempn);
  end;
  with L
    If Is_Empty(Loc_Trace) \& Not Is_Empty(Trace)
      then Pop(Trace, Res_S);
      Pop(Res_S, Res_I);
      First_Chld(Res_I), cmt);
      Push((Res_I, Res_S);
      Push(Trace, Res_S);
      If Not Is_Empty(Loc_Trace)
        then Pop(Loc_Trace, Res_I);
        Copy(Nxt_Sb(Res_I), cmt);
        Push(Res_I, Loc_Trace);
    end;
  end;
  with M
    If Is_Empty(Loc_Trace) \& Not Is_Empty(Trace)
      then Pop(Trace, Res_S);
      Pop(Res_S, Res_I);
      Copy(First_Chld(Res_I), cmt);
Push((Res_I, Res_S);
   Push(Trace, Res_S);
   If Not Is_Empty(Loc_Trace)
       then Pop(Loc_Trace, Res_I);
       Copy(Nxt_Sb(Res_I), cmt);
   Push(Res_I, Loc_Trace);
end;

   Adjust(L);
   Adjust(M);
end;

Common_Post \land Corres \Rightarrow conc.L.Path = #conc.L.Path \land
conc.M.Path = #conc.M.Path \land conc.L.Prec = #conc.L.Prec \land
conc.M.Prec = #conc.M.Prec \land conc.L.Rem = S o \sigma \land
conc.M.Rem = T o \rho;

Proc Reset(L: Nstd_Lst_Psn)
Common_Pre \land Assume 0 \leq L.cmt, L.root \leq max \land
Corres
   var St: Stack of Stack of Integer;
   var Loc: Stack of Integer;
   with L
       Copy(cmt, root);
       Swap(Trace, St);
       Swap(Loc_St, Loc);
   end;
end;

Common_Post \land Corres \Rightarrow conc.L.Path = A \land
conc.L.Prec = A \land conc.L.Rem = A;
References


10. . Byte Magazine, Vol.6 No.8 on Smalltalk .


233


48. Ogden, W.F. CIS 680 Class Notes, OSU CIS Department.


