SOFTWARE SPECIFICATION:

A Comparison of Formal Methods

by
John D. Gannon, James M. Purtilo, Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland

February 23, 2001

Copyright ©1993

Contents

Preface ix

1 Introduction 1

1. A HISTORICAL PERSPECTIVE . . . 5
   - 1.1. Syntax . . . 5
   - 1.2. Testing . . . 6
   - 1.3. Attribute Grammars . . . 7
   - 1.4. Program Verification . . . 7
2. A BRIEF SURVEY OF TECHNIQUES . . . 9
   - 2.1. Axiomatic Verification . . . 9
   - 2.2. Algebraic Specification . . . 10
   - 2.3. Storage . . . 11
   - 2.4. Functional Correctness . . . 13
   - 2.5. Operational Semantics . . . 13
3. SEMANTICS VERSUS SPECIFICATIONS . . . 15
4. LIMITATIONS OF FORMAL SYSTEMS . . . 16
5. PROPOSITIONAL CALCULUS . . . 17
   - 5.1. Truth Tables . . . 19
   - 5.2. Inference Rules . . . 19
   - 5.3. Functions . . . 21
   - 5.4. Predicate Calculus . . . 22
   - 5.5. Quantifiers . . . 25
   - 5.6. Example Inference System . . . 26
6. EXERCISES . . . 27
7. SUGGESTED READINGS . . . 28

2 The Axiomatic Approach 31

1. PROGRAMMING LANGUAGE AXIOMS . . . 31

1.1. Example: Integer Division 34
1.2. Program Termination 38
1.3. Example: Multiplication 38
1.4. Another Detailed Example: Fast Exponentiation . 42
1.5. Yet Another Detailed Example: Slow Multiplication 45
2. CHOOSING INVARIANTS 47
3. ARRAY ASSIGNMENT 48
3.1. Example: Shifting Values in an Array 50
3.2. Detailed Example: Reversing an Array 53
4. PROCEDURE CALL INFERENCE RULES 57
4.1. Invocation 58
4.2. Substitution 59
4.3. Adaptation 62
4.4. Detailed Example: Simple Sums 66
4.5. Recursion in Procedure Calls 72
4.6. Example: Simple Recursion 73
5. EXERCISES 75
6. SUGGESTED READINGS 84

3 Functional Correctness 85
1. PROGRAM SEMANTICS 86
2. SYMBOLIC EXECUTION 88
3. DESIGN RULES 90
3.1. Design of Assignment Statements 90
3.2. Design of Conditional Statements 91
3.3. Verification of Assignment and Conditional Statements 92
4. SEMANTICS OF STATEMENTS 93
4.1. Begin Blocks 93
4.2. Assignment Statement 93
4.3. If Statement 94
4.4. While Statement 94
5. USE OF FUNCTIONAL MODEL 98
5.1. Example: Verification 98
5.2. Example: Design 103
5.3. Multiplication – Again 104
6. DATA ABSTRACTION DESIGN 107
6.1. Data Abstractions 107
6.2. Representation Functions 108
7. USING FUNCTIONAL VERIFICATION 111
8. EXERCISES 112
9. SUGGESTED READINGS 117

4 Predicate Transformers **119**

1. GUARDED COMMANDS . . . 119
   - 1.1. Guarded If Statement . . . 120
   - 1.2. Repetitive statement . . . 120
2. WEAKEST PRECONDITIONS . . . 122
   - 2.1. Axioms . . . 122
   - 2.2. If Statements . . . 122
   - 2.3. Do Statements . . . 125
3. USE OF WEAKEST PRECONDITIONS . . . 128
   - 3.1. Example: Integer Division . . . 128
   - 3.2. Still More Multiplication . . . 129
4. EXERCISES . . . 132
5. SUGGESTED READINGS . . . 134

5 Algebraic Specifications **137**

1. MORE ABOUT DATA ABSTRACTIONS . . . 137
2. OPERATIONAL SPECIFICATIONS . . . 139
3. ALGEBRAIC SPECIFICATION OF ADTS . . . 141
   - 3.1. Developing Algebraic Axioms . . . 144
   - 3.2. Hints For writing algebraic axioms . . . 147
   - 3.3. Consistency . . . 151
   - 3.4. Term Equality . . . 152
4. DATA TYPE INDUCTION . . . 152
   - 4.1. Example: Data Type Induction Proof . . . 155
5. VERIFYING ADT IMPLEMENTATIONS . . . 156
   - 5.1. Verifying Operational Specifications . . . 156
   - 5.2. Verifying Algebraic Specifications . . . 160
   - 5.3. Example: Verifying an Implementation of Stacks 162
   - 5.4. Verifying Applications With ADTs . . . 167
   - 5.5. Example: Reversing an Array Using a Stack . . . 168
6. INDUCTIONLESS INDUCTION . . . 171
   - 6.1. Knuth–Bendix Algorithm . . . 171
   - 6.2. Application of Knuth Bendix to induction . . . 176
   - 6.3. Example Using Knuth–Bendix . . . 179
7. EXERCISES . . . 183
8. SUGGESTED READINGS . . . 191

6 Denotational Semantics **193**

1. THE LAMBDA CALCULUS . . . 193
   - 1.1. Boolean Values as λExpressions . . . 195
   - 1.2. Integers . . . 196

2. Datatypes . . . 197
2.1. Continuous Functions . . . 199
2.2. Continuity . . . 200
2.3. Recursive Functions . . . 201
2.4. Evaluation of FACT Function . . . 203
3. PROGRAMMING SEMANTICS . . . 203
3.1. The Simple Model . . . 204
3.2. Pointers and Aliasing . . . 209
3.3. Continuations . . . 210
4. EXERCISES . . . 212
5. SUGGESTED READINGS . . . 213

7 Specification Models **215**
1. VIENNA DEVELOPMENT METHOD . . . 215
1.1. Overview of VDM . . . 216
1.2. Example: Multiplication One Last Time . . . 220
1.3. Summary of VDM . . . 222
2. TEMPORAL LOGIC . . . 223
2.1. Properties of Temporal Logics . . . 223
2.2. Programming Assertion . . . 228
3. RISK ANALYSIS . . . 231
3.1. Decisions under Certainty . . . 231
3.2. Decisions under Uncertainty . . . 232
3.3. Risk Aversion . . . 234
3.4. Value of Prototyping . . . 235
4. SUGGESTED READINGS . . . 237

References **239**

Author Index **245**

Index **247**

Preface

Formal methods help real programmers write good code.

Well, this is what we believe, anyway. Likewise, we believe verification techniques can scale up for use in large and realistic applications. But least you think we will next profess a belief in the tooth fairy or 'no new taxes' as well, let us quickly acknowledge that the field has considerable engineering to do before industry will accept the economics of treating software mathematically. Many different technologies must be developed and compared with one another, and their utility must be evaluated in realistic case studies.

But how to get this activity out of the campus labs and into the field?

Here at the Computer Science Department at the University of Maryland, recognizing that a corps of formalism-trained researchers and practitioners is necessary for specification technology to really see the growth it needs, we have focused our curriculum in software to stress the importance of being able to reason about programs. This theme affects our graduate and undergraduate programs alike, but it is the former program that this book is all about.

What is this book about?

During the prehistoric era of computer technology (say, about 1970), formal methods centered on understanding programs. At this time, we created the graduate course ***CMSC 630: Theory of Programming Languages***. The theme of the course was programming semantics, explaining what each programming language statement meant. This was the era of the Vienna Definition Language, Algol-68 and denotational semantics. Over the next ten years, program verification, weakest preconditions, and axiomatic verification dominated the field as the goal changed to showing that a program agreed with a formal description of its specification. During the 1980s, software engineering concerns and the ability to write a correct program from this formal specification followed. Our course kept evolving as the underlying technology changed and new models were presented. With all three models, however, the basic problem has remained the same, showing that a program and its formal description are equivalent. We now believe that this field has matured enough that a comprehensive book is possible.

How did we come to write this book?

Over the years that we have taught formal methods to graduate students, we found no one textbook that would compare the various methods for reasoning about software. Certainly there are excellent books, but each deals in only a single approach, and the serious student who seeks a comprehensive view of the field is faced with the task of tracking down original journal papers, translating arcane notations, and making up his or her own sample problems. So the book you are reading now represents our effort to consolidate the available formalisms for the purpose of detailed comparison and study.

This book started as a loose collection of handwritten notes by one of us (Gannon), who sought to stabilize our graduate course in its early years. These notes were cribbed and modified by many of the faculty at Maryland as they did their tour in front of the ***630*** classes. One day the second author (Purtilo) volunteered to teach ***630***, claiming that after years of writing good programs he thought it would be nice to be able to *demonstrate* that they were correct; and so the notes were passed to this wonderfully naive colleague. As an aid in learning the material before he had to teach it to the class, Purtilo was responsible for the initial cleanup and typesetting of the notes, and he expanded the document over the course of several semesters. Finally, the notes were passed to the third author (Zelkowitz). In needing to relearn this material after not teaching the course for about five years, he expanded the notes with new material and, with heroic efforts, repaired some of the second author's proofs. This collection is what is presented here today.

Who should read this book?

The punch line is that this is material we have used extensively, having been tested on many generations of graduate students, to whom we are greatly indebted for many excellent suggestions on organization, accuracy, and style. Of course, you don't need to be a graduate student looking for a research topic to benefit from this book. Even if you don't write code from a formal specification, we have found that an understanding of which program structures are easier to manipulate by formal techniques will still help you write programs that are easier to reason about informally. The basic technology is useful in all programming environments and an intuitive understanding of the techniques presented here are valuable for everyone.

Parts of Chapters 1, 3 and 6 are extensions to the paper "The role of verification in the software specification process" which appeared in ***Advances in Computers 36***, copyright ©1993 by Academic Press, Inc. The material is used with permission of Academic Press, Inc. Thanks go to our colleagues over the years who have directly commented on the shape of either the book or the course: Vic Basili, Rick Furuta, Dick Hamlet, Bill Pugh, Dave Stotts, and Mark Weiser. And a similar thanks to all the folks who indirectly contributed to this book, by having generated all the technology we hope we have adequately cited!

College Park, Maryland
January, 1994

John Gannon
Jim Purtilo
Marvin Zelkowitz

Introduction*

Chapter 1

The ability to produce correct computer programs that meet the needs of the user has been a long standing desire on the part of computer professionals. Indeed, almost every software engineering paper which discusses software quality starts off with a phrase like "Software is always delivered late, over budget and full of errors" and then proceeds to propose some new method that will change these characteristics. Unfortunately, few papers achieve their lofty goal. Software systems *continue* to be delivered late, over budget and full of errors.

As computers become cheaper, smaller, and more powerful, their spread throughout our technological society becomes more pervasive. While comic relief is often achieved by receiving an overdue notice for a bill of $.00 (which can only be satisfied by sending a check for $.00) or in getting a paycheck with several commas in the amount to the left of the decimal point (Alas, such errors are quickly found), the use of computers in real-time applications has more serious consequences.

Examples of such errors are many:

• Several people have died from radiation exposure due to receiving several thousand times the recommended dosage caused by a software error in handling the backspace key of the keyboard. Only mistyping and correcting the input during certain input sequences caused the

*Parts of this chapter, through Section 4, are a revision of an earlier paper [64]. Copyright ©1993 by Academic Press, Inc. Reprinted by permission of Academic Press.

error to appear, and was obviously not detected during program testing.

• Several times recently, entire cities lost telephone service and the national phone network was out of commission for almost ten hours due to software errors. While not an immediate threat to life, the lack of telephone service could be one if emergency help could not be called in time.

• Computers installed in automobiles since the early 1980s are moving from a passive to an active role. Previously, if the auto's computer failed, the car would still operate, but with decreased performance. Now we are entering an era where computer failure means car failure.

• The increase in fly-by-wire airplanes, where the pilot controls only activate computers which actually control the aircraft, are a potential danger source. No longer will the pilot have direct linkages back to the wings to control the flight. There is research in drive-by-wire automobiles using some of this same technology.

• Recently, a New York bank, because of a software error, "overspent" its resources by several billion dollars. Although the money was recovered the next day, the requirement to balance its books daily caused it to borrow this money from the Federal Reserve Bank, at an overnight *real* interest cost of $5 million.

If told to program an anti-lock braking system for a car, would you guarantee financially that it worked? Would you be the first person to use it?

It is clear that the need for correct software systems is growing. While the discussion that creating such software is too complex and expensive, the correct reply is that there is no other choice – we must improve the process. And, as has been demonstrated many times, it often does **not** require increased time or cost to do so.

What is a correct software system?

You probably have an intuitive understanding of the word "correct" that appeared several times already. The simple answer is that the program does what it is supposed to do. But what is that? In order to understand what it is supposed to, we need a description of what it should do. Right now we will informally call it the *specification*. A program is correct if it meets its specification.

Why are developing such precise specifications difficult? Some of the problems include:

• **scale**: The effort required to show even a small program to be partially correct can be great. This will be graphically illustrated in the problems at the end of each chapter.

• **semantics**: We do not remove sources of ambiguity simply by moving from English or a programming language to a mathematical notation. It is very difficult to "say what we mean." Consider the example of sorting an array A of integers: Mathematically we may elect to express our desired target condition as $\forall_i\ 0 \leq i < n, A[i] \leq A[i+1]$. Writing a program to yield this target condition is trivial... simply assign all array entries to be zero.

• **termination**: Many of our techniques will prove that ***in the case that a given program should finish,*** then the desired termination program state will be true. The catch, of course, is that we must determine when (if ever) the program terminates. This can often be more difficult to prove than it is to derive the desired output state.

• **pragmatics**: Even in the case that we show a program to be partially correct, and even show that it terminates, we are still faced with the problem of determining whether our ***compilers*** will correctly implement the basic language constructs used in our program, and likewise whether our run-time environment will maintain the correct input and execution conditions relied on by our program, and likewise whether our implementation will be affected by the finite precision imposed on the number systems of our computation.

It generally has been assumed that calling a program correct and stating that a program meets its specification are equivalent statements. However, meeting a specification is more than that. Correctness has to do with correct functionality. Given input data, does the program produce the right output? What about the cost to produce the software? The time to produce it? Its execution speed? All these are attributes that affect the development process, yet the term "correctness" generally only applies to the notion of correct functionality. We will use the more explicit term *verification* to refer to correct functionality to avoid the ambiguity inherent in the term "correctness."

A simple example demonstrates that system design goes beyond more than correct functionality. Assume as a software manager you have two options:

1. Build a system with ten people in one year for $500,000.
2. Build a system with three people in two years for $300,000.

Assuming both systems produce the same functionality, which do you build?

Correctness (or verification) does not help here. If you are in a rapidly changing business and a year's delay means you have lost entry to the market, then Option 1 may be the only way to go. On the other hand, if you have an existing system that will be overloaded in two to three years and eventually need to replace it, then Option 2 may seem more appropriate.

Rather than "correctness," we will use the term *formal methods* to describe methods which can help in making the above decision. Verification and the quality of the resulting program are certainly a major part of our decision making and development process. However, we also need to consider techniques that address other attributes. Some of these other attributes include:

- **Safety:** Can the system fail with resulting loss of life? With the growth of real-time computer-controlled systems, this is becoming increasingly important. Techniques, such as *software fault tolerance*, are available for analyzing such systems.

- **Performance:** Will the system respond in a reasonable period of time? Will it process the quantity of data needed?

- **Reliability:** This refers to the probability that this system will exhibit correct behavior. Will it exhibit reasonable behavior given data not within its specification? A system that crashes with incorrect data might be correct (i.e., always produces the correct output for valid input), but highly unreliable.

- **Security:** Can unauthorized users gain information from a software system that they are not entitled to? The privacy issues of a software system also need to be addressed. But beyond privacy, security also refers to *non-interference* (that is, can unauthorized users prevent you from using your information, even if they cannot get access to the data itself?), and also *integrity* (that is, can you warrant that your information is free from alteration by unauthorized users?).

- **Resource utilization:** How much will a system cost? How long will it take? What development model is best to use?

• **Trustworthiness:** This is related to both safety and reliability. It is the probability of undetected catastrophic errors in the system. Note that the reliability of a system + the trustworthiness of a system will not equal 1 since there may be errors that are not "catastrophic."

We can summarize some of these attributes by saying, "Note that we never find systems that are correct and we often accept systems that are unreliable, but we do not use systems that we dare not trust. ([9], P. 10).

Most of this book addresses the very important verification issue involved in producing software systems. However, we will also discuss some of these other issues later. In the next section we will briefly discuss verification from an historical prospective, and then we will briefly summarize the techniques that are addressed more fully in later chapters.

1. A HISTORICAL PERSPECTIVE

The modern programming language had its beginnings in the late 1950s with the introduction of FORTRAN, Algol-60, and LISP. During this same period, the concepts of Backus-Naur Form (BNF), context-free languages, and machine-automated syntax analysis using these concepts (e.g., parsing) were all being developed.

Building a correct program and elimination of "bugs"[1] was of concern from the very start of this period. It was initially believed that by giving a formal definition of the syntax of a language, one would eliminate most of these problems.

1.1. Syntax

By *syntax*, we mean what the program looked like. Since this early period, BNF has become a standard model for describing how the components of a program are put together. Syntax does much for describing the meaning of a program. Some examples:

[1] A term Grace Hopper is said to have coined in the late 1950s after finding a bug (insect variety) causing the problem in her card reading equipment.

• **Sequential execution.** Rules of the form:

$$< stmtlist > \rightarrow < stmt >; < stmtlist > \ | \ < stmt >$$

do much to convey the fact that execution proceeds from the first $< stmt >$ to the remaining statements.

• **Precedence of expressions.** The meaning of the expression $2+3\times4$ to be $2 + (3 \times 4) = 14$ and not $(2 + 3) \times 4 = 20$ is conveyed by rules like:

$$< expr > \rightarrow < expr > + < term > \ | \ < term >$$
$$< term > \rightarrow < term > \times < factor > \ | \ < factor > .$$

In this case, $\times$ has "higher" precedence than $+$.

However, there is much that syntax cannot do. If a variable was used in an expression, was it declared previously? In a language with block structure like Ada or Pascal, which allows for multiple declarations of a variable, which declaration governs a particular instance of an identifier? How does one pass arguments to parameters in subroutines? Evaluate the argument once on entry to the subroutine (e.g., *call by value*) or each time it is referenced (e.g., *call by reference*, *call by name*)? Where is the storage to the parameter kept?

All these important issues and many others cannot be solved by simple syntax, so the concept of programming *semantics* (i.e., what the program means) developed.

1.2. Testing

Now it is time for a slight digression. What is wrong with program testing? Software developers have been testing and delivering programs for over 40 years.

The examples cited at the beginning clearly answer this question. Programs are still delivered with errors, although the process is slowly improving.

As Dijkstra has said, "Testing shows the presence of errors, not their absence." He gives a graphic demonstration of the failure of testing. To test the correctness of $A + B$ for 32-bit integers A and B, one needs to perform $2^{32} \times 2^{32} \sim 10^{20}$ tests. Assuming 10^8 tests per second (about the limit in 1990s technology), that would require more than 30,000 years of testing.

Testing is only an approximation to our needs. Verification is the important concept.

1.3. Attribute Grammars

Probably the first semantic model was the attribute grammar of Knuth. In this case, attributes (values) were associated with each node in the syntax tree, and this information could be passed up and down the tree to derive other properties. The technique is particularly useful for code generation in a compiler. For example, the expression evaluation problem above could be solved by the following set of attributes for producing Polish postfix for expressions:

$$\begin{array}{lll}
< expr > \rightarrow & < expr > + < term > & Postfix(expr_1) = Postfix(expr_2) \\
 & & \;||Postfix(term)||+ \\
 & | \; < term > & Postfix(expr) = Postfix(term) \\
< term > \rightarrow & < term > \times < factor > & Postfix(term_1) = Postfix(term_2) \\
 & & \;||Postfix(factor)||\times \\
 & | \; < factor > & Postfix(term) = Postfix(factor)
\end{array}$$

where x_1 and x_2 refer to the left and right use, respectively, of that nonterminal in the production, and $||$ refers to the concatenation operator.

With these rules, it is clear that given the parse tree for the expression $2 + 3 \times 4$, its correct postfix is $2\ 3\ 4\ \times\ +$ yielding the value 14.

While useful as a technique in compiler design, attribute grammars do not provide a sufficiently concise and formal description of what we have informally called the specification of a program.

1.4. Program Verification

The modern concept of program verification was first introduced by Floyd [16] in 1967. Given the flowchart of a program, associated with each arc is a predicate that must be true at that point in the program. Verification then consisted of proving that given the truth of a predicate before a program node, if that node was executed, then the predicate following the node would be true. If this could be proven for each node of the flowchart, then the internal consistency of the entire program could be proven. We would call the predicate associated with the input arc to the program the input condition or *precondition*; the predicate

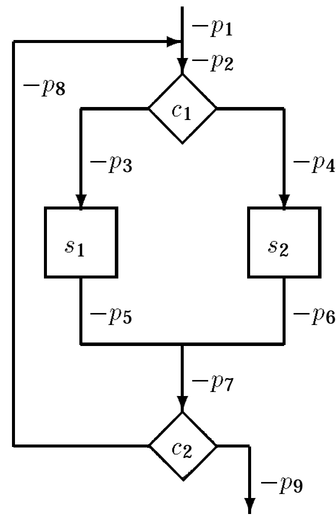


Figure 1.1. Floyd verification model

on the output arc to the program the output condition or *postcondition*; and the pair would be our *specification*, since they determined the input/output behavior of the program.

For example, Figure 1.1 contains four program nodes (c_1 and c_2 are conditional expressions and s_1 and s_2 are statements) and nine predicates ($p_1, \ldots, p_9$) describing properties about the program. In order to show the correctness of this program, we have to prove the following propositions:

$p_1 \lor p_8 \Rightarrow p_2$	$p_2 \land c_1 \Rightarrow p_3$
$p_2 \land \neg c_1 \Rightarrow p_4$	p_3 *and* s_1 *executes* $\Rightarrow p_5$
p_4 *and* s_2 *executes* $\Rightarrow p_6$	$p_5 \lor p_6 \Rightarrow p_7$
$p_7 \land c_2 \Rightarrow p_8$	$p_7 \land \neg c_2 \Rightarrow p_9$

Once we have determined the effects of statements s_1 and s_2, all except one of these propositions are fairly straightforward. The one problem is the very first:

$$p_1 \lor p_8 \Rightarrow p_2$$

Since p_2 depends upon p_8, and developing p_8 depends upon all the previous $p_i s$, the process of generating predicates for loops in the program

becomes very difficult. It would take a further development by Hoare to fix this problem.

In 1969, Hoare [29] introduced the axiomatic model for program verification which put Floyd's model into the formal context of predicate logic. His basic approach was to extend our formal mathematical theory of predicate logic with programming language concepts. His basic notation was: $\{P\}S\{Q\}$, meaning that if P were the precondition before the execution of a statement S, and if S were executed, then postcondition Q would be true. Since a program is a sequence of statements, we simply needed a set of axioms to describe the behavior of each statement type and a mechanism for executing statements sequentially. As will be shown later, this model simplifies but does not eliminate the problems with loop predicates as given in the Floyd model.

2. A BRIEF SURVEY OF TECHNIQUES

Since the late 1960s and the developments of Floyd and Hoare, several models for program verification have been developed. We briefly summarize them here and will describe them in greater detail in later chapters.

2.1. Axiomatic Verification

This is the technique previously described by Floyd and Hoare. We can give a series of axioms describing the behavior of each statement type, and prove, using formal mathematical logic, that the program has the desired pre- and postconditions.

For example, given the two propositions: $\{P\}S_1\{Q\}$ and $\{Q\}S_2\{R\}$, we can infer that if we execute both statements, we get: $\{P\}S_1;S_2\{R\}$. Similarly, if we can prove the following proposition: $R \Rightarrow T$, we can then state: $\{P\}S_1;S_2\{T\}$. Continuing in this manner, we build up a proof for the entire program. We can extend this model to include data declarations, arrays, and procedure invocation.

Dijkstra [10] developed a model similar to Hoare's axiomatic model which he called *predicate transforms*, based upon two notions: (a) the *weakest precondition* of a statement; and (b) guarded commands and nondeterministic execution.

• **Weakest precondition**
The weakest precondition to a given statement S and postcondition Q is the largest initial set of states for which S terminates and Q is true. A weakest precondition is also called a *predicate transformer* since we are able, in many cases, to derive from a given postcondition the precondition that fulfills this definition. If P is the weakest precondition, we write $P = wp(S, Q)$. For example, in order to have a variable x equal to 3 after the assignment statement "$x := x+1$", the program state prior to this statement must have x equal to 2. Therefore, $wp(x := x + 1, x = 3)$ is the ***set*** of all program states such that x has value 2.

As observed in Gries [23], we can prove several theorems from this basic definition:

$$
\begin{array}{l}
wp(S, false) \ = \ false \\
if\ P \Rightarrow Q\ then\ wp(S,P) \Rightarrow wp(S,Q) \\
wp(S, P \vee Q) \ = \ wp(S,P) \vee wp(S,Q) \\
wp(S, P \wedge Q) \ = \ wp(S,P) \wedge wp(S,Q)
\end{array}
$$

• **Guarded commands**
Dijkstra realized that many algorithms were easier to write nondeterministically, that is, "if this condition is true then do this result." A program is simply a collection of these operations, and whenever any of these conditions apply, the operation occurs. This concept is also the basis for Prolog execution.

The basic concept is the guard ($[\!]$). The statement:

$$\textbf{if}\ a_1 \rightarrow b_1\ [\!]\ \cdots\ [\!]\ a_n \rightarrow b_n\ \textbf{fi}$$

means to execute any b_i if the corresponding a_i is true.

From these, we can build up a model very similar to the Hoare axioms, as we will later show in Chapter 4.

2.2. Algebraic Specification

The use of modularization, datatypes, and object oriented programming have led to a further model called *algebraic specifications*, as developed by Guttag. In this model we are more concerned about the behavior of objects defined by programs rather than the details of their implementation. For example, "What defines a data structure called a *stack*?" Any such description invariably includes the example of taking trays off and on a pile of such trays in a cafeteria and moving your hands up

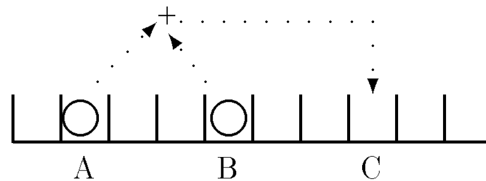


Figure 1.2. Memory model of storage

and down. More formally, we are saying that a $push$ of a tray onto the stack is the inverse of the pop operation taking it off the stack. Or in other words, if S is a stack and x is an object, $push$ and pop obey the relationship

$$pop(push(S, x)) \;=\; S$$

That is, if you add x to stack S and then pop it off, you get S back again.

By adding a few additional relationships, we can formally define how a stack must behave. We do not care how it is implemented as long as the operations of $push$ and pop obey these relationships. That is, these relationships now form a *specification* for a stack.

2.3. Storage

Before discussing the remaining techniques, a slight digression concerning assignment and memory storage is in order. Consider the following statement: $C := A + B$. This statement contains two classes of components: (a) a set of data objects $\{A, B, C\}$ and (b) operators $\{+, :=\}$.

In most common languages like FORTRAN, Pascal, C, or Ada, operators are fixed and changes are made to data. Storage is viewed as a sequence of cells containing “memory objects.” The various operators access these memory objects, change them, (e.g., accessing A and B and adding them together) and placing the resulting object in the location for C (Figure 1.2). An ordered collection of colored marbles is probably the mental image most people have of memory.

On the other hand, we can view data as fixed and manipulate the operator that views the objects (Figure 1.3). In this case, we model an operator as a lens that allows us to see the corrected data as modified

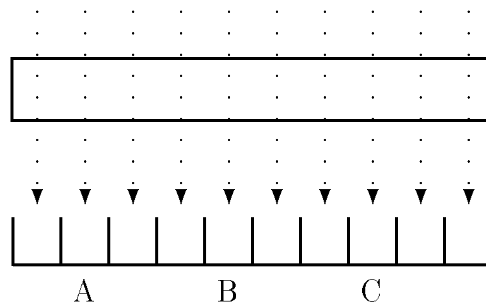


Figure 1.3. Applicative model of storage

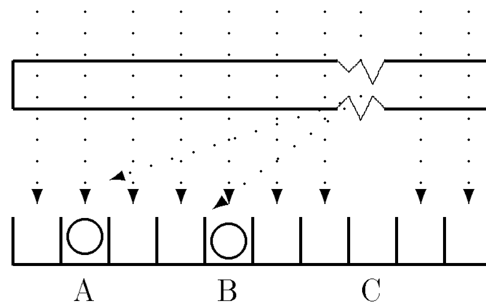


Figure 1.4. Applicative assignment

by the given statement. We call this the applicative view of storage.

In this case, memory is simply a function that returns a value for each location. Execution of an assignment statement simply modifies the accessing function (Figure 1.4). In other words, the modified function is simply the composition of the original function and the new statement.

Pure applicative programming is the basis for languages like LISP; however, most LISP implementations do include memory storage features. As we shall see, this model is also the basis for the denotational and functional models to be next described and is also the basis of the array assignment axiom for the Hoare axiomatic system.

2.4. Functional Correctness

A program can be considered as a function from some input domain to some output domain. If we can also represent a specification as a function, we simply have to show that they are equivalent functions.

Using the box notation of Mills [45], if p is a program, then $\boxed{p}$ is defined to be the function that the program produces. If f is the specification of this program, then verification means showing the equivalence of $f = \boxed{p}$. While this in general is an undecidable property, we can develop conditions of certain programs where we can show this, that is, are there cases where we can compare the expected behavior of the program with the actual behavior?

In particular, if a program p is a sequence of statements $s_1, \cdots, s_n$, then $\boxed{p}$ is just the functional composition of each individual statement function $\boxed{s_1} \circ \cdots \circ \boxed{s_n}$.

We later give several axioms for deriving statement functions from programming language statements and present techniques for proving this equivalence.

2.5. Operational Semantics

The final model of verification we shall discuss is the operational model. In this case, model the program using some more abstract interpreter, show that the program and the abstracted program have equivalent properties, and "execute" the program in the abstract model. Whatever effects the abstracted program show will be reflected in the concrete program.

The first such model based upon this technique was the Vienna Definition Language (VDL).[2] This model extended the parse tree of a program into a "tree interpreter." (See Figure 1.5). While the parse tree (component s_tree) was a static component of this model, some components like the program store (i.e., data storage component s_data) and internal control ($s_control$) were dynamic and changed over time. Other components, like the library (i.e., "microprogrammed" semantic definition of each statement type in $s_library$), were also static. The semantic definition of the language was the set of interpreter routines built into the

[2]Not to be confused with the Vienna Development Method (VDM) to be described later.

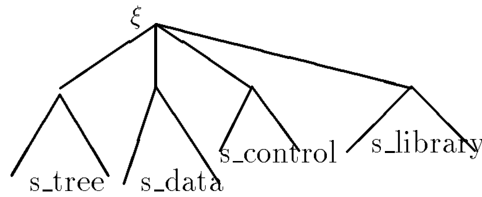


Figure 1.5. Vienna Definition Language Model

library routines. A LISP-like notation was used to define the semantic routines.

VDL was used briefly in the 1970s, and was used as the basis for the standardized definition for the language PL/I, but is largely obsolete today. The major problem is that "debugging" the interpreter routines in the control aspect of the model is not very different from debugging programming language statements. While the model did present the program's semantics at a higher more abstract level, the level was not deemed high enough and the information obtained not strong enough to warrant its use.

A second operational model is still very much in use and has had major impact on the other models and upon practical specification systems. That is the technique of *denotational semantics* developed by Scott and Strachey. In this case, similar to the functional model of Mills, we view a program as function from one domain to another.

A fundamental idea is that we view "memory" as simply a function from a set of identifiers to a set of values. Thus the *state* of a computation at any time is simply a function m_i with the functional signature $id \rightarrow value$ meaning for each $j \in id, m_i(j) = k,\ k \in value$. The "execution" of statement s_{i+1} simply means we transform function m_i into function m_{i+1} by the composition with the function for statement s_{i+1}. (Note the similarity with the Mills functional approach above.)

We give a formal description of the functionality that each statement has on the underlying program store to produce a new program store, and can use this to model complete languages.

This model has had an effect on other models. The basic functional approach is the basis for the Mills functional model. We will later see that the axiomatic array axiom is just the denotational semantics

assignment property.

3. SEMANTICS VERSUS SPECIFICATIONS

In the discussion so far, the terms “semantics,” “verification,” and “specification” have been mostly intermixed with no clear distinction among them. They are highly interrelated, generally describe similar properties, and their order above generally follows the historical development of the field.

Initially (duriing the late 1950s through mid-1960s), the problem was to describe the semantics of a programming language, using techniques like attribute grammars and VDL-like operational semantics. The thrust through the 1970s was the proving of the (functional) correctness of a program, or program verification. Today, we are interested in building valid systems, that is, programs that meet their specifications.

Using the functional correctness box notation, we can describe some of today’s issues. Let s be a specification and p be a program.

• Does $s = \boxed{p}$? This is the verification problem.

• Given s, build program p such that $s = \boxed{p}$. This is the program design problem.

• Given p, find s such that $s = \boxed{p}$. This is what we call *reverse engineering* today. It is an important component in the interest in software reuse. Given a program module, what function does it implement? If we can answer this, semiautomatically, we can more easily reuse the module and reduce development costs.

While all these problems are different, the techniques to address them are the same. Therefore, a study of verification issues has major impact on all of the semantic, verification, specification, and reverse-engineering problems.

4. LIMITATIONS OF FORMAL SYSTEMS

Although research on formal methods is a worldwide activity, cultural differences have emerged. Currently, the general view in the United States is that verification is a mechanism for proving the equivalence between a specification and a program. Since real programs implemented on real computers using real compilers have numerous limitations, the proofs are necessarily hard and verification has made little impact in industry.

On the other hand, the European view is that verification is a mechanism for showing the equivalence between a specification and a design. Since a software design is somewhat removed from the actual implementation, verification is easier, although one still has the remaining problem of showing that the design does agree with the implementation. Because of this, verification is generally more prevalent in European development activities than in the U.S.

Now that you are "sold" on the value of such formal models, we must put these techniques in perspective. Hall [28] listed seven "myths" of formal systems. It is important to understand these concepts as part of learning about the techniques themselves.

1. *Formal methods can guarantee that software is perfect.* As we have shown, all the formal techniques rely on abstracting a program into an abstract specification that closely approximates reality. However, this formal specification is rarely exact, so the resulting program only approximates what you specify. If done well, then this approximation is close enough. However, even simple propositions like "$x + 1 > x$ for integer x" fail with real machines with fixed word size and limited range of integer values.

In addition, we cannot forget that mathematical proofs may have errors in them. Formal proofs certainly help, but are no guarantee of perfection.

2. *They work by proving the programs are correct.* As stated at the beginning, we are interested in more than just functional correctness. Cost, development time, performance, security, and safety are all properties that are part of a complete specification.

3. *Only highly critical systems benefit from their use.* It has been shown that almost any large system will benefit from using formal techniques.

The *cleanroom* is a technique to informally use functional correctness on large software projects. It has been used at IBM, NASA Godddard Space Flight Center and at the University of Maryland (albeit with small student projects in that case). In all instances, reliability was higher, and development effort was sometimes dramatically lower.

4. *They involve complex mathematics.* These techniques involve *precision*, not complex mathematics. There is nothing in this book that a well-informed college undergraduate should not be able to understand. Precision takes much of the ambiguity out of a 300-page informal English specification that is more typical in industry today.

5. *They increase the cost of development.* They do increase the cost of program design, since one must develop the abstract model of the specification more explicitly that is usually done. For managers impatient with programmers not writing any "code," this certainly looks like an increase in costs. But as numerous studies have shown (e.g., the cleanroom studies above), the project's overall costs are lower since the time-consuming and expensive testing phases are dramatically reduced.

6. *They are incomprehensible to clients.* It is our belief that a 300 page English text of ambiguous statements is more incomprehensible. In addition, the role of formal methods is to help the technical staff understand what they are to build. One must still translate this into a description for the eventual user of the system.

7. *Nobody uses them for real projects.* This is simply not true. Several companies depend upon formal methods. Also, various modifications to some of these techniques are used in scattered projects throughout industry. It is unfortunately true, however, that few projects use such techniques, which results in most of the problems everyone is writing about.

The following section will give the notation we will use throughout this book, and the following chapters will present these techniques outlined here in more detail.

5. PROPOSITIONAL CALCULUS

Much of the theory of program verification is built upon mathematical logic. This section is a brief review of some of these concepts.

The *propositional calculus* is a simple language for making statements (*propositions*) that may be viewed as *true* or *false*. Strings in this language are called *formulae*.

The syntax for the propositional calculus is

- set of symbols

variables:A, B, P, Q, R, ...

constants:T, F

connectives:$\wedge$, $\vee$, $\neg$, $\Rightarrow$

parentheses:(,)

- rules for forming legal strings of these symbols, e.g. a formula is defined as

a variable

a constant

a string: if A and B are formulae, so are $A \wedge B$, $A \vee B$, $\neg A$ and $A \Rightarrow B$

We must now define the semantics of propositional calculus. An *interpretation* is a way of understanding a formula, encoding some information. *Truth values* are assigned to formulae as follows:

- T has value *true*
- F has value *false*
- Variables can take on either *true* or *false*
- (A $\wedge$ B) is *true* if A is *true* **and** B is *true*, is *false* otherwise
- (A$\vee$B) is *true* if A is *true* **or** B is *true*, is *false* otherwise
- $\neg$A is *true* if A if *false*, is *false* if A is *true*
- (A$\Rightarrow$B) is *true* if A is *false* **or** B is *true*

Definitions:

truth assignment: A *truth assignment* is a mapping of the variables within a formula into the value *true* or *false*.

satisfiable: A formula is *satisfiable* if there exists some truth assignment under which the formula has truth value *true*.

valid: A formula is a *tautology*, or *valid*, if it has truth value *true* under all possible truth assignments.

unsatisfiable: A formula is *unsatisfiable* if it has the truth value *false* for all possible truth assignments.

decidable: Propositional logic is *decidable*: there exists a procedure to determine, for any formula, if it is satisfiable (and hence valid) or not, e.g., *truth tables*.

5.1. Truth Tables

We can build a *truth table* by first assigning all possible combinations of truth values to variables, and then determining the truth value of each subformula under each truth assignment. For instance, consider the formula $\neg A \lor (A \lor B)$:

A	B	¬A	(A∨B)	¬A∨(A ∨ B)
T	T	F	T	T
T	F	F	T	T
F	T	T	T	T
F	F	T	F	T

The given formula is *valid* because all rows of the truth table show it as *true*. Generally, using truth tables are NP-complete procedures: The number of steps in the decision process is exponentially related to the number n of variables in a formula (2^n).

5.2. Inference Rules

Inference systems allow determination of *tautology* or *unsatisfiability*, but say nothing about formulae in between. An inference system consists of:

- axioms (set of valid formulae)
- inference rules (create new formulae from existing ones)

Two formulae are equivalent if every truth assignment causes their truth values to be equal. Rules of inference are *truth preserving* in that

they transform a formula (conjunction of members of a set of formulae) into an *equivalent* formula. Starting with axioms (which are valid), then each subsequent formula derived with inference rules is also valid. This is *soundness*: If only valid formulae can be derived, the inference system is sound.

We say an inference system is *complete* if it can derive *all* formulae which are valid. For example:

• Axioms

1. $P \Rightarrow (Q \Rightarrow P)$
2. $(S \Rightarrow (P \Rightarrow Q)) \Rightarrow ((S \Rightarrow P) \Rightarrow (S \Rightarrow Q))$
3. $\neg (\neg P) \Rightarrow P$

• Inference rules

1. From $(A \Rightarrow B)$ and A, conclude B (*modus ponens*)
2. From A, may get A' by substituting a variable y for variable x throughout A.

This system is *sound* and *complete*.

Axioms are proven valid with truth tables. Another formula may then be proven valid by discovering (manufacturing) a sequence of rules to apply to the axioms to get the formula.

In some cases, we want a result if some previous assumption is true. For example, if p is assumed true, then can we infer q? In such a situation we will use the notation $p \vdash q$. In fact to be quite formal, given our inference system, any result q that we can infer from our axioms is properly written as $true \vdash q$, or more simply $\vdash q$.

In most situations, $p \vdash q$ and $p \Rightarrow q$ behave quite similarly, and we will avoid this added level of complexity in this book. But they are quite different. $\Rightarrow$ is a binary operator applied to predicates, while $\vdash$ is an inference rule. For the most part we can ignore the difference, however, even in our treatment here, we need to differentiate between the two concepts. For example, we will see that the recursive procedure call rule in Chapter 2 and VDM in Chapter 7 both need to refer to $\vdash$.

If we have a rule of inference $(p \Rightarrow q) \vdash (r \wedge s)$, then we can write it as:

$$\frac{p \Rightarrow q}{r \wedge s}$$

We will interpret this to mean that if we can show the formula above the ‘bar’ ($p \Rightarrow q$) to be valid (either as an assumption or as a previously derived formula), then we can derive the formula below the ‘bar’ ($r \wedge s$) as valid. We shall use this notation repeatedly to give our inference rules in this book.

If the relationship works both ways (i.e., $p \Rightarrow q$ and $q \Rightarrow p$), then we will use the notation:

$$\frac{p}{\overline{q}}$$

Therefore, we can write the two inference rules given above as:

$$\begin{array}{ll} \textbf{Modus ponens}: & \dfrac{A \Rightarrow B, A}{B} \\ \textbf{Substitution of x by y}: & \dfrac{A}{A_y^x} \end{array}$$

5.3. Functions

We shall use the notation $x \stackrel{\triangle}{=} y$ to mean that function x is defined by expression y.

The expression $x \equiv y$ shall mean that logical expressions x and y have the same truth value. We shall also use this notation to mean that program x is defined to be the sequence of statements described by y.

It is often desirable to specify functions as a conditional. For example, the **maximum** function can be specified by the program:

$$\begin{array}{lll} max & \equiv & \textbf{if } a > b \textbf{ then } max := a \\ & & \quad\quad \textbf{else } max := b \end{array}$$

Writing this as a function, we can state:

$$max(a,b) \stackrel{\triangle}{=} if\ a > b\ then\ max := a\ else\ max := b$$

and can then write $max(a,b)$ or $max(x,y)$.

We shall use a notation patterned after the LISP *cond* construct in this book to represent such functions. If b_i is a boolean condition and c_i are functions, then a conditional function has the syntax: $(b_1 \rightarrow c_1)|(b_2 \rightarrow c_2)|\ldots|(b_n \rightarrow c_n)$ with the semantics of evaluating each b_i in turn, and setting the value of the conditional to be c_i for the first b_i which is true. If all b_i are false, then the statement is undefined. If b_n is the default case (i.e., the expression **true**), then it can be omitted with the last term becoming (c_n). The *Identity* function is written as $()$.

Therefore, we will generally write functions, like max, using the following syntax:

$$max(a,b) \stackrel{\triangle}{=} (a > b \ \rightarrow \ max := a) \mid (max := b)$$

It is important to differentiate between total and partial functions. The functions $(a \rightarrow b)|\neg a \rightarrow c)$ and $(a \rightarrow b)|(c)$ are both total since they are defined for every possible input state (i.e., either a or $\neg a$ must be true). However, the function $(a \rightarrow b)$ is only partial since it is undefined if a is false. While seemingly an anomalous condition, this occurs frequently in programming (i.e., an infinite **while** loop that does not terminate.

One last piece of functional notation. Throughout this book, we will often want to refer to the computation performed by a program as some expression involving the variables in the program. For example, if we have the procedure $mult$ that multiplies two arguments together, then by writing $y := mult(a,b)$, we would like to say that $y = ab$.

However, the execution of $mult$ may change the final values of a and b, so the equality $y = ab$ may actually be false after the execution of procedure $mult$. This is the inherent difference between functional mathematics and programming. We need a notation to refer to the initial values of a and b so that we can refer to the final result. We will use $\leftharpoondown$ as this symbol. $\overleftharpoondown{a}$ means the original value that a had before execution of the procedure began. Thus for $y := mult(a,b)$, we can state $y = \overleftharpoondown{a}\,\overleftharpoondown{b}$.

5.4. Predicate Calculus

We may now describe the predicate calculus, a more powerful language for making statements that can be understood to be *true* or *false*. Its

syntax is:

- predicate symbols $(P, Q, R, ..., <, >, =, ...)$
- variables $(x, y, z, ...)$
- function names $(f, g, h, ...)$
- constants (0-argument functions) $(a, b, c, ...)$
- quantifiers ($\exists$, $\forall$)
- logical connectives ($\wedge$, $\vee$, $\neg$, $\Rightarrow$)
- logical constants (T, F)

An *atomic formula* is either:

- a logical constant (T or F)
- any predicate $P_n(t_1, \ldots, t_n)$ where t_i is a *term*, formed from variables, constants, or function names, as one would expect.

For example

$$P(x, f(T, g(x, y)))$$

is a predicate expression consisting of constant T, variables x and y, and functions f, g, and P.

A *well-formed formula (wff)* is defined as:

- any atomic formula
- if α is a wff, so are $\neg\alpha$, $\forall x\alpha$, $\exists x\alpha$
- if α and β are wffs, so are ($\alpha \vee \beta$), ($\alpha \wedge \beta$), ($\alpha \Rightarrow \beta$)
- liberal use of parentheses throughout

For example

$$(\neg(\forall x P(x)) \Rightarrow (\exists y \neg P(y)))$$

is a *valid* wff that is a tautology, and it contains no function names.

Functions give a means of calculating the "name" of an item to be discussed, as opposed to simply specifying the name as a constant. The

added power is akin to using arrays with variables as subscripts, as opposed to using only constants as subscripts.

We say a variable in a wff is *bound* if it is in the scope of some quantifier (shown by parentheses unless obvious). A variable is *free* if it is not bound. For example: in $\forall x \exists y (Q(x, f(y, z)))$ x and y are bound, and z is free.

In predicate calculus, an interpretation I is analogous to truth assignment in propositional calculus.

- set of elements D (the *domain*)
- each n-place function name f_n is assigned a total function $\langle f_n \rangle$: $D^n \rightarrow D$
- each constant, each free variable is mapped into D (i.e., given a value)
- each n-place predicate name P_n is assigned a total function $\langle P_n \rangle$: $D^n \rightarrow \{$ T , F $\}$.

If W is a wff, and I is an interpretation, then (W, I) denotes the truth value of the statement that the two together represent.

1. W is *valid* iff (W,I) is *true* for every I.
2. W is *satisfiable* iff (W,I) is *true* for some I.
3. W is *unsatisfiable* iff (W,I) is *true* for no I (is *false* for all I).

However, the predicate calculus is not decidable. There exist W such that it cannot be determined whether there is an interpretation I such that (W, I) is valid. There are complete and sound inference systems for all three of the following:

1. *valid:* can prove wff W by deriving W from the axioms.
2. *unsatisfiable:* can prove wff W by deriving the *negation* of W from the axioms (assuming ¬ (A ∧¬ A) is an axiom of the system).
3. *satisfiable:* W is satisfiable if can say nothing about W, since neither derivation (A and ¬ A) will terminate.

5.5. Quantifiers

The propositional calculus is a *zero-order* language containing only constants without functions. The predicate calculus is a *first-order* language containing functions that compute values. There exist *higher-order* languages (second, third, etc.), having functions that compute (functions that compute ... (functions that compute values) ...).

Example:

$$\exists x \forall y (x < y \vee x = y)$$

This is common notation, though the relation could be written more accurately as $\vee(<(x, y), =(x, y))$.

However, is that statement *True* or *false* (i.e., a tautology)? That depends on the interpretation:

- If we let D be the set of natural numbers, with normal arithmetic the this wff is *true*.
- If let D be the set of all integers, then this wff is *false*.

Often we need to change the identifiers used in a wff. Informally, consider this: The "meaning" is unchanged by substituting k for i:

```
var j : integer;
function f(i : integer) : integer;
begin f := i + j end
```

becoming

```
var j : integer;
function f(k : integer) : integer;
begin f := k + j end
```

However the "meaning" is changed by substituting j for i, where the new text becomes

$$\textbf{var } j : \textbf{integer};$$
$$\textbf{function } f(j : \textbf{integer}) : \textbf{integer};$$
$$\textbf{begin } f := j + j \textbf{ end}$$

because j is already bound in the scope it was substituted into. We can only replace *free* occurrences of an identifier by another free identifier (or expression). To preserve "meaning" of wff under substitution, must not alter the free/bound status of any of its identifiers. For example:

$$\exists x(x < y)$$

We can substitute $f(z, w)$ for y to get $\exists x(x < f(z, w))$. We cannot substitute $f(z, x)$ for y, since x is bound already.

More formally, A term t is *substitutable* for x in a wff A iff for each (free) variable y in t, x does not occur free in a subterm of A of the form $\exists y\ F$ (or $\forall y\ F$). The new wff obtained by this substitution is denoted A^x_t.

5.6. Example Inference System

Logical Axioms:

1. $\neg F \vee F$
2. $x = x$
3. $\neg(x_1 = y_1) \vee \ldots \vee \neg(x_n = y_n) \vee f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$
4. $\neg(x_1 = y_1) \vee \ldots \vee \neg(x_n = y_n) \vee \neg P(x_1, \ldots, x_n) \vee P(y_1, \ldots, y_n)$
5. $\neg A^x_t \vee \exists x\ A$ (provided t is substitutable for x in A)

Rules of Inference:

1. $\dfrac{A}{B \vee A}$

2. $\dfrac{A \vee (B \vee C)}{(A \vee B) \vee C}$

3. $\dfrac{A \vee B, \neg A \vee C}{B \vee C}$

4. $\dfrac{\neg A \vee B}{\neg \exists x(A \vee B)\ provided\ x\ is\ not\ free\ in\ B}$

5. $\dfrac{A, B}{A}$

Often we "beef up" an inference system to make proofs about certain entities easier (less lengthy) by adding more axioms that are tailored to talk about the objects we are interested in. For example, we can assume the following arithmetic axioms:

1. $\forall x(x \neq \mathbf{0} \Rightarrow \exists y(y + \mathbf{1} = x))$
2. $\forall x \forall y(x + \mathbf{1} = y + \mathbf{1} \Rightarrow x = y)$
3. $\forall x \forall y(x < y + \mathbf{1} \Rightarrow (x < y) \vee (x = y))$
4. $\forall x \neg(x < \mathbf{0})$
5. $\forall x \forall y(x < y) \vee (x = y) \vee (x > y)$
6. $\forall x(x + \mathbf{0} = x)$
7. $\forall x \forall y(x + (y + \mathbf{1}) = (x + y) + \mathbf{1})$
8. $\forall x(x \times \mathbf{0} = \mathbf{0})$
9. $\forall x \forall y(x(y + \mathbf{1}) = xy + x)$
10. $\forall x \forall y \forall z(x - y = z \Rightarrow x = y + z)$

This restricts our focus to the objects the new axioms discuss (here, positive integers).

Adding axioms like this can lead to incompleteness of the inference system, since these new "axioms" are not necessarily tautologies, but instead are true only under one (or a few) interpretations.

Several theorems, which can all be verified by appropriate trace tables, involving conditional statements will be used in this book. Their inference rules are given as follows:

1. $$\frac{(a \rightarrow b)|(\neg a \rightarrow c)}{(a \rightarrow b)|(c)}$$

2. $$\frac{(a \rightarrow (b \rightarrow c))}{(a \wedge b \rightarrow c)}$$

3. $$\frac{(a \rightarrow c)|(b \rightarrow c)}{(a \vee b \rightarrow c)}$$

4. $$\frac{(a \rightarrow (b \vee c))}{(a \rightarrow b)|(a \rightarrow c)}$$

5. $$\frac{A \Rightarrow (B \vee C)}{(A \Rightarrow B) \vee (A \Rightarrow C)}$$

6. $$\frac{A \Rightarrow (B \Rightarrow C)}{(A \wedge B) \Rightarrow C}$$

6. EXERCISES

Problem 1. Give as complete a list of attributes that you can to describe a software system.

Problem 2. Define the binary operator $\Leftrightarrow$ as:

$$A \Leftrightarrow B \triangleq (A \Rightarrow B) \wedge (B \Rightarrow A)$$

Show that DeMorgan's laws are valid:

$$(A \wedge B) \Leftrightarrow \neg(\neg A \vee \neg B)$$
$$(A \vee B) \Leftrightarrow \neg(\neg A \wedge \neg B)$$

7. SUGGESTED READINGS

Jon Bentley's collection of writings

- J. Bentley, *Programming Pearls*, Addison-Wesley, Reading, MA, 1986.

is an excellent source of ideas for refining practical programming and problem-solving skills. Chapter Four in particular, "Writing Correct Programs," gives a nice introduction to the types of proof methods which will be discussed in this book. Likewise the book,

- D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.

is a magnificent textbook describing the process of program specification and development using predicate transforms.

Background material for this book consists of the following:

Understanding of context free languages, parsing, and general language structure. Compiler books like the following are helpful:

- A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.
- C. N. Fischer and R. J. LeBlanc, *Crafting a compiler*, Benjamin Cummings, Menlo Park, CA, 1988.

Understanding the structure of programming languages and their execution model in typical von Neumann architectures:

• T. W. Pratt, *Programming Languages: Design and Implementation*, Second Edition, Prentice-Hall, Englewood Cliffs, NJ, 1984.

• R. Sethi, *Programming Languages: Concepts and Constructs*, Addison-Wesley, Reading MA, 1989.

Some references for specification systems include:

• A. Hall, "Seven Myths of Formal Methods," *IEEE Software,* Vol. 7, No. 5, September 1990, pp. 11-19.

• J. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer,* Vol. 23, No. 9, September 1990, pp. 8-24.

• J. Woodcock and M. Loomes, *Software Engineering Mathematics*, Addison-Wesley, Reading, MA, 1988. (This is a theory using a *Z-like notation*.)

• D. Craigen (Ed.), *Formal methods for trustworthy computer systems*, Springer-Verlag, New York, 1990.

Chapter 2

The Axiomatic Approach

An axiomatic approach requires both axioms and rules of inference. For our purposes, we will accept the axioms of arithmetic which characterize the integers and the usual rules of logical inference as discussed earlier (Chapter 1, Section 5). However, we do not seek to model just the 'world of integers' but also the 'world of computation,' and hence we need additional axioms and inference rules corresponding to the constructs we intend to use as our programming language.

1. PROGRAMMING LANGUAGE AXIOMS

In order to build programs into our inference system, we must be able to model a computation as a logical predicate. We adopt the notation $\{P\}S\{Q\}$, where P and Q are assertions about the state of the computation and S is a statement. This expression is interpreted as "If P, called the ***precondition***, is true before executing S and S terminates normally, then Q, called the ***postcondition***, will be true." This will allow us to model a simple "Algol-like" programming language. We do this by adding the inference rules of composition and consequence:

$$\textbf{Composition :} \quad \frac{\{P\}S_1\{Q\},\ \{Q\}S_2\{R\}}{\{P\}S_1;S_2\{R\}}$$

$$\textbf{Consequence}_1: \quad \frac{\{P\}S\{R\},\ \ R \Rightarrow Q}{\{P\}S\{Q\}}$$

$$\textbf{Consequence}_2: \quad \frac{P \Rightarrow R,\ \ \{R\}S\{Q\}}{\{P\}S\{Q\}}$$

If we are to have confidence in a program which has been proven to be partially correct by this axiomatic method, then it is important for us to believe in the axioms and inference rules accepted up front. A good way to find motivation for the choice of these inference rules is to examine the Floyd-style flow chart associated with each drawing.

The rule of composition is the basic mechanism which allows us to "glue" together two computations (i.e., two statements). If the postcondition of one statement is the same as the precondition of the following statement, then the execution of both of them proceeds as expected.

The rules of consequence build in our predicate logic rules of inference. We can use these to permit us to use alternative pre- and postconditions to any statement as long as we can derive them via our mathematical system.

Given this logical model, we need to build in the semantics of a given programming language. We will use the following BNF to describe a simple Algol-like language:

$$\begin{aligned} < stmt > ::= \ & < stmt >; < stmt > \\ & |\textbf{if}\ < expr >\ \textbf{then}\ < stmt > \\ & |\textbf{if}\ < expr >\ \textbf{then}\ < stmt >\ \textbf{else}\ < stmt > \\ & |\textbf{while}\ < expr >\ \textbf{do}\ < stmt > \\ & | < id > := < expr > \end{aligned}$$

where $< id >$ and $< expr >$ have their usual meanings. Note: The examples will be kept simple, and we will not worry about potential ambiguities such as programs like: $< stmt >; if \ \ < expr > \ \ then \ \ < stmt >; < stmt >$. that is, is this a "statement and an **if**" or "statement, **if**, and statement?" The examples will be clear as to meaning.

The basic approach is a "backwards verification" method. For example, for the assignment, we will define the axiom so that given the result of an assignment statement, what must have been true prior to the execution of the statement in order for the result to occur? Or in other words, given the postcondition to an assignment statement, what is its precondition?

We accept the assignment axiom ***schema***

$$\{P^x_y\}x := y\{P\}$$

Here P^x_y represents the expression P where all free occurrences of x have been replaced by y. For example,

$$\{z = x + y - 3\}x := x + y + 1\{z = x - 4\}$$

represents the effect of replacement of x by $x + y + 1$.

In addition to the above axiom schema, we accept the following rules of inference for the **if** and **while** statements:

$$\textbf{Conditional}_1: \quad \frac{\{P \wedge B\}\ S\ \{Q\},\ \ P \wedge \neg B \Rightarrow Q}{\{P\}\textbf{if}\ B\ \textbf{then}\ S\{Q\}}$$

$$\textbf{Conditional}_2: \quad \frac{\{P \wedge B\}\ S_1\ \{Q\},\ \ \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

$$\textbf{While}: \quad \frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\textbf{while}\ B\ \textbf{do}\ S\ \{P \wedge \neg B\}}$$

The **if** statement rules are fairly obvious. Each of the two possible rules simply traces the execution of the corresponding $if - then$ and $if - then - else$ statement, and given precondition P, determines under what conditions postcondition Q will be true.

The axiom for the **while** statement is perhaps as major a contribution by Hoare [29] as the entire model of verification that he also developed. The basic loop has the structure:

$$\begin{array}{l}\textbf{while}\ something - is - true \\ \qquad\qquad\qquad \textbf{do}\ something\end{array}$$

There must be some property that remains true throughout the execution of the loop. So if we call the body of the loop S, then we will call the property that must remain true P, and we have the result: “If P is

true and we execute the loop, then P will remain true." The condition 'and we execute the loop' is just the predicate on the **while** statement B. This leads to the condition that: $\{P \wedge B\}S\{P\}$ as the antecedent property on the **while** axiom. So if this antecedent is true, then after the loop 'terminates,' we still will have P true, and since the loop terminated, B must now be false, hence the axiom as given. Note we have not proven that the loop *does* terminate. That must be shown and is outside of this axiom system (see Section 1.2).

The property that remains true within the loop is called the ***invariant*** and it is at the heart of developing axiomatic proofs for programs. While there is no algorithmic method for developing invariants, we later give some ideas on how to proceed.

1.1. Example: Integer Division

Consider the following example of a Hoare-style proof of partial correctness of a program to compute the remainder and quotient from a division of two integers, x and y:

$$
\begin{array}{l}
\mathbf{PROGRAM} \equiv \{ \\
\quad q := 0; \\
\quad r := x; \\
\quad \mathbf{while}\ y \leq r\ \mathbf{do} \\
\quad\quad r := r - y; \\
\quad\quad q := 1 + q\}
\end{array}
$$

Our input condition is $\{x \geq 0 \wedge y > 0\}$, and our desired output condition is $\{\neg(y \leq r) \wedge x = r + yq\}$. Note how the output condition characterizes the desired relationship between values in order for r and q to represent the remainder and quotient, respectively. Also note that the "input" to this program is assumed to be whatever is contained in the variables upon starting execution of this program.

If our output condition is to be true, it must be so as a result of application of a **while** rule of inference, corresponding to the execution of the only **while** loop in our program. In order to apply the rule, we must identify the P and B in the rule's antecedent "$P \wedge B$." This immediately suggests that our B must be $B \equiv r \geq y$, and that our P (which is referred to as the ***invariant***) must therefore be $P \equiv x = qy + r\ \wedge\ 0 \leq r$. Hence the inference rule which could be applied would be

$$\frac{\{P \wedge r \geq y\} r := r - y; q := q + \mathbf{1}\{P\}}{\{P\}\ \textbf{while}\ r \geq y\ \textbf{do}\ r := r - y; q := q + \mathbf{1}\{\ P \wedge \neg B\}}$$

Continuing to reason backwards from our target assertion, we try to show how the antecedent to this **while** rule would be derived, that is, we must show that the assertion which characterizes the program state just before execution of the **while** loop is maintained by continued execution of the while loop. In short, this inductive step is called "showing that the invariant is maintained." We start with our invariant P and work backwards through the body of our while loop to see what conditions before the loop will imply P: First, our assignment axiom schema can be brought in, yielding

$$\{x = (q + \mathbf{1})y + r \wedge \mathbf{0} \leq r\} q := q + \mathbf{1}\{P\}$$

and an additional use of the axiom shows us

$$\{\underbrace{x = (q + \mathbf{1})y + (r - y) \wedge \mathbf{0} \leq r - y}_{x = qy + r \wedge \mathbf{0} \leq r - y}\} r := r - y$$
$$\{x = (q + \mathbf{1})y + r \wedge \mathbf{0} \leq r\}$$

By use of our rule of composition applied to the above two steps, we get

$$\frac{\begin{array}{l}\{x = qy + r \wedge \mathbf{0} \leq r - y\} r := r - y\{x = (q + \mathbf{1})y + r \wedge \mathbf{0} \leq r\}, \\ \quad \{x = (q + \mathbf{1})y + r \wedge \mathbf{0} \leq r\} q := q + \mathbf{1}\{P\}\end{array}}{\{x = qy + r \wedge \mathbf{0} \leq r - y\} r := r - y; q := q + \mathbf{1}\{P\}}$$

Now by using a rule of consequence, we can show that the invariant is indeed maintained:

$$\frac{\begin{array}{l}\{x = qy + r \wedge \mathbf{0} \leq r - y\} r := r - y; q := q + \mathbf{1}\{P\}, \\ \quad P \wedge r \geq y \Rightarrow x = qy + r \wedge \mathbf{0} \leq r - y\end{array}}{\{P \wedge r \geq y\} r := r - y; q := q + \mathbf{1}\{P\}}$$

Now we must determine whether or not the "initialization" steps in our program will yield us the invariant P. Again working backwards, we utilize our assignment axiom

$$\{x = qy + x \wedge \mathbf{0} \leq x\} r := x\{P\}$$

utilize it again for

$$\{\underbrace{x = 0y + x \wedge 0 \leq x}_{(x=x\wedge 0\leq x)\equiv 0\leq x}\}q := 0\{x = qy + x \wedge 0 \leq x\}$$

and so by using our rule of composition, we may infer

$$\frac{\{0 \leq x\}q := 0\{x = qy + x \wedge 0 \leq x\}, \{x = qy + x \wedge 0 \leq x\}r := x\{P\}}{\{x = x \wedge 0 \leq x\}q := 0; r := x\{P\}}$$

Use composition to add the loop initialization to the loop body:

$$\frac{\begin{array}{l}\{0 \leq x\}q := 0; r := x\{P\},\\ \{P\}\textbf{while}\ r \geq y\ \textbf{do}\ r := r - y; q := q + 1\{P \wedge \neg B\}\end{array}}{\{0 \leq x\}q := 0; r := x;\ \textbf{while}\ r \geq y\ \textbf{do}\ \cdots\{P \wedge \neg B\}}$$

The input assertion implies the antecedent of our initialization code, and so we may use a rule of consequence to replace $0 \leq x$ with the input assertion:

$$\frac{\begin{array}{l}\{0 \leq x\}q := 0; r := x; \textbf{while}\ r \geq y\ \textbf{do} \cdots \{P \wedge \neg B\},\\ \{x \geq 0 \wedge y > 0 \Rightarrow 0 \leq x\}\end{array}}{\{x \geq 0 \wedge y > 0\}q := 0; r := x;\ \textbf{while}\ r \geq y\ \textbf{do} \cdots \{P \wedge \neg B\}}$$

Since $P \wedge \neg B$ is our desired output assertion, we have completed a demonstration of partial correctness for this program.

The program generates two values: q and r. We can use these to have this program model integer division (quotient or $quot$) and the remainder function (rem or mod) as follows:[1]

$$\begin{array}{rcl}(quot(x, y) = q) & \triangleq & (\exists r \mid y > r \geq 0\ \wedge\ x = r + yq)\\ (rem(x, y) = r) & \triangleq & (\exists q \mid y > r \geq 0\ \wedge\ x = r + yq)\end{array}$$

[1] This is left as an exercise.

We will use these results later.

The example presented is organized to give as much insight as possible as to *how* each step is chosen. As with so many proofs, we start with the desired result and "reason back" to find out what we needed to start with. This is fine. However, it is also useful to consider the "straightforward" proof as well, as demonstrated by Hoare [29] in his original CACM paper. The above discussion can be organized into a precise proof of our program's partial correctness as follows:

1.	$\mathbf{true} \Rightarrow\ x = x + y \times 0$	lemma[2]
2.	$\{x = x + y \times 0\}\ r := x\ \{x = r + y \times 0\}$	assignment
3.	$\{x = r + y \times 0\}\ q := 0\ \{x = r + yq\}$	assignment
4.	$\{\mathbf{true}\}\ r := x\ \{x = r + y \times 0\}$	consequence(1,2)
5.	$\{\mathbf{true}\} r := x; q := 0 \{x = r + yq\}$	composition(4,3)
6.	$x = r + yq\ \wedge\ y \leq r \Rightarrow x = (r - y) + y(1 + q)$	lemma[2]
7.	$\{x = (r - y) + y(1 + q)\}\ r := r - y$ $\{x = r + y(1 + q)\}$	assignment
8.	$\{x = r + y(1 + q)\}\ q := 1 + q\ \{x = r + yq\}$	assignment
9.	$\{x = (r - y) + y(1 + q)\}\ r := r - y; q := 1 + q$ $\{x = r + yq\}$	composition(7,8)
10.	$\{x = r + yq\ \wedge\ y \leq r\}\ r := r - y; q := 1 + q$ $\{x = r + yq\}$	consequence(6,9)
11.	$\{x = r + yq\}\ \mathbf{while}\ y \leq r\ \mathbf{do}\ (r := r - y;$ $q := 1 + q)\{\neg y \leq r \wedge x = r + yq\}$	while(10)
12.	$\{\mathbf{true}\}\ \mathbf{PROGRAM}\ \{desired\ output\}$	composition(5,11)

Note how the proof above isn't *quite* the same result as what we originally derived. The key difference is in the precondition: Our reproduction of Hoare's proof shows **true** as the predcondition, effectively stating "you can start in any program state for the following steps to lead to the postcondition." In contrast, our first derivation used the precondition $\{x \geq 0 \wedge y > 0\}$, which is slightly more restrictive, since fewer initial program states satisfy this predicate. In fact, the reason we choose the invariant in our first derivation was probably more difficult to understand as a result of this ("Why add the $0 \leq r$?"). Both proofs are valid, so why make one more difficult? Study the program to see how it would behave in case our first precondition is not true when the program is 'executed.'

[2]These Lemmas should be proven as an exercise.

1.2. Program Termination

We have shown the "partial correctness" of this introductory example, that is, if the program begins execution in a state satisfying the precondition, and if the program terminates, then the postcondition will be true. But how do we know the program actually terminates? We can often show termination by showing that the following two properties hold:

1. Show that there is some property P which is positive within a loop.

2. Show that for each iteration of the loop, P is decremented by a fixed amount. That is $P_i > P_{i+1}$.

If both properties are true, and if the second property causes P to decrement (yet still be positive), then the only way this can be consistent is for the loop to terminate, or else the first property must become false at some point.

Applying this principle to the previous example: The only variables affecting the loop test are y and r. The former does not change through execution, therefore we concentrate our investigation on what happens to the latter, r. Its initial value is x, which we know from the initialization of the program. In the case that y is strictly greater than x on input, then termination is certain, since the body of the **while** loop would never be executed. Otherwise, r starts out greater than or equal to y and the loop begins execution. In the body of the loop, r is decremented by a positive value (we know it is positive by the precondition); in fact, this decrement is unavoidable. Hence, we may infer that in a finite number of iterations of the loop, r will be decremented to where it is no longer the case that $y \leq r$ and therefore the loop will exit. All execution paths have been accounted for, and hence we conclude that the program will indeed terminate when started in a state satisfying the precondition.

1.3. Example: Multiplication

As a second example we will prove the total correctness of the following program which computes the product of A and B:

$$
\begin{array}{ll}
 & \{B \geq \mathbf{0}\} \\
1. & MULT(A,B) \equiv \{ \\
2. & \quad a := A \\
3. & \quad b := B \\
4. & \quad y := \mathbf{0} \\
5. & \quad \textbf{while } b > \mathbf{0} \textbf{ do} \\
6. & \qquad y := y + a \\
7. & \qquad b := b - \mathbf{1}\} \\
 & \{y \;=\; \overleftarrow{A}\overleftarrow{B}\}
\end{array}
$$

Note: Since A and B are not modified in the program, we can state the postcondition simply as: $y = AB$.

The general approach is to work backwards, starting with the postcondition, to show that the precondition yields that postcondition. In this case, the program terminates with a **while** statement (lines 5-7). Therefore we need to determine the invariant for the loop. If I is the invariant and X is the **while** statement conditional, then we have to show the following two properties:

1. $\{I \wedge X\}\ lines\ \mathbf{6} - \mathbf{7}\{I\}$
2. $(I \wedge \neg X) \Rightarrow (y = AB)$

The first property will allow us to build the **while** statement, while the second property shows that we get the multiplication answer that we desire.

In order to develop the invariant I, it is necessary to determine what property does not change within the loop. The goal of the loop is to compute AB, which is a constant. Since the loop is increasing y while decreasing b, combining them gives a clue to our invariant. Since ab is just AB initially, we get a clue that our invariant might be something like:

$$y + ab = AB$$

Since we want $b = \mathbf{0}$ at the end, forcing y to be equal to AB, our invariant becomes:

$$I \stackrel{\triangle}{=} (y + ab = AB)\ \wedge\ (b \geq \mathbf{0})$$

Since X is just $b > 0$, $I \wedge \neg X \Rightarrow (y = AB)$ is shown as follows:

$$\begin{array}{l} I \wedge \neg X \equiv \\ (y + ab = AB) \ \wedge \ (b \geq 0) \ \wedge \ \neg(b > 0) \equiv \\ (y + ab = AB) \ \wedge \ (b \geq 0) \ \wedge \ (b \leq 0) \equiv \\ (y + ab = AB) \ \wedge \ (b = 0) \equiv \\ (y + a0 = AB) \equiv \\ (y = AB) \end{array}$$

In order to show that I is the invariant of the loop:

Line 7. $b := b - 1$

$$\begin{array}{ll} \{y + a(b-1) = AB \wedge (b-1) \geq 0\} b := b - 1 & \\ \{y + ab = AB \wedge b \geq 0\} & \quad Assignment\ axiom \end{array}$$

Line 6. $y := y + a$

$$\begin{array}{ll} \{y + a + a(b-1) = AB \wedge b - 1 \geq 0\} y := y + a & \\ \quad \{y + a(b-1) = AB \wedge b - 1 \geq 0\} & \quad Assign.\ axiom \\ y + ab = AB \Rightarrow y + a + a(b-1) = AB & \quad Theorem \\ \dfrac{\begin{array}{l} y + ab = AB \Rightarrow (y + a) + a(b-1) = AB, \\ \quad \{y + a + a(b-1) = AB \wedge b - 1 \geq 0\} y := y + a \\ \quad \{y + a(b-1) = AB \wedge b - 1 \geq 0\} \end{array}}{\begin{array}{l} \{y + ab = AB \wedge b - 1 \geq 0\} y := y + a \\ \quad \{y + a(b-1) = AB \wedge b - 1 \geq 0\} \end{array}} & \quad Consequence \end{array}$$

Combining lines 6-7 using Rule of Composition.

$$\dfrac{\begin{array}{l} \{y + ab = AB \wedge b - 1 \geq 0\} y := y + a \{y + a(b-1) = AB \wedge b - 1 \geq 0\}, \\ \quad \{y + a(b-1) = AB \wedge b - 1 \geq 0\} b := b - 1 \{y + ab = AB \wedge b \geq 0\} \end{array}}{\{y + ab = AB \wedge b - 1 \geq 0\} y := y + a; b := b - 1 \{y + ab = AB \wedge b \geq 0\}}$$

We get the loop invariant as follows:

$$\begin{array}{l} I \wedge X \equiv \\ (y + ab = AB) \wedge (b \geq 0) \wedge (b > 0) \Rightarrow \\ (y + ab = AB) \wedge (b > 0) \Rightarrow \\ (y + ab = AB) \wedge b - 1 \geq 0 \end{array}$$

By the rule of consequence we get:

$$\{y + ab = AB \wedge b > 0\} y := y + a; b := b - 1 \{y + ab = AB \wedge b \geq 0\}$$

The above is just $\{I \wedge X\} lines\ 6 - 7 \{I\}$. Therefore we can use our **while** axiom:

$$\frac{\{y + ab = AB \wedge b > 0\} y := y + a; b := b - 1 \{y + ab = AB \wedge b \geq 0\}}{\{y + ab = AB \wedge b \geq 0\} \mathbf{while} \cdots \{y + ab = AB \wedge b \geq 0 \wedge \neg b > 0\}}$$

We have already shown that this results in $y = AB$.

We finish up by going backwards through the initial assignment showing that the program's precondition yields this invariant:

$$\begin{array}{rl} \{0 + ab = AB \wedge b \geq 0\} y := 0 \{y + ab = AB \wedge b \geq 0\} & Line\ 4 \\ \{0 + aB = AB \wedge B \geq 0\} b := B \{0 + ab = AB \wedge b \geq 0\} & Line\ 3 \\ \{0 + AB = AB \wedge (B \geq 0)\} a := A \{0 + aB = AB \wedge B \geq 0\} & Line\ 2 \end{array}$$

Since $B \geq 0 \Rightarrow 0 + AB = AB \wedge B \geq 0$ we get by consequence:

$$\{B \geq 0\} a := A \{0 + aB = AB \wedge B \geq 0\}$$

Combining lines 2-4 by the rule of composition, we complete the proof:

$$\{B \geq 0\} \mathbf{MULT}(\mathbf{A}, \mathbf{B}) \{y = AB\}$$

Termination

We have only shown that *if* the program terminates, then it gives us the desired answer. We must also show termination.

1. Let property P be $b > 0$. P is obviously true in loop.

2. The only change to b in loop is $b := b - 1$. So P is decremented in loop.

Since both termination properties are true, the loop must terminate, and we have shown total correctness of the program.

1.4. Another Detailed Example: Fast Exponentiation

Consider the following program to find exponents:

$$
\begin{array}{l}
\{n \geq \mathbf{0} \wedge x \neq \mathbf{0}\} \\
\textbf{PROGRAM} \equiv \{ \\
\quad k := n; \\
\quad y := \mathbf{1}; \\
\quad z := x; \\
\quad \textbf{while } k \neq \mathbf{0} \textbf{ do} \\
\quad\quad \textbf{if } odd(k) \\
\quad\quad\quad \textbf{then } k := k - \mathbf{1}; y := y * z \\
\quad\quad\quad \textbf{else } k := k/\mathbf{2}; z := z * z\} \\
\{y = x^n\}
\end{array}
$$

(As in the previous example, $x = \overleftarrow{x}$ and $n = \overleftarrow{n}$.) First we identify the invariant. Since k is going down towards zero, we might try something like $y = x^{n-k} \wedge k \geq \mathbf{0} \wedge x \neq \mathbf{0}$, but this isn't invariant. A better invariant turns out to be $P \equiv yz^k = x^n \wedge k \geq \mathbf{0} \wedge x \neq \mathbf{0}$.

To establish the invariant, we make three applications of our axiom of assignment:

$$
\begin{array}{l}
\{\underbrace{yx^k = x^n \wedge k \geq \mathbf{0} \wedge x \neq \mathbf{0}}_{\equiv P'}\} z := x\{P\} \\
\{\underbrace{\mathbf{1}x^k = x^n \wedge k \geq \mathbf{0} \wedge x \neq \mathbf{0}}_{\equiv P''}\} y := \mathbf{1}\{P'\} \\
\{\underbrace{\mathbf{1}x^n = x^n \wedge n \geq \mathbf{0} \wedge x \neq \mathbf{0}}_{\equiv P'''}\} k := n\{P''\}
\end{array}
$$

We then use these statements in rules of composition:

$$
\frac{\{P''\}y := \mathbf{1}\{P'\},\ \{P'\}z := x\{P\}}{\{P''\}y := \mathbf{1}; z := x\{P\}}
$$

$$
\frac{\{P'''\}k := n\{P''\},\ \{P''\}y := \mathbf{1}; z := x\{P\}}{\{P'''\}k := n; y := \mathbf{1}; z := x\{P\}}
$$

Now, since

$$
P''' \equiv (x^n = x^n \wedge n \geq \mathbf{0} \wedge x \neq \mathbf{0}) \equiv (n \geq \mathbf{0} \wedge x \neq \mathbf{0})
$$

which is our input condition, then invoking a rule of consequence will establish our invariant.

Next we verify that the loop invariant is invariant. By use of axiom of assignment, we know

$$\{\underbrace{y * (z * z)^k = x^n \wedge k \geq 0 \wedge x \neq 0}_{\equiv E'}\} z := z * z \{P\}$$

and

$$\{\underbrace{y * (z^2)^{k/2} = x^n \wedge k/2 \geq 0 \wedge x \neq 0}_{\equiv E''}\} k := k/2 \{E'\}$$

We can compose these two expressions by

$$\frac{\{E''\} k := k/2 \{E'\}; \{E'\} z := z * z \{P\}}{\{E''\} k := k/2; z := z * z \{P\}}$$

Likewise, by assignment we know

$$\{\underbrace{y * z * z^k = x^n \wedge k \geq 0 \wedge x \neq 0}_{\equiv T'}\} y := y * z \{P\}$$

and

$$\{\underbrace{y * z * z^{k-1} = x^n \wedge k - 1 \geq 0 \wedge x \neq 0}_{\equiv T''}\} k := k - 1 \{T'\}$$

so these in turn can be composed by

$$\frac{\{T''\} k := k - 1 \{T'\},\ \{T'\} y := y * z \{P\}}{\{T''\} k := k - 1; y := y * z \{P\}}$$

Next, since a: $y * z^k = x^n \wedge x \neq 0 \Rightarrow y * z * z^{k-1} = x^n$
b: $k \geq 0 \wedge k \neq 0 \Rightarrow k - 1 \geq 0$
c: $x \neq 0 \Rightarrow x \neq 0$
we can apply the rule of consequence as follows:

$$\frac{P \wedge k \neq 0 \wedge \ odd(k) \ \Rightarrow T'', \{T''\}k := k-1; y := y*z\{P\}}{\{P \wedge k \neq 0 \wedge \ odd(k)\}k := k-1; y := y*z\{P\}}$$

Likewise, since

a: $y * z^k = x^n \wedge x \neq 0 \wedge \neg\, odd(k) \ \Rightarrow y * (z^2)^{k/2} = x^n$
b: $k \geq 0 \wedge k \neq 0 \wedge \neg\, odd(k) \ \Rightarrow k/2 \geq 0$
c: $x \neq 0 \Rightarrow x^2 \neq 0$

we can apply a rule of consequence by:

$$\frac{P \wedge k \neq 0 \wedge \neg\, odd(k) \ \Rightarrow E'', \{E''\}k := k/2; z := z*z\{P\}}{\{P \wedge k \neq 0 \wedge \neg\, odd(k)\}\ k := k/2; z := z*z\ \{P\}}$$

This allows us to apply our **if** rule of inference

$$\frac{\{P \wedge k \neq 0 \wedge \ odd(k)\}k := k-1; y := y*z\{P\}, \quad \{P \wedge k \neq 0 \wedge \neg\, odd(k)\}k := k/2; z := z*z\{P\}}{\{P \wedge k \neq 0\}\mathbf{if}\ odd(k)\ \mathbf{then}\ \cdots\ \mathbf{else}\ \cdots\ \{P\}}$$

followed by our **while** rule of inference

$$\frac{\{P \wedge k \neq 0\}\mathbf{if}\ odd(k)\ \mathbf{then}\ \cdots\ \mathbf{else}\ \cdots\ \{P\}}{\{P\}\mathbf{while}\ k \neq 0\ \mathbf{do}\ \cdots\{P\ \wedge\ k = 0\}}$$

Now we compose the loop with its initialization:

$$\frac{\{n \geq 0 \wedge x \neq 0\}k := n; y := 1; z := x\{P\}, \quad \{P\}\mathbf{while}\ k \neq 0\ \mathbf{do}\ \cdots\{P \wedge k = 0\}}{\{n \geq 0 \wedge x \neq 0\}k := n; y := 1; z := x;\ \mathbf{while}\ k \neq 0\ \mathbf{do}\ \cdots\{P \wedge k = 0\}}$$

Finally, we show that the postcondition is a consequence of our input:

$$\frac{\{n \geq 0 \wedge x \neq 0\}k := n; y := 1; z :=;\ \mathbf{while}\ k \neq 0\ \mathbf{do}\ \cdots \quad \{P \wedge k = 0\}, P \wedge k = 0 \Rightarrow y = x^n}{\{n \geq 0 \wedge x \neq 0\}k := n; y := 1; z := x;\ \mathbf{while}\ k \neq\ \ \mathbf{do}\ \cdots\{y = x^n\}}$$

1.5. Yet Another Detailed Example: Slow Multiplication

Consider the following (relatively silly) program to perform multiplication:

$$
\begin{array}{l}
\textbf{PROGRAM} \equiv \{ \\
\quad s := \mathbf{0}; \\
\quad \textbf{while } x \neq \mathbf{0} \textbf{ do} \\
\quad\quad t := \mathbf{0} \\
\quad\quad \textbf{while } t \neq y \textbf{ do} \\
\quad\quad\quad s := s + \mathbf{1}; \\
\quad\quad\quad t := t + \mathbf{1}; \\
\quad\quad x := x - \mathbf{1}\}
\end{array}
$$

The desired output condition is $s = \overleftarrow{x}\,\overleftarrow{y}$. In this proof, it is convenient to use the following definitions, corresponding to the two loop invariants:

$$
\begin{aligned}
P &\triangleq s = (\overleftarrow{x} - x)\overleftarrow{y} \wedge y = \overleftarrow{y} \\
Q &\triangleq s = (\overleftarrow{x} - x)\overleftarrow{y} + t \wedge y = \overleftarrow{y}
\end{aligned}
$$

Starting with the innermost **while** statement, we have the following expressions by use of the axiom of assignment:

$$
\begin{array}{l}
\{Q^{t}_{t+1}\} t := t + \mathbf{1} \{Q\} \\
\{Q^{t}_{t+1}{}^{s}_{s+1}\} s := s + \mathbf{1} \{Q^{t}_{t+1}\}
\end{array}
$$

By composition of the above lines we infer:

$$
\{Q^{t}_{t+1}{}^{s}_{s+1}\} s := s + \mathbf{1}; t := t + \mathbf{1} \{Q\}
$$

Next, we observe that

$$
\begin{aligned}
Q^{t}_{t+1}{}^{s}_{s+1} &\equiv s + \mathbf{1} = (\overleftarrow{x} - x)\overleftarrow{y} + t + \mathbf{1} \wedge y = \overleftarrow{y} \\
&\equiv s = (\overleftarrow{x} - x)\overleftarrow{y} + t \wedge y = \overleftarrow{y} \\
&\equiv Q
\end{aligned}
$$

and therefore we trivially know that $Q \wedge t \neq y \Rightarrow Q^{t}_{t+1}{}^{s}_{s+1}$, which in turn allows us to apply a rule of consequence to infer

$$\frac{Q \wedge t \neq y \Rightarrow Q^{t\ \ s}_{t+1\ s+1}\ ,\ \{Q^{t\ \ s}_{t+1\ s+1}\}s := s+1; t := t+1\{Q\}}{\{Q \wedge t \neq y\}s := s+1; t := t+1\{Q\}}$$

Our **while** inference rule then allows us

$$\frac{\{Q \wedge t \neq y\}s := s+1; t := t+1\{Q\}}{\{Q\}\mathbf{while}\, t \neq y\ \mathbf{do}\ \cdots\{Q \wedge t = y\}}$$

Next, our axiom of assignment provides that

$$\{P^x_{x-1}\}x := x-1\{P\}$$

so the rule of consequence implies that

$$\frac{Q \wedge t = y \Rightarrow P^x_{x-1}, \{P^x_{x-1}\}x := x-1\{P\}}{\{Q \wedge t = y\}x := x-1\{P\}}$$

since

$$\begin{array}{ll} Q \wedge t = y & \equiv s = (\overleftarrow{x} - x)\overleftarrow{y} + t \wedge y = \overleftarrow{y} \wedge t = y \\ P^x_{x-1} & \equiv s = (\overleftarrow{x} - (x-1))\overleftarrow{y} \wedge y = \overleftarrow{y} \\ & \equiv s = (\overleftarrow{x} - x)\overleftarrow{y} + \overleftarrow{y} \wedge y = \overleftarrow{y} \end{array}$$

and
a: $y = \overleftarrow{y} \Rightarrow y = \overleftarrow{y}$.
b: $s = (\overleftarrow{x} - x)\overleftarrow{y} + t \wedge t = y \wedge y = \overleftarrow{y} \Rightarrow s = (\overleftarrow{x} - x)\overleftarrow{y} + \overleftarrow{y}$.

Hence we may compose this expression with our **while** loop by:

$$\frac{\{Q\}\mathbf{while}\, t \neq y\ \mathbf{do}\ \cdots\{Q \wedge t = y\}, \{Q \wedge t = y\}x := x-1\{P\}}{\{Q\}\mathbf{while}\, t \neq y\ \mathbf{do}\ \cdots\ ; x := x-1\{P\}}$$

By assignment we know $\{Q^t_0\}t := 0\{Q\}$, so this can be composed with the previous expression yielding:

$$\{Q^t_0\}t := 0;\ \mathbf{while}\, t \neq y\ \mathbf{do}\ \cdots; x := x-1\{P\}$$

Since $Q_0^t \equiv s = (\overleftarrow{x} - x)\overleftarrow{y} \wedge y = \overleftarrow{y}$ and $P \Rightarrow Q_0^t$, the preceding expression can be used in a rule of consequence to infer

$$\{P \wedge x \neq 0\} t := 0;\ \textbf{while}\ t \neq y\ \textbf{do}\ \cdots; x := x - 1\{P\}$$

which can then be immediately used in another application of our **while** rule of inference yielding:

$$\{P\}\textbf{while}\ x \neq 0\ \textbf{do}\ \cdots\{P \wedge x = 0\}$$

This expression together with $s = (\overleftarrow{x} - x)\overleftarrow{y} \wedge y = \overleftarrow{y} \wedge x = 0 \Rightarrow s = \overleftarrow{x}\,\overleftarrow{y}$ imply

$$\{P\}\textbf{while}\ x \neq 0\ \textbf{do}\ \cdots\{s = \overleftarrow{x}\,\overleftarrow{y}\}$$

due to our rule of consequence. Finally, moving into the initialization part of our program, assignment gives us:

$$\{P_0^s\}s := 0\{P\}$$

then composed with the **while** body by

$$\frac{\{P_0^s\}s := 0\{P\}, \{P\}\textbf{while}\ x \neq 0\ \textbf{do}\ \cdots\{s = \overleftarrow{x}\,\overleftarrow{y}\}}{\{P_0^s\}s := 0;\ \textbf{while}\ x \neq 0\ \textbf{do}\ \cdots\{s = \overleftarrow{x}\,\overleftarrow{y}\}}$$

At last, since $P_0^s \equiv 0 = (\overleftarrow{x} - \overleftarrow{x})\overleftarrow{y} \wedge \overleftarrow{y} = \overleftarrow{y}$, we know by rule of consequence that

$$\frac{\textbf{true} \Rightarrow P_0^s, \{P_0^s\}s := 0;\ \textbf{while}\ \cdots\{s = \overleftarrow{x}\,\overleftarrow{y}\}}{\{\textbf{true}\}\ \textbf{PROGRAM}\ \{s = \overleftarrow{x}\,\overleftarrow{y}\}}$$

2. CHOOSING INVARIANTS

There are clearly two 'tough' problems lurking within this axiomatic approach to proving partial correctness, and these are

• writing a “reasonable” target specification for your program, and

• identifying (i.e. guessing) the invariant for use of our **while** loop inference rule.

For the most part, the first of these above problems is beyond what we have traditionally dealt with in this course, and has been relegated to courses in design and software engineering (if at all). The second of these problems is what stops us from automating the proofs in axiomatic methods.

So the burden of dealing with invariants is shifted to the human problem solver. There are a few points that can partially ease this burden for us. First, we can clearly identify the loop test B from simple inspection. Likewise we know (by the “backwards reasoning” already illustrated in earlier examples here) what assertion will be needed right after execution of the **while** loop. One part of our invariant, then, can be suggested by “factoring out” the $\neg B$ from this assertion. For instance, if by reasoning back we determine that the output condition for the loop must be

$$O \triangleq \quad x \geq \mathbf{0} \wedge \mathbf{0} \leq r < y \wedge x = qy + r$$

and if our loop test is $B \equiv r \geq y$, then it follows from our understanding of the inference rule that part of our invariant must be “$O - \neg B$,” that is,

$$x \geq \mathbf{0} \wedge \mathbf{0} \leq r \wedge x = qy + r$$

After coming up with some assertion I, then a check on the proposed invariant that I ***and*** the negated loop condition are strong enough to prove the output assertion from the loop. If $I \wedge \neg B$ cannot give you the next desired assertion, then there is little reason to spend time insuring that $I \wedge B$ maintain invariance through execution of the loop body.

3. ARRAY ASSIGNMENT

Consider a use of our existing axiom of assignment as in the expression below:

$$\{?\}x[i] := y\{x[j] = z\}$$

What do we need *before* an assignment to an array element to guarantee the given equality *after* the assignment? If the expression only uses array entries in the right-hand side of the assignment, then our old rule still works with a problem. However, we need an inference rule to handle when the target of an assignment is an array value.

As mentioned in the introductory chapter, the key is to consider array assignments as operations which change the value of an entire array rather than a single element. We will consider the entire set of values that every element of an array has, and view an array assignment as a function which transforms this set of values into a new set with a specific member of that set altered.

$x[i] := y$ assigns to x the function like the old value of x except that it maps i to y. That is,

$$\alpha(x, i, y)[j] \triangleq \begin{cases} y, & \textit{when } j = i \\ x[j], & \textit{when } j \neq i \end{cases}$$

Hence, we can discuss and manipulate the different "states" attained by our array during a program execution simply by choosing the appropriate function name. Therefore, our new axiom schema is

$$\{P^x_{\alpha(x,i,y)}\}x[i] := y\{P\}$$

We will later see (in Chapter 6) that this axiom is just the denotational semantics definition of assignment. Below are examples of the new "function" names that can arise to represent different array states due to assignment:

$$\{\alpha(x, k, 7)[4] = 4 \wedge \alpha(x, k, 7)[y] = 5\}x[k] := 7\{x[4] = 4 \wedge x[y] = 5\}$$
$$\{\alpha(x, k, k-1)[5] = 5\}x[k] := k - 1\{x[5] = 5\}$$
$$\{\alpha(\alpha(x, j, j+1), k, k-1)[5] = 5\}x[j] := j + 1\{\alpha(x, k, k-1)[5] = 5\}$$

We can use this new axiom with our previous rules of inference to prove properties of programs containing arrays, such as:

$$\{\alpha(x,k,7)[1]=7\}x[k]:=7\{x[1]=7\}$$

$$\{\alpha(x,1,7)[1]=7\}k:=1\{\alpha(x,k,7)[1]=7\}$$

$$\mathbf{true} \;\Rightarrow\; \alpha(x,1,7)[1]=7$$

$$\frac{\{\mathbf{true}\}\ k:=1\ \{\alpha(x,k,7)[1]=7\};\{\alpha(x,k,7)[1]=7\}x[k]:=7\{x[1]=7\}}{\{\mathbf{true}\}\ k:=1;x[k]:=7\{x[1]=7\}}$$

3.1. Example: Shifting Values in an Array

Consider the following program (adapted from [53]) for shifting the values in an array one element to the left:

$$\begin{array}{l} \mathbf{PROGRAM} \equiv \{ \\ \quad k := a; \\ \quad \mathbf{while}\ k < b\ \mathbf{do} \\ \quad\quad k := k+1; \\ \quad\quad x[k-1] := x[k]\} \end{array}$$

For this program, the input condition is $\{a \le b \ \wedge\ x = \overleftarrow{x}\}$, and our desired output assertion is $\{\forall i, a \le i < b, x[i] = \overleftarrow{x}[i+1]\}$.

First we identify the invariant as being

$$\begin{array}{ll} I \triangleq & (a \le k \le b) \wedge (\forall i, a \le i < k, x[i] = \overleftarrow{x}[i+1]) \\ & \wedge(\forall i, k \le i \le b, x[i] = \overleftarrow{x}[i]) \end{array}$$

Our input assertion trivially implies

$$(a \le a \le b)\ \wedge\ (\forall i, a \le i < a, x[i] = \overleftarrow{x}[i+1])\ \wedge\ (x = \overleftarrow{x})$$

since the first clause contains an additional equality, the third clause is accepted without change, and the second clause is true vacuously and can hence be added in conjunction.

Since we have

$$\{(a \le a \le b)\ \wedge\ (\forall i, a \le i < a, x[i] = \overleftarrow{x}[i+1])\ \wedge\ (x = \overleftarrow{x})\}k := a\{I\}$$

due to the axiom of assignment, we can then use a rule of consequence to infer that

$$\{(a \leq b) \;\wedge\; (x = \overleftarrow{x})\}\; k := a\; \{I\}$$

Hence we have established the invariant.

Our next step is to show that the loop body maintains I. When Q is defined by

$$\begin{aligned} Q \triangleq \quad & (a \leq k \leq b) \wedge (\forall i, a \leq i < k, \alpha(x, k-\mathbf{1}, x[k])[i] = \overleftarrow{x}[i+\mathbf{1}]) \\ & \wedge(\forall i, k \leq i \leq b, \alpha(x, k-\mathbf{1}, x[k])[i] = \overleftarrow{x}[i]) \end{aligned}$$

then by the array axiom of assignment, we know

$$\{Q\}\; x[k-\mathbf{1}] := x[k]\; \{I\}$$

By a similar use of the array axiom of assignment,

$$\begin{aligned} &\{(a \leq k+\mathbf{1} \leq b) \wedge \\ &\quad (\forall i, a \leq i < k+\mathbf{1}, \alpha(x, k+\mathbf{1}-\mathbf{1}, x[k+\mathbf{1}])[i] = \overleftarrow{x}[i+\mathbf{1}]) \wedge \\ &\quad (\forall i, k+\mathbf{1} \leq i \leq b, \alpha(x, k+\mathbf{1}-\mathbf{1}, x[k+\mathbf{1}])[i] = \overleftarrow{x}[i])\} \\ &k := k+\mathbf{1}\; \{Q\} \end{aligned}$$

By simple evaluations, the antecedent in the above line is shown to be equivalent to

$$\begin{aligned} &(a \leq k+\mathbf{1} \;\wedge\; k < b) \;\wedge\; (\forall i, a \leq i \leq k, \alpha(x, k, x[k+\mathbf{1}])[i] = \overleftarrow{x}[i+\mathbf{1}]) \\ &\quad \wedge (\forall i, k < i \leq b, \alpha(x, k, x[k+\mathbf{1}])[i] = \overleftarrow{x}[i]) \;\equiv\; P \end{aligned}$$

and therefore, due to a rule of consequence, we can infer that $\{P\}k := k+1\{Q\}$. We now apply the above lines in a rule of composition, by

$$\frac{\{P\}k := k+\mathbf{1}\{Q\},\; \{Q\}x[k-\mathbf{1}] := x[k]\{I\}}{\{P\}k := k+\mathbf{1};\; x[k-\mathbf{1}] := x[k]\{I\}}$$

In order for us to apply our **while** inference rule later on, we will need to show how our expression P “matches” the pattern $I \wedge B$ needed in the rule. To do this, we need the rule of consequence that will show

$$\frac{(I \wedge B) \Rightarrow P,\ \{P\}k := k+1;\ x[k-1] := x[k]\{I\}}{\{I \wedge B\}k := k+1; x[k-1] := x[k]\{I\}}$$

Clearly, B must be $k < b$. Now we will consider each of the clauses of P in turn:

1. We know $a \leq k \leq b$ from I and also $k < b$ from B, hence $(a \leq k+1\ \wedge\ k < b)$ in P.

2. The expression $\forall i, a \leq i \leq k, \alpha(x, k, x[k+1])[i] = \overleftarrow{x}[i+1]$ can be simplified using the definition of α. If $i \neq k$, the expression simplifies to $\forall i, a \leq i < k, x[i] = \overleftarrow{x}[i+1]$. If $i = k$, the expression simplifies to

$$x[k+1] = \overleftarrow{x}[i+1]$$

or (since $i = k$) $x[k+1] = \overleftarrow{x}[k+1]$. Thus the original expression is equivalent to

$$(\forall i, a \leq i < k, x[i] = \overleftarrow{x}[i+1])\ \wedge\ (x[k+1] = \overleftarrow{x}[k+1])$$

The first clause of this conjunction is exactly a clause of I, and the second clause of this expression directly follows from the last clause of I.

3. The expression $\forall i, k < i \leq b, \alpha(x, k, x[k+1])[i] = \overleftarrow{x}[i]$ can also be simplified because i's range of values ($k+1$ through b) does not include k.

$$(\forall i, k < i \leq b, \alpha(x, k, x[k+1])[i] = \overleftarrow{x}[i])\ \equiv\ (\forall i, k < i \leq b, x[i] = \overleftarrow{x}[i])$$

The simplified expression is implied by the last clause of I.

Therefore, $I \wedge B \Rightarrow P$.

Now that we have the right pattern, we can directly apply our **while** axiom, to infer

$$\{I\}\mathbf{while}\ B\ \mathbf{do}\ k := k+1;\ x[k-1] := x[k]\{I \wedge \neg B\}$$

which in turn can be composed with the initialization part of our program yielding

$$\{a \leq b \;\wedge\; x = \overleftarrow{x}\}\; k := a;\; \textbf{while} \cdots \;\{I \;\wedge\; k \geq b\}$$

Since $I \wedge k \geq b$ implies that $k = b$, which in turn implies the first clause of I is identically $\forall i, a \leq i < b, x[i] = \overleftarrow{x}[i+1]$, a rule of consequence allows us to infer that

$$\{a \leq b \;\wedge\; x = \overleftarrow{x}\} k := a;\; \textbf{while} \;\cdots\; \{\forall i, a \leq i < b, x[i] = \overleftarrow{x}[i+1]\}$$

where the consequent is our desired output assertion.

3.2. Detailed Example: Reversing an Array

In all its gory detail, here is a program to reverse an array $x[0 \ldots n]$, along with a proof of partial correctness:

```
PROGRAM ≡ {
    i := 0;
    while i ≤ n/2 do
        t := x[i];
        x[i] := x[n − i];
        x[n − i] := t;
        i := i + 1}
```

Its input condition is $n \geq 1 \;\wedge\; \forall k, 0 \leq k \leq n, x[k] = \overleftarrow{x}[k]$ and the desired output condition is

$$\forall k, 0 \leq k \leq n/2, x[k] = \overleftarrow{x}[n-k] \wedge x[n-k] = \overleftarrow{x}[k]$$

The invariant we choose is

$$\begin{aligned} I \triangleq \quad & (i \leq n/2 + 1) \wedge \\ & (\forall k, 0 \leq k < i, x[k] = \overleftarrow{x}[n-k] \wedge x[n-k] = \overleftarrow{x}[k]) \wedge \\ & (\forall k, i \leq k \leq n-i, x[k] = \overleftarrow{x}[k]) \end{aligned}$$

By our axiom of assignment,

$$\begin{aligned} &\{(0 \leq n/2) \;\wedge\; (\forall k, 0 \leq k < 0, x[k] = \overleftarrow{x}[n-k] \wedge x[n-k] = \overleftarrow{x}[k]) \wedge \\ &\quad (\forall k, 0 \leq k \leq n-0, x[k] = \overleftarrow{x}[k]\} i := 0 \;\{I\} \end{aligned}$$

Since the antecedent of this expression is implied by our input condition (since the input directly implies clauses one and three of this expression and clause two is satisfied vacuously), we can conclude by a rule of consequence that

$$\{\mathbf{input}\}\ i := \mathbf{0}\ \{I\}$$

Next we must show that the loop maintains the invariant. By repeated use of our axioms of assignment, we obtain the following sequence of expressions:

$$\begin{aligned}
&\{(i+\mathbf{1} \le n/\mathbf{2}+\mathbf{1}) \wedge (\forall k, \mathbf{0} \le k < i+\mathbf{1}, x[k] = \overleftarrow{x}[n-k] \wedge \\
&\quad x[n-k] = \overleftarrow{x}[k]) \wedge \underbrace{(\forall k, i+\mathbf{1} \le k \le n-i-\mathbf{1}, x[k] = \overleftarrow{x}[k]}_{\equiv S}\} \\
&\quad i := i+\mathbf{1}\{I\}
\end{aligned}$$

$$\begin{aligned}
&\{(i \le n/\mathbf{2}) \wedge \\
&\quad (\forall k, \mathbf{0} \le k \le i, \alpha(x, n-i, t)[k] = \overleftarrow{x}[n-k] \wedge \\
&\quad \alpha(x, n-i, t)[n-k] = \overleftarrow{x}[k]) \wedge \\
&\quad \underbrace{(\forall k, i+\mathbf{1} \le k \le n-i-\mathbf{1}, \alpha(x, n-i, t)[k] = \overleftarrow{x}[k]}_{\equiv R}\}\ x[n-i] := t\{S\}
\end{aligned}$$

$$\begin{aligned}
&\{(i \le n/\mathbf{2}) \wedge \\
&\quad (\forall k, \mathbf{0} \le k \le i, \alpha(\alpha(x, i, x[n-i]), n-i, t)[k] = \overleftarrow{x}[n-k] \wedge \\
&\quad \alpha(\alpha(x, i, x[n-i]), n-i, t)[n-k] = \overleftarrow{x}[k]) \wedge \\
&\quad \underbrace{(\forall k, i+\mathbf{1} \le k \le n-i-\mathbf{1}, \alpha(\alpha(x, i, x[n-i]), n-i, t)[k] = \overleftarrow{x}[k]}_{\equiv Q}\} \\
&\quad x[i] := x[n-i]\{R\}
\end{aligned}$$

$$\begin{aligned}
&\{(i \le n/\mathbf{2}) \wedge \\
&\quad (\forall k, \mathbf{0} \le k \le i, \alpha(\alpha(x, i, x[n-i]), n-i, x[i])[k] = \overleftarrow{x}[n-k] \wedge \\
&\quad \alpha(\alpha(x, i, x[n-i]), n-i, x[i])[n-k] = \overleftarrow{x}[k]) \wedge \\
&\quad \underbrace{(\forall k, i+\mathbf{1} \le k \le n-i-\mathbf{1}, \alpha(\alpha(x, i, x[n-i]), n-i, x[i])[k] = \overleftarrow{x}[k]}_{\equiv P}\} \\
&\quad t := x[i]\{Q\}
\end{aligned}$$

By repeated use of our inference rule of composition, we thus know that

$$\{P\}\ \cdots \mathit{while\ body} \cdots\ \{I\}$$

However, in order to apply our **while** rule of inference, we must have the expression $I \wedge B$ as the antecedent (where B is $i \leq n/2$.) Hence, we must determine whether $I \wedge B \Rightarrow P$. If so, then a rule of consequence will give us the desired pattern for application of the **while** inference rule. That, in turn, would allow us to compose the **while** with the initialization of our program, yielding

$$\{\mathbf{input}\}\mathbf{PROGRAM}\{I \wedge \neg B\}$$

$I \wedge \neg B \Rightarrow \mathbf{output}$ because $i \leq n/2+1) \wedge i > n/2)$ imply $i = n/2+1$. Thus applying a rule of consequence would finish off our proof. Hence, all that remains for us is to verify that $I \wedge B \Rightarrow P$.

>From this point on, it will be convenient for us to refer to the three clauses of our invariant I as I_1, I_2, and I_3. Likewise, P is a conjunction of three clauses. P's first clause is trivially implied by $I \wedge B$, and each of its remaining two clauses can in turn be broken down to a pair of conjunctions each. We will consider each of these in turn as enumerated below to complete the proof.

• Show that $I \wedge B \Rightarrow$

$$\forall k, 0 \leq k \leq i, \alpha(\alpha(x, i, x[n-i]), n-i, x[i])[k] = \overleftarrow{x}[n-k]$$

By simplifying the expression containing α notation, we obtain three cases.

1. If $k = n-i$, the expression simplifies to $(\forall k, 0 \leq k \leq i, x[i] = \overleftarrow{x}[n-k])$. Substituting i for $n-k$, we can also eliminate the quantifier to obtain $x[i] = \overleftarrow{x}[i]$.

2. If $k \neq n-i \wedge k = i$, the expression simplifies to $(\forall k, 0 \leq k \leq i, x[n-i] = \overleftarrow{x}[n-k])$. Substituting i for k, we can also eliminate the quantifier to obtain $x[n-i] = \overleftarrow{x}[n-i]$.

3. If $k \neq n-i \wedge k \neq i$, the expression simplifies to $(\forall k, 0 \leq k < i, x[i] = \overleftarrow{x}[n-k])$.

Reassembling the parts, we have

$$(x[i] = \overleftarrow{x}[i]) \ \wedge \ (x[n-i] = \overleftarrow{x}[n-i]) \ \wedge \ (\forall k, 0 \leq k < i, x[i] = \overleftarrow{x}[n-k])$$

The first two conjuncts are implied by I_3. (Note that I_3 is not vacuously true since $i < n/2$ and $n - i > n/2$.) The final conjunct is implied by I_2.

• Show that $I \wedge B \Rightarrow$

$$\forall k, 0 \leq k \leq i, \alpha(\alpha(x, i, x[n-i]), n-i, x[i])[n-k] = \overleftarrow{x}[k]$$

By simplifying, we obtain the following cases.

1. If $i = k$, the expression simplifies to $(\forall k, 0 \leq k \leq i, x[i] = \overleftarrow{x}[k])$. Substituting i for k, we can eliminate the quantifier to obtain $x[i] = \overleftarrow{x}[i]$.

2. If $i \neq k \wedge n-k = i$, the expression simpifies to $(\forall k, 0 \leq k \leq i, x[n-i] = \overleftarrow{x}[k]$. Substituting $n - i$ for k, we can eliminate the quantifier to obtain $x[n-i] = \overleftarrow{x}[n-i]$.

3. If $i \neq k \wedge n-k \neq i$, the expression simpifies to $(\forall k, 0 \leq k < i, x[n-k] = \overleftarrow{x}[k])$.

Reassembling the parts, we have

$$(x[i] = \overleftarrow{x}[i]) \ \wedge \ (x[n-i] = \overleftarrow{x}[n-i]) \ \wedge \ (\forall k, 0 \leq k < i, x[n-k] = \overleftarrow{x}[k])$$

which is identical to the previous case.

• Show that $I \wedge B \Rightarrow$

$$\forall k, i+1 \leq k \leq n-i-1, \alpha(\alpha(x, i, x[n-i]), n-i, x[i])[k] = \overleftarrow{x}[k]$$

Since the range of values for k includes neither i nor $n-i$, the expression can be simplified to

1. $\forall k, i+1 \leq k \leq n-i-1, x[k] = \overleftarrow{x}[k]$.

This last expression is implied by I_3.

4. PROCEDURE CALL INFERENCE RULES

We are now in a position to extend the language we are modeling to include code blocks and procedure calls. As we extend our language, so must we extend the inference rules available to us for reasoning about procedures. Our goal is only to prove a procedure's body once, and then use that result as needed when that procedure is called throughout the main code body.

In order to define scope rules and local variables, variable x inside statement S will be a local variable if it does not affect any pre- or postcondition around S. That is, if $\{P\}S\{Q\}$ is proven, and variable x does not appear in either P or Q, then we can make x a local variable inside S. By defining S^x_y to mean substitute all occurrences of x by a variable y that does not appear in either P or Q, then we have the following inference rule for declarations:

$$\textbf{Declaration:} \quad \frac{\{P\}S^x_y\{Q\}}{\{P\}\ \textbf{begin new}\ x;\ S\ \textbf{end}\ \{Q\}}$$

A trivial program illustrating the use of this new inference rule is:

$$\begin{array}{l} \textbf{PROGRAM} \equiv \{ \\ \quad x := 1; \\ \quad \textbf{begin} \\ \quad\quad \textbf{new}\ x; \\ \quad\quad x := 2 \\ \quad\quad y := x + 1 \\ \quad \textbf{end} \\ \quad y := x + y\} \end{array}$$

We should be able to prove that with precondition **true**, then a postcondition is $y = 4$. We have to show that assignment of 2 to x within the **begin** block has no effect on the outer value of x.

1. $\{x + y = 4 \ \wedge \ x = 1\}y := y + x\{y = 4 \ \wedge \ x = 1\}$ $\quad assignment$

This gives the postcondition for the **begin** block.

In determining the internal **begin** block, the postcondition involves x. So we must use a "local" variable different from x. Since it will be

bound to the block we are creating, the name we use has no real effect, so choose any name different from x. Therefore, choose variable z that does not appear in either pre- or postcondition for the block to replace the local variable x. Therefore, the body of the block becomes:

$$z := 2$$
$$y := z + 1$$

Verifying the block:

$$
\begin{array}{lll}
1. & \{x+z=3 \ \wedge \ x=1\}y:=z+1 & assignment \\
 & \quad \{x+y=4 \ \wedge \ x=1\} & \\
2. & \{x=1 \ \wedge \ x=1\}z:=2\{x+z=3 \ \wedge \ x=1\} & assignment \\
3. & \underline{\{x=1\}z:=2; y:=z+1\{x+y=4 \ \wedge \ x=1\}} & decl.(x \ for \ z) \\
 & \{x=1\}\mathbf{begin\ new}\ x; x:=2; y:=x+1\ \mathbf{end} & \\
 & \quad \{x+y=4 \ \wedge \ x=1\} & \\
4. & \{1=1\}x:=1\{x=1\} & assignment \\
5. & \{1=1\}x:=1\{x=1\}, & composition(4,3) \\
 & \quad \underline{\{x=1\}\mathbf{begin}\ldots\mathbf{end}\{x+y=4\wedge x=1\}} & \\
 & \underline{\{1=1\}x:=1;\mathbf{begin}\cdots\{x+y=4\wedge x=1\}} & \\
6. & \mathbf{true} \ \Rightarrow 1=1, & consequence(5) \\
 & \underline{\{1=1\}x:=1;\ \mathbf{begin}\cdots\{x+y=4\wedge x=1\}} & \\
 & \{\mathbf{true}\}\ \mathbf{PROGRAM}\ \{y=4\wedge x=1\} &
\end{array}
$$

The next rules deal directly with procedure calls. Once a body of code has been verified separately, our next step will usually be to encapsulate this body in a procedure, then condition that procedure name for use in the caller's context. Conditioning a **call** for use in a main program body is itself a two step activity: first the ***actual*** parameter names must be installed, and then the assertions which describe procedure's behavior must be adapted for use in the caller's context.

4.1. Invocation

If we can show that $\{P\}S\{Q\}$ then we should be able to replace S by a procedure ω containing S. The problem is the arguments to ω may contain variables that appear in P and Q.

Our rule of ***invocation***, wherein we can capture a code body as a procedure, is

$$\textbf{Invocation:} \qquad \frac{\omega(x):(v)\ \textbf{procedure}\ S,\ \{P\}S\{Q\}}{\{P\}\ \textbf{call}\ \omega(x):(v)\ \{Q\}}$$

where x and v represent ***lists*** of nonlocal variables which ***can*** change and which do ***not*** change in the body of code S, respectively. Note that these do ***not*** directly correspond to our usual notion of "by reference" versus "by value" parameters. The key difference is that we distinguish here between parameters based on whether they change in the ***body*** of the code, as opposed to whether the parameter would have been changed after exiting from the body and returning to the caller. Furthermore, we explicitly disallow the body S from ever referencing a nonlocal variable (hence all program variables which are used ***must*** appear in either x or v.) Informally, we would read this rule as "If ω is a procedure whose body is implemented by S, and if we know executing S yields Q from P, then we can infer that calling ω from state P will yield Q."

4.2. Substitution

Next is a rule of ***substitution*** for obtaining an expression involving ***actual*** (instead of ***formal***) parameters in a procedure call. We would like a simple rule like the following:

$$\textbf{Almost substitution:} \qquad \frac{\{P\}\ \textbf{call}\ \omega(x):(v)\ \{Q\}}{\{P^{x\ v}_{a\ e}\}\ \textbf{call}\ \omega(a):(e)\ \{Q^{x\ v}_{a\ e}\}}$$

where x and v again represent the lists of formal parameters which change and do not change, respectively; and a and e represent the ***actual*** parameters in the call.

It seems like we should simply be able to replace x and v by a and e everywhere in P, S and Q and have substitution work. Unfortunately, the use of aliases – multiple names for the same object – prevents such a simple substitution rule. Consider the procedure:

$p(x):(y)$ **procedure** $\equiv$ {
  precondition $true$
  postcondition $x = y + 1$
  $x := y + 1;$
  }

and the invocation call $p(a) : (a)$. Our proposed substitution rule allows us to conclude $a = a + 1$! Our problem is that we have aliased a to x and a to y in the **call** statement. To reduce aliasing problems, we forbid duplicate identifiers from appearing in the list of actual variable parameters, and the same identifier from appearing in both actual parameter lists.

Unfortunately, this simple restriction does not eliminate aliasing problems. The pre- and post-conditions may contain assertions about non-local variables. Additional precautions are necessary if the non-local variables appear as actual parameters. Consider the following program:

$$
\begin{array}{l}
\mathbf{PROGRAM} \equiv \{ \\
\quad p(a) : (b, c)\ \mathbf{procedure}\ \equiv\ \{ \\
\quad\quad \mathbf{precondition}\ b = d \wedge c > \mathbf{0} \\
\quad\quad \mathbf{postcondition}\ a = b - c \wedge a < d \\
\quad\quad a := b - c\} \\
\quad e := d; \\
\quad f := \mathbf{1}; \\
\quad \mathbf{call}\ p(d) : (e, f)\}
\end{array}
$$

With precondition **true**, we can prove the postcondition $d = e - f \wedge d < d$! First show that the procedure body meets its specifications by starting with our axiom of assignment:

$$\{b - c = b - c \wedge (b - c) < d\}a := b - c\{a = b - c \wedge a < d\}$$

Our rule of consequence shows

$$\frac{b = d \wedge c > \mathbf{0} \Rightarrow b - c < d, \{b - c < d\}a := b - c\{a = b - c \wedge a < d\}}{\{b = d \wedge c > \mathbf{0}\}a := b - c\{a = b - c \wedge a < d\}}$$

Now apply the rule of invocation to obtain:

$$\{b = d \wedge c > \mathbf{0}\}\mathbf{call}\ p(a) : (b, c)\{a = b - c \wedge a < d\}$$

Next we apply our "simple" rule of substitution, substituting d for a, e for b and f for c to obtain:

$$\{e = d \wedge f > \mathbf{0}\}\mathbf{call}\, p(d) : (e, f)\{d = e - f \wedge d < d\}$$

We could then use assignment twice for:

$$\{e = d \wedge \mathbf{1} > \mathbf{0}\}f := \mathbf{1}\{e = d \wedge f > \mathbf{0}\}$$
$$\{d = d\}e := d\{e = d\}$$

Then composition for:

$$\{\mathbf{true}\}\ e := d; f := \mathbf{1}\{e = d \wedge f > \mathbf{0}\}$$

then

$$\{\mathbf{true}\}\ e := d; f := \mathbf{1};\ \mathbf{call}\ \cdots\{d = e - f \wedge d < d\}$$

which is clearly a result we do not desire. Of course, the problem is in the clash between the actual parameter d in the main program's call to p and the variable d as a "changeable" parameter in the procedure body. The error is in using too simple a rule of substitution.

The corrected rule of substitution is:

$$\mathbf{Substitution:} \qquad \frac{\{P\}\ \mathbf{call}\,\omega(x) : (v)\ \{Q\}}{\{P^{k}_{k'}\,{}^{x}_{a}\,{}^{v}_{e}\}\ \mathbf{call}\,\omega(a) : (e)\ \{Q^{k}_{k'}\,{}^{x}_{a}\,{}^{v}_{e}\}}$$

where x and v again represent the lists of formal parameters which change and do not change, respectively; a and e represent the ***actual*** parameters in the call; and the lists k and k' have the following special interpretation: k is a list of all symbols which are free in P and Q but do not appear in the procedure interface, and which furthermore correspond to symbols appearing in either a or e. This list is thus obtained by first enumerating all free variables in P and Q, crossing out those appearing in either x or v, and then crossing out those that do ***not*** appear in either a or e. Any of these symbols k which then appear in P or Q must then be replaced with completely new symbols k'. It is essential that this substitution be performed ***first*** in applying this substitution rule.

Returning to our previous example, the rule of substitution identifies the variable d as being a member of this list k and we introduce a

new variable d' in its place. Hence, at the point where substitution is applied, we should first obtain the expression

$$\{b = d' \wedge c > \mathbf{0}\}\mathbf{call}\; p(a) : (b, c)\{a = b - c \wedge a < d'\}$$

and only ***then*** apply our simple rule of substitution to obtain

$$\{e = d' \wedge f > \mathbf{0}\}\mathbf{call}\; p(d) : (e, f)\{d = e - f \wedge d < d'\}$$

We would then only be able to obtain the global postcondition

$$d = e - f \wedge d < d'$$

which is not nearly as unsettling a result.

4.3. Adaptation

Once we have an expression for procedure calls using the actual parameters, we are ready to take any assertions used to describe the procedure and adapt them to the context of the call. What we want is a rule similar to the previous rule of consequence, as in the following *incorrect* rule:

$$\textbf{Improper consequence}: \quad \frac{\{P\}\mathbf{call}\;\omega(a) : (e)\;\{R\},\quad P\;\wedge\;R \Rightarrow Q}{\{P\}\mathbf{call}\;\omega(a) : (e)\;\{Q\}}$$

Why can't we use such a rule? The reason is that we only know the pre- and postcondition to the call to ω, we do not know the details of ω. It is possible that there are changes to the parameter a or to other free variables within P and Q. These side effects make the above an invalid rule of inference.

Instead we need a more complex rule of ***adaptation***, given as:

$$\textbf{Adaptation}: \quad \frac{\{R\}\;\mathbf{call}\;\omega(a) : (e)\;\{S\}}{\{\exists k\;(R\;\wedge\;\forall a(S \Rightarrow T))\}\mathbf{call}\;\omega(a) : (e)\;\{T\}}$$

where k is the list of all variables free in R and S but not occurring in a, e, or T. While the motivation for why this rule appears as it does is obscure at first, what we are saying is that there must be some assignment to the free variables of R and S (i.e., k) which makes a true precondition R to the **call** of ω, such that for any possible alteration within ω to the parameter a, S will still imply T. If so, then we can use our "rule of consequence" above to produce T as a postcondition.

In order to better understand the role of the axioms to represent procedure calls, the following examples are presented.

Universal quantification in adaptation

The following example illustrates the need for universal quantification of parameters whose values may vary. Consider the following procedure p:

$$
\begin{array}{l}
p(x,y) : () \ \mathbf{procedure} \ \equiv \ \{ \\
\quad \mathbf{precondition} \ x = 3 \wedge y = 4 \\
\quad \mathbf{postcondition} \ x * y = 12 \\
\quad \mathbf{int} \ t; \\
\quad t := x; \\
\quad x := y; \\
\quad y := t; \\
\quad \}
\end{array}
$$

Suppose we want to show:

$$\{a = 3 \ \wedge \ b = 4\}\mathbf{call}\ p(a,b) : ()\{a * b = 12 \ \wedge \ a < b\}$$

After using invocation and substitution, we can conclude:

$$\{a = 3 \ \wedge \ b = 4\}\mathbf{call}\ p(a,b) : ()\{a * b = 12\}$$

Using the improper rule of consequence, we could combine this result with a proof of the implication:

$$(a = 3 \ \wedge \ b = 4 \ \wedge \ a * b = 12) \Rightarrow (a * b = 12 \ \wedge \ a < b)$$

to achieve our goal.

However, using adaptation, the results of invocation and substitution only permit us to conclude:

$$(a = 3 \ \wedge \ b = 4) \ \wedge \ (\forall a, b(a * b = 12) \Rightarrow (a * b = 12 \ \wedge \ a < b))$$
$$\textbf{call}\ p(a,b) : ()\{a * b = 12 \ \wedge \ a < b\}$$

Since any quantified variable can be renamed, we could rewrite the conclusion as:

$$(a = 3 \ \wedge \ b = 4) \ \wedge \ (\forall m, n(m * n = 12) \Rightarrow (m * n = 12 \ \wedge \ m < n))$$
$$\textbf{call}\ p(a,b) : ()\{a * b = 12 \ \wedge \ a < b\}$$

Thus, it is clear that assertions about values of variable parameters before a call (e.g., $a = 3 \ \wedge \ b = 4$) cannot be used to prove assertions about their values after the call (e.g., $m < n$).

Existential quantification in adaptation

Consider the trivial program

$$\{x = x'\}x := x + 1\{x = x' + 1\}$$

with the pre- and postconditions as given. We might elect to encapsulate this fragment into a procedure (call it $\omega(x) : ()$) and try to prove the program

$$\{true\}\ z := 1;\ \textbf{call}\ \omega(z) : ()\{z = 2\}$$

After proving the body of the procedure using assignment, we use invocation to conclude:

$$\frac{\{x = x'\}x := x + 1\{x = x' + 1\}}{\{x = x'\}\ \textbf{call}\ \omega(x) : ()\{x = x' + 1\}}$$

Then using our rule of substitution, we can obtain:

$$\frac{\{x = x'\}\ \textbf{call}\ \omega(x) : ()\{x = x' + 1\}}{\{z = x'\}\ \textbf{call}\ \omega(z) : ()\{z = x' + 1\}}$$

Using an adaptation rule without existentally quantifying x' we obtain:

$$\frac{\{z = x'\}\ \textbf{call}\ \omega(z) : ()\{z = x' + 1\}}{\{(z = x' \wedge \forall z(z = x' + 1 \Rightarrow z = 2))\}\ \textbf{call}\ \omega(z) : ()\{z = 2\}}$$

Now when we use the assignment axiom, we have:

$$\{1 = x' \wedge \forall z(z = x' + 1 \Rightarrow z = 2))\} z := 1 \{(z = x' \wedge \forall z(z = x' + 1 \Rightarrow z = 2))\}$$

or simply:

$$\{1 = x'\}\ z := 1\ \{(z = x' \wedge \forall z(z = x' + 1 \Rightarrow z = 2))\}$$

Unfortunately, we cannot show ($true \Rightarrow 1 = x'$) so that we can apply a rule of consequence to conclude

$$\{true\}\ z := 1\ \{(z = x' \wedge \forall z(z = x' + 1 \Rightarrow z = 2))\}$$

However, had we used the existential quantifier in adaptation, we would have concluded:

$$\frac{\{z = x'\}\ \textbf{call}\ \omega(z) : ()\{z = x' + 1\}}{\{\exists x'(z = x' \wedge \forall z(z = x' + 1 \Rightarrow z = 2))\}\ \textbf{call}\ \omega(z) : ()\{z = 2\}}$$

The result of the adaptation axiom asks whether there is any assignment to the free variables of the pre- and postconditions (e.g., 1 in this case) such that no matter what happens to z inside ω, the postcondition will imply the derived result. If our program's input assertion implies this, then we can always get the derived answer. Or in other words, we want to be able to prove:

$$z = 1 \Rightarrow \exists x'(z = x' \wedge \forall z(z = x' + 1 \Rightarrow z = 2))$$

But clearly this is the case, since we can satisfy the implication by selecting $x' = 1$ in the existential quantifier.

Since this last statement is just a logical implication, we can apply our normal rule of consequence to conclude:

$$\{z = 1\}\ \textbf{call}\ \omega(z) : ()\{z = 2\}$$

Now the assignment axiom yields:

$$\{1 = 1\}\ z := 1\ \{z = 1\}$$

and the proof can be completed using composition.

4.4. Detailed Example: Simple Sums

The following is a detailed example (adapted from [61]) of how our inference rules for procedure call and parameters can be employed.

$$
\begin{array}{l}
\textbf{PROGRAM} \equiv \{ \\
\quad \textbf{procedure } e(x) : (y, z) \equiv \{ \\
\quad\quad \textbf{precondition } \equiv z \geq 0 \land y \neq 0 \\
\quad\quad \textbf{postcondition } \equiv x = y^z \\
\quad\quad\quad \textbf{new } i; \\
\quad\quad\quad i := 0; \\
\quad\quad\quad x := 1; \\
\quad\quad\quad \textbf{while } i < z \textbf{ do} \\
\quad\quad\quad\quad i := i + 1; \\
\quad\quad\quad\quad x := xy\} \\
\quad \textbf{new } a, p, s, n; \\
\quad a := 1; \\
\quad n := 3; \\
\quad s := 0; \\
\quad \textbf{while } a \leq m \textbf{ do} \\
\quad\quad \textbf{call } e(p) : (a, n); \\
\quad\quad s := s + p; \\
\quad\quad a := a + 1\}
\end{array}
$$

For this program, the input condition is $m \geq 0$ and the output condition is $s = \sum_{i=1}^{m} i^3$.

Procedure body

Proof: We begin by showing that the procedure e satisfies its specification, that is, $\{y \neq 0 \land z \geq 0\} body\ of\ e\{x = y^z\}$. The loop invariant for the **while** statement in the procedure body is $y \neq 0 \land x = y^i \land 0 \leq i \leq z$.

First, we verify that the invariant holds before execution of the **while** statement. Twice using our axiom of assignment we see that

$$
\begin{array}{l}
\{1 = y^i \land 0 \leq i \leq z\} x := 1 \{x = y^i \land 0 \leq i \leq z\} \\
\{1 = y^0 \land 0 \leq 0 \leq z\} i := 0 \{1 = y^i \land 0 \leq i \leq z\}
\end{array}
$$

then these can be composed to yield

$$\{1 = y^0 \wedge 0 \leq 0 \leq z\} i := 0; x := 1 \{x = y^i \wedge 0 \leq i \leq z\}$$

Since $y \neq 0 \wedge z \geq 0 \Rightarrow 1 = y^0 \wedge 0 \leq 0 \leq z$, then a rule of consequence assures us that

$$\{y \neq 0 \wedge z \geq 0\} i := 0; x := 1 \{x = y^i \wedge 0 \leq i \leq z\}$$

Next we verify that the invariant is maintained by the loop. Twice applying our axiom of assignment, we find:

$$\{xy = y^i \wedge 0 \leq i \leq z\} x := xy \{x = y^i \wedge 0 \leq i \leq z\}$$
$$\{xy = y^{i+1} \wedge 0 \leq i+1 \leq z\} i := i+1 \{xy = y^i \wedge 0 \leq i \leq z\}$$

hence, by composition we know

$$\{xy = y^{i+1} \wedge 0 \leq i+1 \leq z\} i := i+1; x := xy \{x = y^i \wedge 0 \leq i \leq z\}$$

Our **while** loop conditional is clearly $i < z$, and since $x = y^i \wedge 0 \leq i \leq z \wedge i < z \Rightarrow xy = y^{i+1} \wedge 0 \leq i+1 \leq z$, a rule of consequence implies that

$$\{x = y^i \wedge 0 \leq i \leq z \wedge i < z\} i := i+1; x := xy \{x = y^i \wedge 0 \leq i \leq z\}$$

This expression verifies that the invariant is maintained, and further is clearly in the right form to apply our **while** rule of inference. Hence we know

$$\{x = y^i \wedge 0 \leq i \leq z\} \mathbf{while} \cdots \{x = y^i \wedge 0 \leq i \leq z \wedge \neg(i < z)\}$$

Now, since $i \leq z \wedge i \geq z \Rightarrow i = z$, then $x = y^i$ and $i = z$ imply that $x = y^z$ which is our desired output condition. All that now remains is for us to compose the initialization part of our program with the loop body, by

$$\frac{\{y \neq 0 \wedge z \geq 0\} i := 0; x := 1 \{x = y^i \wedge 0 \leq i \leq z\}, \quad \{x = y^i \wedge 0 \leq i \leq z\} \mathbf{while} \cdots \{x = y^z\}}{\{y \neq 0 \wedge z \geq 0\} \mathbf{procedure} \{x = y^z\}}$$

Main program proof

The loop invariant is

$$I \stackrel{\triangle}{=} s = \sum_{i=1}^{a-1} i^n \ \wedge \ 1 \leq a \leq m+1 \ \wedge \ n = 3$$

We are able to establish that it holds after the program initialization part by first utilizing our axiom of assignment three times:

$$\underbrace{\{0 = \sum_{i=1}^{a-1} i^n \wedge 1 \leq a \leq m+1 \wedge n = 3\}}_{\equiv P} s := 0\{I\}$$
$$\underbrace{\{0 = \sum_{i=1}^{a-1} i^3 \wedge 1 \leq a \leq m+1 \wedge 3 = 3\}}_{\equiv P'} n := 3\{P\}$$
$$\underbrace{\{0 = \sum_{i=1}^{1-1} i^3 \wedge 1 \leq 1 \leq m+1\}}_{\equiv P''} a := 1\{P'\}$$

Hence, the obvious composition of the above expressions yields

$$\{P''\}a := 1; n := 3; s := 0\{I\}$$

Now since $m \geq 0 \Rightarrow m+1 \geq 1$ and it is vacuously true that the sum in P'' is zero, application of our rule of consequence implies that

$$\{m \geq 0\}a := 1; n := 3; s := 0\{I\}$$

We must now verify that the invariant is maintained by the **while** loop. By the axiom of assignment, we know each of

$$\underbrace{\{s = \sum_{i=1}^{a+1-1} i^n \wedge 1 \leq a+1 \leq m+1 \wedge n = 3\}}_{\equiv Q} a := a+1\{I\}$$

$$\underbrace{\{s + p = \sum_{i=1}^{a} i^n \wedge 1 \leq a+1 \leq m+1 \wedge n = 3\}}_{\equiv Q'} s := s+p\{Q\}$$

Hence the composition of these expressions allows us to infer

$$\{Q'\}s := s+p; a := a+1\{I\}$$

We are now in a position to move our line of reasoning through the procedure call. Previously, we verified the body of our procedure e. Hence, we may use our rule of invocation as follows:

$$\frac{\{z \geq 0 \wedge y \neq 0\}\ body\ of\ e\ \{x = y^z\}}{\{z \geq 0 \wedge y \neq 0\}\ \mathbf{call}\ e(x) : (y, z)\ \{x = y^z\}}$$

Next, we can condition this expression to suit the actual call by our rule of substitution:

$$\frac{\{z \geq 0 \wedge y \neq 0\}\ \mathbf{call}\ e(x) : (y, z)\ \{x = y^z\}}{\{n \geq 0 \wedge a \neq 0\}\ \mathbf{call}\ e(p) : (a, n)\ \{p = a^n\}}$$

Finally, we must adapt the assertions in this procedure call to this program's context:

$$\frac{\{n \geq 0 \wedge a \neq 0\}\ \mathbf{call}\ e(p) : (a, n)\ \{p = a^n\}}{\{n \geq 0 \wedge a \neq 0 \wedge (\forall p, p = a^n \Rightarrow Q')\}\ \mathbf{call}\ e(p) : (a, n)\ \{Q'\}}$$

This can now be composed with the expression describing the remaining part of this loop, yielding

$$\frac{\begin{array}{l}\{n \geq 0 \wedge a \neq 0 \wedge (\forall p, p = a^n \Rightarrow Q')\}\mathbf{call}\ e(p) : (a, n)\ \{Q'\}, \\ \quad \{Q'\}s := s+p; a := a+1\{I\}\end{array}}{\begin{array}{l}\{n \geq 0 \wedge a \neq 0 \wedge (\forall p, p = a^n \Rightarrow Q')\}\mathbf{call}\ e(p) : (a, n); \\ \quad s := s+p; a := a+1\{I\}\end{array}}$$

Hence, in order to complete our verification that the invariant is maintained by the **while** loop, we must show that

$$I \wedge a \leq m \Rightarrow n \geq 0 \wedge a \neq 0 \wedge (\forall p, p = a^n \Rightarrow Q')$$

For if this is the case, then this can be used in a rule of consequence with the previous expression, showing us that $I \wedge a \leq m\{body\ of\ while\}I$. At this point we claim that the implication is true, defer its proof to the next section, and complete our current line of reasoning.

Once the invariant is shown to be maintained, then we can directly apply our **while** inference rule, yielding

$$\{I\}\mathbf{while}\ a \leq m\ \mathbf{do}\ \cdots\ \{I \wedge a > m\}$$

This can be composed with the initialization part of our main program, yielding

$$\{m \geq 0\}a := 1; n := 3; s := 0; \mathbf{while}\ \cdots\{I \wedge a > m\}$$

Since $I \wedge a > m \Rightarrow\ s = \sum_{i=1}^{m} i^3$, which is our desired output condition, then our rule of consequence implies that

$$\{m \geq 0\}\mathbf{PROGRAM}\{s = \sum_{i=1}^{m} i^3\}$$

Key step

We now return to a key step, showing that

$$I \wedge a \leq m \Rightarrow n \geq 0 \wedge a \neq 0 \wedge (\forall p, p = a^n \Rightarrow Q')$$

This can be seen by considering each of the parts of the desired consequent separately, a valid approach since it is a conjunction.

- $I \wedge a \leq m \Rightarrow n \geq 0 \wedge a \neq 0$: The invariant forces $n = 3$ which guarantees that $n \geq 0$, and likewise the invariant guarantees that $1 \leq a$ which certainly implies that $a \neq 0$.

• $I \wedge a \leq m \Rightarrow (\forall p, p = a^n \Rightarrow Q')$: Since the invariant contains no occurrences of p, the universal quantifier can be moved further outside giving us $\forall p, I \wedge a \leq m \Rightarrow (p = a^n \Rightarrow Q')$. This can be directly rewritten as

$$\forall p, I \wedge a \leq m \wedge p = a^n \Rightarrow s + p = \sum_{i=1}^{a} i^n \wedge 1 \leq a + 1 \leq m + 1 \wedge n = 3$$

(which also includes an expansion of our symbol Q.) In turn, we know that each of
a. $1 \leq a \wedge a \leq m \Rightarrow 1 \leq a + 1 \leq m + 1$
b. $n = 3 \Rightarrow n = 3$, and
c. $s = \sum_{i=1}^{a-1} i^n \wedge p = a^n \Rightarrow s + p = \sum_{i=1}^{a} i^n$

This verifies our desired implication, and the proof is complete.

Derivation of adaptation

In order to gain more insight as to why this rule of adaptation looks as it does, we provide another more general rule of inference (one not even involving procedure calls) and suggest how the earlier rule of adaptation can follow from it.

Theorem: If code body S modifies only the value of variable x and $\{P\}S\{Q\}$ is known, then $\{P \wedge \forall x(Q \Rightarrow R)\}S\{R\}$.

In order to prove this theorem, it is useful to have an informal understanding of "program state," by which we mean a symbol table listing ***all*** variables used by the program along with the corresponding values. When running a program, we execute each statement by operating on the program state to produce a new program state, that is, a symbol table where one of the entry's values might have been changed.

Proof: Say we have a state α_1 which satisfies $P \wedge \forall x(Q \Rightarrow R)$. This means that (a) α_1 makes P true, and (b) α_1 has the property that no matter what value you assign to variable x in the α_1 state, then the resulting program state still allows you to infer $Q \Rightarrow R$. Our objective is to show that R holds after execution of S; call this new state α_2. By assumption, α_2 only differs from α_1 at x, and also we know that α_2 satisfies Q. Also, since only x could have changed, α_2 still allows us to know $Q \Rightarrow R$. Hence $Q \wedge (Q \Rightarrow R)$ allow us to infer R. □

It should be clear that we can augment this proof to deal with ***lists*** of variables x, not just single values. Likewise, it should be possible to specify this proof for the case when our code S is a procedure call in particular; since procedure calls can only alter the "in-out" parameters, those are the only symbols we need include in our list x here. All that remains is to determine where the existential quantifiers come into play.

4.5. Recursion in Procedure Calls

So far we have not dealt with proofs of a procedure which calls itself (either directly or indirectly.) In order to do so, however, we require a strengthened rule of invocation. The existing rule would not allow us to prove the body of a procedure without knowing that the assertions surrounding the nested call could be proven, yet we can not verify those assertions at the call without having first proven the body!

The solution lies with induction: First we will show that the assertions surrounding a procedure ***body*** can be "conditionally" verified if the assertions at the point of the recursive call are assumed to hold. Then we will show a "base case" for induction holds as well by verifying the part of the body that does not contain any recursive call. Together, these will allow us to infer that the invocation of a recursive procedure is true by induction. Therefore, we have the **Recursive Invocation** rule:

$$\frac{\omega(x) : (v)\ \mathbf{procedure}\ S\ ,\ \{R\}\mathbf{call}\ \omega(x) : (v)\{Q\}\ \vdash \{R\}S\{Q\}}{\{R\}\mathbf{call}\ \omega(x) : (v)\{Q\}}$$

which we read "If ω is a procedure whose body is implemented by S, and if in addition we know that $\{R\}S\{Q\}$ is deducible from the assumption[3] that $\{R\}\mathbf{call}\ \omega(x) : (v)\{Q\}$, then we can infer that $\{R\}\mathbf{call}\ \omega(x) : (v)\{Q\}$." In many ways this is analogous to what motivated our development of the ***invariant*** for while loops, another inductive argument where we initially had to assume that the assertion I was true in order to show that execution of the body maintained I. Here, instead, we perform induction based on the state produced by execution of the entire procedure.

[3]Note that this is one of the few cases where we need $p \vdash q$ rather than $p \Rightarrow q$ since we are assuming that the recursive call *does* work in order to prove that the body of the call is consistent with that assumption.

4.6. Example: Simple Recursion

Consider the following simple, recursive program for obtaining sums of the form $\sum_{i=1}^{l} i$:

$\{l \geq 1\}$
PROGRAM $\equiv$ {
  $s(r) : (k)$**procedure** $\equiv$ {
    precondition $k \geq 1$
    postcondition $r = \sum_{i=1}^{k} i$
    new t;
    if $k = 1$
      then $r := 1$
      else
        $t := k - 1$;
        call $s(r) : (t)$;
        $r := r + k$}
  call $s(t) : (l)$}
$\{t = \sum_{i=1}^{l} i\}$

Proof of procedure body

Proof: Reasoning back through the body of the procedure, we first account for the base case by starting with an axiom of assignment:

$$\{1 = \sum_{i=1}^{k} i\}\ r := 1\ \{r = \sum_{i=1}^{k} i\}$$

Since $k = 1 \Rightarrow 1 = \sum_{i=1}^{k} i$ then a consequence of our above axiom is

$$\{k = 1\}\ r := 1\ \{r = \sum_{i=1}^{k} i\}$$

Next we account for the inductive step. Using assignment,

$$\{r + k = \sum_{i=1}^{k} i\}\ r := r + k\ \{r = \sum_{i=1}^{k} i\}$$

Since $r = \sum_{i=1}^{k-1} i \ \Rightarrow\ r + k = \sum_{i=1}^{k} i$, then a consequence of this is that

$$\{r = \sum_{i=1}^{k-1} i\}\ r := r + k\ \{r = \sum_{i=1}^{k} i\}$$

By our original rule of invocation, followed immediately by substitution, we can infer:

$$\frac{\{k \geq 1\}\ \mathbf{call}\ s(r) : (k)\ \{r = \sum_{i=1}^{k} i\}}{\{t \geq 1\}\ \mathbf{call}\ s(r) : (t)\{\ r = \sum_{i=1}^{t} i\}}$$

Notice that it is directly at this point that we have ***assumed*** the validity of the routine s recursively. Now, we apply our rule of adaptation to this expression, yielding

$$\underbrace{\{t \geq 1 \wedge \forall r(r = \sum_{i=1}^{t} i \Rightarrow r = \sum_{i=1}^{k-1} i)\}}_{\equiv P}\ \mathbf{call}\ s(r) : (t)\ \underbrace{\{r = \sum_{i=1}^{k-1} i\}}_{\equiv Q}$$

Hence we may use composition:

$$\frac{\{P\}\ \mathbf{call}\ s(r) : (t)\ \{Q\}\ ,\ \{Q\}\ r := r + k\ \{r = \sum_{i=1}^{k} i\}}{\{P\}\ \mathbf{call}\ s(r) : (t); r := r + k\ \{r = \sum_{i=1}^{k} i\}}$$

We now use our assignment axiom by

$$\{k - 1 \geq 1 \wedge \forall r(r = \textstyle\sum_{i=1}^{k-1} i \Rightarrow r = \sum_{i=1}^{k-1} i)\}\ t := k - 1\ \{P\}$$

which can be composed with our previous expression to yield

$$\begin{array}{l} \{k - 1 \geq 1 \wedge \forall r(r = \sum_{i=1}^{k-1} i \Rightarrow r = \sum_{i=1}^{k-1} i)\} \\ \quad t := k - 1;\ \mathbf{call}\ s\ \cdots ; r := r + k \\ \quad \{r = \sum_{i=1}^{k} i\} \end{array}$$

Observe that the universally quantified subexpression above is a tautology, hence we obtain:

$$\{k - 1 \geq 1\} t := k - 1;\ \mathbf{call}\ s\ \cdots ; r := r + k\{r = \textstyle\sum_{i=1}^{k} i\}$$

With trivial uses of rule of consequence, we can condition the above expressions to match the needs of our "conditional" inference rule, which we show applied below to join our basis and inductive cases:

$$\frac{\{k \geq 1 \wedge k = 1\}\ r := 1\ \{r = \sum_{i=1}^{k} i\}, \quad \{k \geq 1 \wedge k \neq 1\}\ t := k - 1 \cdots \{r = \sum_{i=1}^{k} i\}}{\{k \geq 1\}\ \textbf{if}\ k = 1\ \textbf{then}\ \cdots \{r = \sum_{i=1}^{k} i\}}$$

Hence we can conclude

$$\{k \geq 1\}\ \textbf{begin}\ \textit{proc body}\ \textbf{end}\ \{r = \textstyle\sum_{i=1}^{k} i\}$$

Proof of main program

By use of our rule of recursive invocation, and since above we showed that we could deduce the partial correctness of the procedure by inductively assuming the routine's correctness, then

$$\frac{s(r) : (k)\ \textbf{proc}\ \textit{proc body}\ ,\ \{k \geq 1\}\ \textbf{begin}\ \textit{proc body}\ \textbf{end}\ \{r = \sum_{i=1}^{k} i\}}{\{k \geq 1\}\ \textbf{call}\ s(r) : (k)\ \{r = \sum_{i=1}^{k} i\}}$$

By substitution, we thus know

$$\frac{\{k \geq 1\}\ \textbf{call}\ s(r) : (k)\ \{r = \sum_{i=1}^{k} i\}}{\{l \geq 1\}\ \textbf{call}\ s(t) : (l)\ \{t = \sum_{i=1}^{l} i\}}$$

which is our desired result.

□

5. EXERCISES

Problem 1. Consider the multiplication problem described in Section 1.3.

```
P ≡ {
    a := A;
    b := B;
    y := 0;
    while b > 0 do
        y := y + a
        b := b − 1}
    {y = A × B}
```

1. Show that the precondition $\{A = 0\}$ also leads to a correct proof of the program.

2. What do your answers to the above and the previously generated precondition say about the domain for A and B and for preconditions and invariants in general?

Problem 2. Consider the program

```
{a > b > 0}
    r := a mod b;
    while r > 0 do
        a := b;
        b := r;
        r := a mod b;
```

$$\{b = \mathbf{gcd}(\overleftarrow{a}, \overleftarrow{b})\}$$

The postcondition means b is the greatest common divisor of the input values a and b. Using Hoare techniques, prove the total correctness of this program.

Problem 3. Consider the following program:

```
P ≡ {
    s := sum;
    while s ≥ 10 do
        x := s div 10;
        y := s − 10 ∗ x;
        s := x + y;
    if (s mod 3) = 0 then ans := true
            else ans := false   }
```

A number is divisible by 3 if its digital sum is divisible by 3. The digital sum is reached by adding together the digits of a number, and repeating the process until the sum is less than 10. For example, $72 \rightarrow 7+2 = 9$ which is divisible by 3. But $68 \rightarrow 6+8 = 14 \rightarrow 1+4 = 5$ which is not divisible by 3. Show that the above program computes the digital sum, give the pre- and postconditions for this program and demonstrate the total correctness of this program.

Problem 4. The following program is to determine if the array $a[1..l]$ contains a palindrome.

$$
\begin{array}{l}
\mathbf{P} \equiv \{ \\
\quad p := \mathbf{true}; \\
\quad i := \mathbf{1}; \\
\quad \mathbf{while}\ (i < l/\mathbf{2} + \mathbf{1}) \wedge p\ \mathbf{do} \\
\quad\quad \mathbf{if}\ a[i] = a[l - i + \mathbf{1}] \\
\quad\quad\quad \mathbf{then}\ i := i + \mathbf{1} \\
\quad\quad\quad \mathbf{else}\ p := \mathbf{false}\}
\end{array}
$$

Give input and output specifications, and verify the partial correctness of this program.

Problem 5. Demonstrate the partial correctness of the following program:

$$
\begin{array}{l}
\{n \geq \mathbf{1}\ \wedge\ (\forall k, \mathbf{0} \leq k < n, a[k] \leq a[k+\mathbf{1}])\ \wedge\ a[\mathbf{0}] \leq x < a[n]\} \\
\mathbf{PROGRAM} \equiv \{ \\
\quad i := \mathbf{0}; \\
\quad j := n; \\
\quad \mathbf{while}\ i + \mathbf{1} \neq j\ \mathbf{do} \\
\quad\quad h := (i + j)/\mathbf{2}; \\
\quad\quad \mathbf{if}\ a[h] \leq x \\
\quad\quad\quad \mathbf{then}\ i := h \\
\quad\quad\quad \mathbf{else}\ j := h\} \\
\{(a[i] \leq x < a[i+\mathbf{1}]) \wedge (\forall k, \mathbf{0} \leq k < n, a[k] \leq a[k+\mathbf{1}])\}
\end{array}
$$

Problem 6. Consider the program in Section 3.1. Change the assignment statement to the following:

$$x[k] := x[k-1]$$

What does the program do? Give the postcondition and demonstrate the correctness of the program to that postcondition.

Problem 7. Demonstrate the partial correctness of the following program:

$$
\begin{array}{l}
\{n > 0\} \\
\textbf{PROGRAM} \equiv \{ \\
\quad k := 0; \\
\quad x[0] := 1; \\
\quad \textbf{while } k < n \textbf{ do} \\
\quad\quad k := k + 1; \\
\quad\quad x[k] := k * x[k-1]; \} \\
\{\forall i, 0 \le i \le n, x[i] = i!\}
\end{array}
$$

Problem 8. Using Hoare's method, prove the partial correctness of the following program fragment:

$$
\begin{array}{l}
\{N \ge 0\} \\
I := 1; \\
\textbf{while } I \le N \textbf{ do} \\
\quad \textbf{if } V[I] = E \textbf{ then} \\
\quad\quad \textbf{if } N > 1 \textbf{ then } V[I] := V[N]; \\
\quad\quad N := N - 1 \\
\quad \textbf{else } I := I + 1 \\
\{\forall k,\ 1 \le k \le N,\ V[k] \ne E\}
\end{array}
$$

Problem 9. The following code fragment merges two sorted arrays a and b to produce a new sorted array c. Verify that the elements of c appear in ascending order.

```
k := 0;
i := 1;
j := 1;
while (i ≤ m) and (j ≤ n) do begin
      k := k + 1;
      if a[i] ≤ b[j] then begin
            c[k] := a[i];
            i := i + 1;
            end
      else begin
            c[k] := b[j];
            j := j + 1;
            end
      end
```

Problem 10. Prove the partial correctness of the following procedure:

```
{(x > 0 ∧ y ≥ 0)}
q(a) : (x, y)procedure ≡ {
      if x > y
            then a := (y = 0)
            else if x = y
                  then a := true
                  else i := y − x; q(a) : (x, i)}
{a = (mod(y, x) = 0)}
```

Problem 11. Consider the program

```
{n ≥ 1}
z(a) : (i, n)procedure ≡ {
      precondition : i ≤ n
      postcondition : ∀j, i ≤ j ≤ n, a[j] = 0
      new m
            if i ≠ n then
                  m := i + 1;
                  call z(a) : (m, n);
            a[i] := 0}
j := 0;
call z(a) : (j, n)
{∀i, 1 ≤ i ≤ n|a[i] = 0}
```

Using Hoare techniques and assuming that the array a is everywhere defined, prove the partial correctness of this program.

Problem 12. Consider the procedure

$$
\begin{array}{l}
max(x,a,i):(n)\textbf{procedure} \equiv \{ \\
\quad \textbf{if } n = i \textbf{ then} \\
\quad\quad x \ := \ a[i] \\
\quad \textbf{else} \\
\quad\quad i \ := \ i+1; \\
\quad\quad \textbf{call } max(x,a,i):(n); \\
\quad\quad i \ := \ i-1; \\
\quad\quad \textbf{if } a[i] > x \textbf{ then } x \ := \ a[i]\}
\end{array}
$$

Provide reasonable pre- and postconditions to this procedure, and, using Hoare techniques, verify the partial correctness of this procedure with respect to those pre- and postconditions.

Problem 13. Use the ***max*** procedure and your pre- and postconditions for it from the previous problem, and provide a precondition which allows you to prove the partial correctness of the main program:

$$
\begin{array}{l}
i \ := \ 2 \\
\textbf{call } max(y,b,i):(n)
\end{array}
$$

where b is an array of values indexed from 1 through n, and the postcondition is

$$y = maximum\{b[j] : 1 \le j \le n\}$$

Problem 14. Consider the procedure

$\{1 \le i \le n\}$
$swap(a, i) : (n)$**procedure** $\equiv$
    begin
    new t
    $t := a[i];$
    $a[i] := a[n - i + 1];$
    $a[n - i + 1] := t$
    end
$\{a[i] = \overleftarrow{a}[n - i + 1] \ \wedge \ a[n - i + 1] = \overleftarrow{a}[i]\}$

Verify the partial correctness of this procedure with respect to the pre- and postconditions.

Problem 15. The following program reverses the elements of the integer array $a[1..n]$. As usual, assume the values of n and all array elements are set before the program begins execution. Write the appropriate pre- and postconditions, and prove the partial correctness of this program.

```
procedure shift(var b : array of integer; i : integer);
    new j
    {for j := 1 to i do b[j] := b[j + 1]}
PROGRAM ≡ {
    i := n − 1;
    while i > 0 do
        t := a[1];
        shift(a, i);
        a[i + 1] := t;
        i := i − 1}
```

Problem 16. Verify the partial correctness of the following program:

```
{n ≥ 1}
     i := 2;
     while i ≤ n do
          Prime[i] := true;
          i := i + 1
     i := 2;
     while i ≤ n do
          j := 2i;
          while j ≤ n do
               Prime[j] := false;
               j := j + i
          i := i + 1
{∀ 2 ≤ i ≤ n(¬PRIME[i] ⇔ ∃ 2 ≤ j, k ≤ n, i = kj)}
```

Problem 17. Prove the partial correctness of the following recursive procedure.

```
procedurep(var x : integer);
/ * pre : x ≥ 0 ∧ x = x0; post : x = 2 * x0 * /
begin
     if x ≠ 0 then begin
          x := x − 1;
          p(x);
          x := x + 2
          end
end;
```

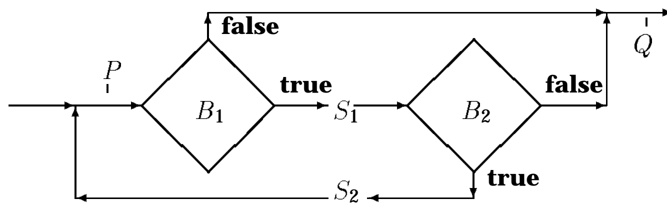
Problem 18. Write an appropriate precondition and postcondition for the following recursive procedure. Verify the procedure's implementation.

```
procedure p(var x : t; s, f : integer);
begin
     if s ≤ f then begin
          x[s] := 0;
          p(x, s + 1, f)
     end
end;
```

Problem 19. First, write a rule of inference for the **repeat**... **while** language construct. Likewise, make one for **repeat**... **until**. Then write a rule of inference to describe the semantics of the construct

while B_1 **do** S_1 **exit** B_2 **;** S_2 **od**

which corresponds to the following flowchart:



In each case, present an argument that will support your choice of inference rules.

Problem 20. Consider the language we have modeled using Hoare techniques so far in class. We now wish to extend this language to include a **return** statement within procedures, and of course we will need a corresponding rule of inference to model this new language construct.

- Derive an inference rule to model the semantics of a **return** statement.

- Use the inference rule you developed above to prove the partial correctness of the following procedure:

$\{\textbf{true}\}$
$p(i) : ()\textbf{procedure} \equiv \{$
  $i := 7;$
  $\textbf{return};$
  $i := 8\}$
$\textbf{program} \equiv \{$
  $\textbf{call}\ p(i) : ()\}$
$\{i = 7\}$

6. SUGGESTED READINGS

The original work on this axiomatic approach is due to Floyd and Hoare, and entries in the following list ought to be viewed as "classics."

• R.W. Floyd, "Assigning Meanings to Programs," *Symposium in Applied Mathematics* 19, American Mathematical Society, 1967, pp. 19-32.

• C.A.R. Hoare, "An Axiomatic Basic for Computer Programming," *Communications of the ACM*, Vol. 12, No. 10, October, 1969, pp. 576-580, 583.

• C.A.R. Hoare, "Procedures and Parameters, An Axiomatic Approach," *Symposium on the Semantics of Algorithmic Languages*, Springer-Verlag, New York, 1971, pp. 102-116.

• C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," *Acta Informatica*, Vol. 2, 1973, pp. 335-355.

• D.C. Luckham and N. Suzuki, "Verification of Array, Record, and Pointer Operations in Pascal," *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 2, October, 1979, pp. 226-244.

Readers will find these classics nicely augmented by these works:

• J.H. Remmers, "A Technique for Developing Loop Invariants," *Information Processing Letters*, Vol. 18, 1984, pp. 137-139.

• J.C. Reynolds, *The Craft of Computer Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

Examples of more recent work that are related to the original work of Floyd and Hoare (either applying it or extending it) are found in:

• L. Lamport and F.B. Schneider, "The 'Hoare Logic' of CSP, and All That," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 2, April 1984, pp. 281-296.

• D. Gries and G. Levin, "Assignment and Procedure Call Proof Rules," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 4, October 1980, pp. 564-579.

Functional Correctness*

Chapter 3

.

Let α be a string representing a source program. For example, a Pascal program is just the linear string:

$$PROGRAM\ main(input, output); \cdots END.$$

We express the mathematical function denoted by program α by a *box* notation. $\boxed{\alpha}$ represents the function that computes the same values as program α.

Although a function is the intuitive model of a specification, often we simply want one feasible solution out of many possibilities. For example, by choosing one optimal strategy among several equivalent ones (e.g., equivalent optimal moves in a game-playing program), we usually do not care which solution the program employs. Because of this, we only need to define a specification as a *relation*. If r is such a specification relation, it is equivalent to a program p by the following *correctness* theorem:

Theorem: Program p is correct with respect to specification relation r if and only if $domain(r \cap \boxed{p}) = domain(r)$.

*Parts of this chapter are a revision of an earlier paper [64]. Copyright ©1993 by Academic Press, Inc. Reprinted by permission of Academic Press.

In other words, if we take the subset of r by considering those pairs in r that are also in function $\boxed{p}$ (i.e., $r \cap \boxed{p}$) we have a function. If this function has the same domain as r, then $\boxed{p}$ includes a pair of values for each member of relation r and we get a feasible (or correct) implementation of the specification. In what follows, however, we will use the simpler case, where we have chosen the more restricted specification function f instead of the more general relation r with the corresponding correctness theorem of $f \subset \boxed{p}$.

1. PROGRAM SEMANTICS

A program is a sequence of declarations followed by a sequence of statements. Every program maps a set of values for every variable into a new set of values. We pattern the following development on techniques also used in denotational semantics (which we will discuss in greater detail in Chapter6). We can define the meaning of such a program as follows.

id.Id is a primitive set of objects we will call the *variables* or *identifiers* in a program.

value.$Value$ is a primitive set of values that each member of id can take. We will assume that $value$ is the set of $integers$ for most of this discussion, but it can be any discrete set of values.

state. The primitive storage mechanism is the $state$. If id is a set of variable names and $value$ is a set of values, a $state$ is a function with the signature $state : id \rightarrow value$. For every member of set id, we associate a value in $value$.

Since a $state$ is a function, it is a set of ordered pairs. Thus for state s we can discuss $(a, b) \in s$, or $s : id \rightarrow value$ and $s(a) = b$.

Given state s, if a is a variable in a program, we define $\boxed{a}(s)$ as $s(a)$.

This concept of a $state$ represents the formal model for program storage (e.g., activation records).

expressions. If $expr$ is an expression, $\boxed{expr}$ is the function that accesses the current state function and evaluates the expression (i.e., "returns its value"). More formally, we state that $\boxed{expr} : state \rightarrow value$.

We typically can define the $expr$ function via the BNF for the language. Consider the BNF rule:

$$< expr > \rightarrow < expr > + < term > \mid < term >$$

If (x, a) and (y, b) represent entries in the state function s representing variables x and y, then we define:

$$\boxed{x + y}(S) =$$
$$\boxed{x}(S) + \boxed{y}(S) =$$
$$S(x) + S(y) = a + b$$

statements. If P is a statement, then $\boxed{P}$ is a function that maps a $state$ into a $state$, that is, each statement maps a set of values for all variables into a new set of values. If P is a declaration, then the resulting state includes a $(id, value)$ pair for the newly declared variable. For example, if s is the state $\{(x, 1)\ ,\ (y, 2), \cdots\}$, then the function $\boxed{y := x}$ when applied to s results in the state $\{(x, 1)\ ,\ (y, 1)\}$.

statement execution. It is easy to see the correspondence between sequential execution and function composition. If p is the sequence $p_1; p_2; \ldots p_n$ of statements, then

$$\boxed{p} = \boxed{p_1; p_2; \ldots p_n} = \boxed{p_1} \circ \boxed{p_2} \circ \ldots \boxed{p_n} = p_n \ \ldots \ p_2(\ p_1()\ ) \ldots)$$

All that is needed to complete this definition is the primitive $\boxed{}$ for each required statement type.

Consider Pascal. The function $\boxed{p}$ for program:

$$PROGRAM\ MAIN(INPUT, OUTPUT); BEGIN\ S_1; S_2; \cdots END.$$

is given by: $\boxed{PROGRAM\ MAIN(INPUT, OUTPUT)} \circ \boxed{S_1} \circ \boxed{S_2} \circ \ldots \boxed{\cdot}$ where the signature for $\boxed{PROGRAM \ldots}$ is $value \rightarrow state$, for $\boxed{\cdot}$ it is $state \rightarrow value$, and $state \rightarrow state$ for all other statements. Hence a program maps a value to a value and is composed of functions that map states to states. Details of how to handle individual statement types like assignments, conditionals and iteration are given later.

Developing a program requires several separate activities:

1. Development of a specification that expresses the task to be performed.

2. Refinement of that specification into a formal explicit statement that captures the intended functionality of the specification; and

3. Development of a program that correctly implements that functionality.

Most of this chapter is concerned with the transition between these last two steps. The transition between steps 1 and 2 is admittedly difficult and some heuristics will be given.

With this notation, there are three separate activities that can be investigated:

1. If f is a function and if p is a program, show $\boxed{p} = f$ (i.e., verification).

2. If f is a function, develop program p such that $\boxed{p} = f$ (i.e., program design). As a practical matter, we only care that $f \subset \boxed{p}$ since any value in $\boxed{p}$ and not in f represents a value computed by the program which is outside of its specifications and not of interest to us.

3. If p is a program, then find a function f such that $\boxed{p} = f$ (i.e., reverse engineering). Given a program, determine its specifications. Some heuristics are given, but the basic method is to "guess" a solution and show by Rules 1 and 2 above that it is the correct solution.

2. SYMBOLIC EXECUTION

Symbolic execution is an aid in showing functional composition. In order to show that $\boxed{p} = f$, symbolically execute program p and show that the resulting function is the same as f. For example, consider the sequence:

$$x := x + 1;$$
$$y := x + y;$$
$$x := y + 1$$

Since we know that

$\boxed{\text{x:=x+1; y:=x+y; x:=y+1}} = \boxed{\text{x:=x+1}} \circ \boxed{\text{y:=x+y}} \circ \boxed{\text{x:=y+1}}$

we can symbolically execute each statement function. We use a trace table where we write under **Part** the relevant statement function, and under each relevant variable the new value that results from that execution by substituting into the statement function the current value that each variable has at that point in the computation. This results in a new function that can transform each variable into its new value. For the above, we get the trace table:

Part	**x**	**y**
$x := x + 1$	$x + 1$	
$y := x + y$		$(x + 1) + y$
$x := y + 1$	$(x + 1 + y) + 1$ = x+y+2	

This states that x is transformed by the function $\boxed{x := x + y + 2}$ and simultaneously y is transformed by $\boxed{y := x + y + 1}$.

The extension of the trace table to handle conditionals (e.g., **if** statements) requires the use of a **condition** column. We write the predicate that must be true at that point in the computation for that execution path to proceed and develop trace tables for each such path through the program.

For example, the program sequence

```
x := x + y;
if x > y then
    x := x − 1
```

has two possible execution sequences (i.e., $x > y$ and $x \leq y$), and two corresponding traces:

Part	Condition	x	y
$x := x + y$		$x + y$	
$if\ x > y$	$(x + y) > y$		
$x := x - 1$		$(x + y) - 1$	

and

Part	Condition	x	y
$x := x + y$		$x + y$	
$if\ x > y$	$(x + y) \leq y$		

We summarize these two tables by saying that the function they represent is: "if $x+y > y$, then the function is $\boxed{x := x + y - 1}$ and if $x+y\ \leq y$ then the function is $\boxed{x := x + y}$." In the next section we show how to write this as a conditional assignment function.

3. DESIGN RULES

As given earlier, software development consists of (a) specification design; (b) formalizing the specification; and (c) developing the source program. We use a functional notation for (b) that is closely tied to the eventual source program. This notation includes (a) concurrent assignment; (b) conditional assignment; and (c) loop verification. This notation was strongly influenced by the earlier work by McCarthy on LISP.

3.1. Design of Assignment Statements

Concurrent assignment is defined as simultaneous assignment. The function: $(x, y, z := y, z, x)$ simultaneously accesses the current values of variables y, z, and x and stores them, respectively, into variables x, y, and z. Mathematically, we are stating that the state function that results will have the same values for all state variables other than x, y, and z, and that these three will have new values.

Given statement p, showing that $\boxed{p}$ does implement this concurrent assignment is simply a matter of building its trace table. The more

interesting problem is how to develop p given some concurrent assignment as its specification. This leads to three design heuristics for concurrent assignment:

1. All values on the right side of the intended concurrent assignment (i.e., all values that are needed by a lefthand side variable) must be computable at each step.

2. At each step, if a variable can be assigned its intended value, do so; otherwise introduce a temporary variable and assign to it a value that must be preserved.

3. Stop when all variables on the left side of the intended concurrent assignment have been assigned their intended values (i.e., finished).

If we "execute" a trace table as we develop each assignment statement, then we are also verifying that the design works as we wish. Once the values in the trace table are the values we desire, then we have shown that the assignment statements we have written do indeed implement the intended concurrent assignment.

An important point to remember is that the three design rules are heuristics and not an algorithm. They indicate how to search for a solution and how to check if the solution is correct. They do not give the solution. We have not replaced the art of programming by an implementable methodology to automatically build correct programs from specifications.

3.2. Design of Conditional Statements

Using the conditional function, we can represent the design of conditional statements as a series of **if** statements that test each condition in turn. For example, given the specification:

$$(b_1 \to c_1)|(b_2 \to c_2)| \ldots |(b_n \to c_n)$$

the program can be written directly as:

if b_1 **then** c_1
else **if** b_2 **then** c_2
else **if** b_3 **then** c_3
...

If all the b_i are false, since the code is everywhere defined, the specifications are actually a (correct) subset of this source program.

3.3. Verification of Assignment and Conditional Statements

Assume p is the program to be verified and it consists of only **if** and assignment statements. There are only a finite number of execution paths through the program. For each path, compute the condition that must be true to execute that path and use a trace table to determine what happens to the variables by executing that given path. Assume $p_1, p_2, \cdots$ are the conjunctions of all conditions on each execution path, and $a_1, a_2, \cdots$ are the corresponding concurrent assignments. The function f that this implements is:

$$(p_1 \rightarrow a_1)|(p_2 \rightarrow a_2)| \cdots |(p_n \rightarrow a_n)$$

Consider the following example:

$$\begin{array}{ll} x := x + y & \{1\} \\ y := y - x; & \{2\} \\ \textbf{if } x + y > 0 \textbf{ then} & \\ \quad y := x + y & \{3\} \\ \quad \text{else} & \\ \quad y := -x - y & \{4\} \end{array}$$

This has two execution sequences, 1-2-3 and 1-2-4, with two different traces:

if is true

Part	Condition	x	y
$x := x + y$		$x + y$	y
$y := y - x$		$x + y$	$-x$
$if\ x + y > 0$	$(x + y) - x > 0$		
$y := x + y$			$x + y - x =$ y

if is false – so ¬ (if) is true

Part	**Condition**	**x**	**y**
$x := x + y$		$x + y$	y
$y := y - x$		$x + y$	$-x$
$if\ x + y > 0$	$(x + y) - x \leq 0$		
$y := -x - y$			$-(-x) - (x + y) = -y$

This gives the function:

$$(y > 0 \rightarrow x, y := x + y, y)|(y \leq 0 \rightarrow x, y := x + y, -y)$$

or since the assignment to y (i.e., $(y > 0 \rightarrow y := y)$ and $(y \leq 0 \rightarrow y := -y)$) is just function $abs(y)$, the function reduces to: $(x, y := x + y, abs(y))$.

While we could have left our answer as a conditional assignment, replacing it as a concurrent assignment using the absolute value function leads to a more understandable solution. Knowing when (and how) to apply such reductions is probably as complex an issue as any encountered in axiomatic verification.

4. SEMANTICS OF STATEMENTS

We have given the semantics for state transition and have described a functional notation for conditionals. We must now give the semantics for each statement type in order to fully describe the meaning for a program.

4.1. Begin Blocks

The **begin** statement adds no semantics to a program; therefore we will define the **begin** block as follows:

$$\boxed{\text{begin S end}} = \boxed{\text{S}}$$

4.2. Assignment Statement

The assignment statement has a meaning similar to the assignment axiom in Chapter 2.

$$\boxed{x := y}(s) = \{(i,j) \mid (i,j) \in s \wedge i \neq x\} \cup \{(x,y)\}$$

We can also write it as:

$$\boxed{x := y} = \{(s,s') \mid (i \neq x \rightarrow s'(i) = s(i)) \mid (i = x \rightarrow s'(x) = \boxed{y}(s)\}$$

4.3. If Statement

The **if** statement simply traces both possible paths through the program:

$$\begin{aligned} \boxed{if\ b\ then\ s_1\ else\ s_2} &= (\boxed{b} \rightarrow s_1) \mid (\neg\boxed{b} \rightarrow s_2) \\ \boxed{if\ b\ then\ s_1} &= (\boxed{b} \rightarrow s_1) \mid () \end{aligned}$$

Remember for the **if** - **then**, you need the identity alternative to make sure that the function is total.

4.4. While Statement

In order to handle full program functionality, we must address loops. Given a functional description f and a **while** statement p, we first describe three verification rules that prove that f and p are equivalent. These will be denoted as **V.I-III**. Once we have these verification conditions, we would like to use them as design guidelines to help develop p, given only f. We call these five design rules **B.I-V**.

The **while** statement $\boxed{while\ b\ do\ d}$ is defined recursively via the **if** to mean:

$$\boxed{while\ b\ do\ d} = \boxed{if\ b\ then\ begin\ d; while\ b\ do\ d\ end}$$

That is, if b is true, perform d and repeat the **while**. Via a simple trace table we get the same result as:

$$\begin{aligned} \boxed{while\ b\ do\ d} &= \boxed{if\ b\ then\ d; while\ b\ do\ d} \qquad (**) \\ &= \boxed{if\ b\ then\ d} \circ \boxed{while\ b\ do\ d} \end{aligned}$$

Let f be the meaning of the **while** statement, that is, $f = \boxed{while\ b\ do\ d}$. By substituting back into (**) above, we get the first condition that:

V.I$f = \boxed{if\ b\ then\ d} \circ f$

What other conditions on f insure that it is indeed the specification of the **while** statement? If f is undefined for some input x, then both sides of the equation are undefined. In order to insure that this cannot happen, we require that f is defined whenever $\boxed{while}$ is defined, or that $domain(\boxed{while}) \subset domain(f)$. (Note: For ease in reading, we will use $\boxed{while}$ to stand for $\boxed{while\ b\ do\ d}$.)

Similarly, if $\boxed{while}$ is everywhere the identity function then any f will fulfill the equation, because, if $\boxed{while}$ is everywhere the identity, then similarly $\boxed{if\ b\ then\ d}$ must also be the identity and the recursive equation reduces to $f = () \circ f = f$. Thus we must also have $domain(f) \subset domain(\boxed{while})$. This yields:

V.II$domain(f) = domain(\boxed{while\ b\ do\ d})$

Consider any state $s \in domain(\boxed{while})$. If $\boxed{b}(s)$ is true, that is, expression b in state s is true, then from (**), $s_1 = \boxed{d}(s)$ and $s_1 \in domain(\boxed{while})$. This will be true, for $s_2, s_3, \cdots$ until at some point, $\boxed{b}(s_n)$ is false and both $\boxed{if\ b\ then\ d}(s_n)$ and $\boxed{while\ b\ do\ d}(s_n)$ equal s_n.

This s_n is a member of $domain(\boxed{while})$ and of $range(\boxed{while})$. More importantly, if $\boxed{b}(s)$ evaluates to false, then $\boxed{while}(s) = s$, or stated another way, $\boxed{while}(s) = s$ for all states s where $\boxed{b}(s)$ is false. This is just a restriction on the $\boxed{while}$ function to those states where b is false, which is the function $(\neg(b) \rightarrow \boxed{while})$. This must be equal to the identity function () also restricted to the same domain, or just: $(\neg(b) \rightarrow ()\ )$. Any candidate function f must also have this property, yielding the third constraint:

V.III$(\neg b \rightarrow f) = (\neg b \rightarrow ())$

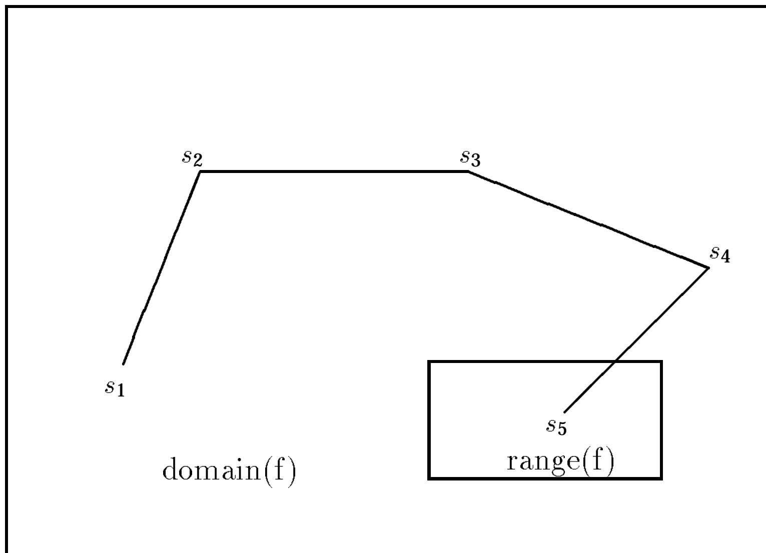


Figure 3.1. Domain and range of while function

Design of while loops

Consider the problem of designing a loop. Given a specification f, how can we design a **while** from the three statement verification conditions given above?

From V.III, the **while** terminates when $\boxed{b}$ evaluates to false, and $range(\boxed{while})$ is just the set of states where $\boxed{b}$ is false. But since we can apply $\boxed{while}$ to this state initially, it is also part of $domain(\boxed{while})$. Therefore, $range(\boxed{while}) \subset domain(\boxed{while})$ (see Figure 3.1). Since f must also have this property, we get:

D.I$range(f) \subset domain(f)$.

Similarly, we have shown that for an f where $\boxed{b}(s)$ is false, $\boxed{while}(s) = s$, we must also have $f(s) = s$, because if $\boxed{\text{b}}$(s) is false, the body d is not executed. But these are just the points in range(f). Therefore, we get the second design constraint:

D.II $if\ s \in range(f), then\ f(s) = s.$

D.I and D.II must be true if f is the meaning of a **while** statement. Therefore they show the existence of a possible solution. From D.II, we know f must be an identity on $range(f)$ in order to be implemented with a **while**. We can restate this as:

D.III $\boxed{b}$ evaluates to true in $domain(f) - range(f)$ and false in $range(f)$.

Similarly to the assignment design, we develop the **while** loop via:

- **D.IV** Develop d so that all values are preserved for f.
- **D.V** Show that f is everywhere defined, that is, the loop must terminate for all $x \in domain(f)$.

Given function f we develop a **while** statement such that $\boxed{while} = f$ as follows:

- **Existence.** Verify conditions D.I and D.II. If these cannot be satisfied, then no such **while** statement can be written.
- **Range determination.** Use D.III to develop some predicate b such that $\boxed{b}$ is false on $range(f)$ and true on $domain(f) - range(f)$. Since f and $\boxed{while}$ are to be the same function, b becomes the predicate for the loop.
- **Loop body.** Use D.IV to develop an appropriate d. These guidelines do not give absolute solutions to this problem but do indicate how to verify whether the d is a solution once found. It is comparable to finding the loop invariant in an axiomatic proof.
- **Termination.** Prove that the selected b and d causes the loop to terminate (condition D.V.). If proven that the loop terminates, and since the second step shows that $\boxed{b}(x)$ is false for $x \in range(f)$, this shows that the loop will terminate with some x in this range.

5. USE OF FUNCTIONAL MODEL

5.1. Example: Verification

Show that the function $f = (A \le B \rightarrow A, B := (B - (B - A)/2), (B - (B - A)/2)|())$ **is implemented by the source program:**

```
1  while A < B do
2      begin
3      A := A + 1;
4      if A < B then
5          B := B - 1
6          end
```

where A and B are integers and division means integer truncated division (e.g., $1/2 = 0$**).**

The approach we follow is to first determine the functionality of the assignment statement (line 3), then the **if** statement (lines 4-5), then the entire **begin** block (lines 2-6) and finally the functionality of the entire segment (lines 1-6).

1. **Line 3.** $A := A + 1$.

This is just the concurrent assignment: $(A := A + 1)$.

2. **Lines 4-5.** $d_1 = \boxed{if\ A < B\ then\ B := B - 1}$

If $A < B$ is true, evaluate the function $B := B - 1$, and if it is false, skip the **then** statement and do nothing, e.g., the *Identity* function. The conditional assignment can be written as:

$$d_1 = (A < B \rightarrow B := B - 1)|()$$

3. **Lines 2-6.** $d_2 = \boxed{begin\ A := A + 1; if\ A < B\ then\ B := B - 1\ end}$

$$\begin{aligned} d_2 = &\ \boxed{\text{A:=A+1}} \circ \boxed{\text{if A<B then B:=B-1}} \\ = &\ (A := A + 1) \circ d_1 \\ = &\ (A := A + 1) \circ ((A < B \rightarrow B := B - 1)|()) \end{aligned}$$

Develop a trace table for the **begin** block. There will be two paths through this block (e.g., first alternative for d_1 and second), and hence there will be two trace tables:

Part	Condition	A	B
$3 : A := A + 1$		$A + 1$	
$4 : if\ A < B$	$(A + 1) < B$		
$5 : B := B - 1$			$B - 1$

and

Part	Condition	A	B
$3 : A := A + 1$		$A + 1$	
$4 : if\ A < B$	$(A + 1) \geq B$		

We then get: $d_2 = (A + 1 < B \rightarrow A, B := A + 1, B - 1)|(A := A + 1)$

4. **Lines 1-6** Show

$$f = \boxed{while\ A < B\ do\ begin\ A := A + 1; if\ A < B\ then\ B := B - 1\ end}$$

To accomplish this, we must show that function f meets the three **V** constraints of Section 4.4. We will do this in the order V.III, V.II, and V.I.

1. **Show (V.III) (**$\neg(A < B) \rightarrow f) = (\neg(A < B) \rightarrow ())$

$$
\begin{aligned}
&(\neg(A < B) \rightarrow f) = \\
&(A \geq B \rightarrow (A \leq B \rightarrow A, B := B - (B - A)/2, B - (B - A)/2)|()) = \\
&(A \geq B \wedge A \leq B \rightarrow (A, B := B - (B - A)/2, B - (B - A)/2)) \\
&\qquad |(A \geq B \rightarrow ()) = \\
&(A = B \rightarrow A, B := B - (B - A)/2, B - (B - A)/2)|(A \geq B \rightarrow ()) = \\
&(A = B \rightarrow A, B := A, B)|(A \geq B \rightarrow ()) = \\
&(A \geq B \rightarrow ())
\end{aligned}
$$

2. **Show (V.II)** $domain(f) = domain(\boxed{while})$

f is defined for all A and B. For $A \leq B$, an explicit assignment is given, and for all other A and B, f is the identity function.

The $\boxed{while}$ function is also defined for all A and B. If $A \geq B$, the body of the **while** does not execute giving the identity function for such A and B. If $A < B$, then for each pass through the loop, A is increased by 1 and B may be decremented by 1. At some point, $B - A$ must reach 0 or become negative. If $B - A \leq 0$, then $B \leq A$ and the **while** loop terminates. So for all A and B, the **while** statement must terminate and will generate some value for A and B.

3. **Show (V.I)** $f = \boxed{if\ b\ then\ d} \circ f$

The meaning of the body of the **if** statement (*4*) is the previously defined function:

$$d_2 = (A + 1 < B \rightarrow A, B := A + 1, B - 1)|(A := A + 1)$$

The problem then reduces to showing that:

$$f = \boxed{if\ A < B\ then\ (A + 1 < B \rightarrow A, B := A + 1, B - 1)|(A := A + 1)}$$
$$\circ((A \leq B \rightarrow A, B := B - (B - A)/2, B - (B - A)/2)|())$$

In order to show this, we need to generate the set of functions that represent each separate path through each possible trace table. If we let c_1 be the **if** expression $A < B$, c_2 be $A + 1 < B$ in d_2, and c_3 be $A \leq B$ in f, then there are six possible paths through this function yielding six different trace tables, each deriving a different function g_i:

c_1	c_2	c_3	**function**
true	true	true	g_1
true	false	true	g_2
true	true	false	g_3
true	false	false	g_4
false	–	true	g_5
false	–	false	g_6

(Note, if c_1 is false, then d_2 is not evaluated giving only six possibilities rather than the full complement of eight that normally occurs with three predicates.) We need to show that: $f = g_1|g_2|g_3|g_4|g_5|g_6$

1. g_1

Part	Condition	A	B
c_1 *true*	$A < B$		
c_2 *true*	$A + 1 < B$	$A + 1$	$B - 1$
c_3 *true*	$A + 1$ $\leq B - 1$	$B - 1$ $-(B - 1 - (A + 1))/2$ $= B - (B - A)/2$	$B - 1$ $-(B - 1 - (A + 1))/2$ $= B - (B - A)/2$

The resulting predicate $(A < B) \wedge (A + 1 < B) \wedge (A + 1 \leq B - 1)$ reduces to $A < B - 1$.

$$g_1 = (A < B - 1 \rightarrow A, B := B - (B - A)/2, B - (B - A)/2)$$

2. g_2

Part	Condition	A	B
c_1 *true*	$A < B$		
c_2 *false*	$A + 1 \geq B$	$A + 1$	
c_3 *true*	$A + 1 \leq B$	$B - (B - (A + 1))/2$	$B - (B - (A + 1))/2$

The resulting predicate is: $(A < B) \wedge (A + 1 \geq B) \wedge (A + 1 \leq B)$, which reduces to $A = B - 1$. By substituting $B - 1$ for A, we get:

$$\begin{aligned} g_2 = (&A = B - 1 \rightarrow A, B := \\ &B - (B - (A + 1))/2, B - (B - (A + 1))/2) \\ = (&A = B - 1 \rightarrow A, B := B - (B - (B - 1 + 1))/2, \\ &B - (B - (B - 1 + 1))/2) \\ = (&A = B - 1 \rightarrow A, B := B, B) \end{aligned}$$

However, if $A = B - 1$, then $(B - A)/2 = 0$. Thus we can write g_2 as:

$$g_2 = \quad (A = B - 1 \rightarrow A, B := B - (B - A)/2, B - (B - A)/2)$$

3. g_3

Part	**Condition**	**A**	**B**
c_1 is true	$A < B$		
c_2 is true	$A + 1 < B$	$A + 1$	$B - 1$
c_3 is false	$A + 1 > B - 1$		

This leads to the condition $(A < B) \wedge (A + 1 < B) \wedge (A + 1) > (B - 1)$. We get $(A < B - 1)$ and $(A > B - 2)$ which is the null function.

4. **(4)** g_4

Part	**Condition**	**A**	**B**
c_1 is true	$A < B$		
c_2 is false	$A + 1 \geq B$	$A + 1$	
c_3 is false	$A + 1 > B$		

The resulting condition is $(A < B) \wedge (A + 1 \geq B) \wedge (A + 1 > B)$. But $(A < B)$ and $(A + 1 > B)$ are mutually disjoint, making g_4 null.

5. **(5)** g_5

Part	**Condition**	**A**	**B**
c_1 is false	$A \geq B$		
c_3 is true	$A \leq B$	$B-(B-A)/2$	$B-(B-A)/2$

The resulting condition is $(A \geq B) \wedge (A \leq B)$ or $A = B$. For $A = B$, $B-(B-A)/2 = B = A$.

$$g_5 = (A = B \rightarrow A, B := A, B)$$

6. **(6)** g_6

Part	**Condition**	**A**	**B**
c_1 is false	$A \geq B$		
c_3 is false	$A > B$	A	B

The resulting condition is $(A \geq B) \wedge (A > B)$ or just $(A > B)$.

$$g_6 = (A > B \rightarrow A, B := A, B)$$

Show $f = g_1|g_2|g_3|g_4|g_5|g_6$

Since g_3 and g_4 are null, we have to show that $f = g_1|g_2|g_5|g_6$.

$$\begin{aligned}
&g_1 \mid g_2 \mid g_5 \mid g_6 = \\
&\quad (A < B-1 \rightarrow A, B := B-(B-A)/2, B-(B-A)/2) \mid \\
&\quad (A = B-1 \rightarrow A, B := B-(B-A)/2, B-(B-A)/2) \mid \\
&\quad (A = B \rightarrow A, B := A, B) \mid \\
&\quad (A > B \rightarrow A, B := A, B)
\end{aligned}$$

The first 2 terms reduce to:

$$(A < B \rightarrow A, B := B-(B-A)/2, B-(B-A)/2)$$

for $A = B$, the third term becomes:

$$\begin{aligned}
&(A = B \rightarrow A, B := A, B) = \\
&(A = B \rightarrow A, B := B-(B-A)/2, B-(B-A)/2)
\end{aligned}$$

and the last is:

$$(A > B \rightarrow A, B := A, B) = (A > B \rightarrow ())$$

We have therefore shown that:

$$\begin{aligned}
&g_1 \mid g_2 \mid g_5 \mid g_6 = \\
&\quad (A \leq B \rightarrow A, B := B - (B - A)/2, B - (B - A)/2) | (A > B \rightarrow ()) = \\
&\quad (A \leq B \rightarrow A, B := B - (B - A)/2, B - (B - A)/2) | () = \\
&\quad f
\end{aligned}$$

5.2. Example: Design

Develop a while loop for the following specification: $f(x, y) = (x > 100 \rightarrow x, y := x, x + 1) \mid (x, y := x, y)$

In order to develop this program from its specifications, use the four-step process based upon rules D.I through D.V. First determine if f is realizable by a **while** loop.

D.I: Is $range(f) \subset domain(f)$**?**

Since f is defined for all input values, $domain(f)$ includes all values of x and y. $Range(f)$ is a subset of (x, y), so condition D.I is true.

D.II: For $(x, y) \in range(f)$**, do we have an identity function, that is,** $f(x, y) = (x, y)$**?**

There are two cases for x: $x > 100$ and $x \leq 100$. For the case of $x > 100$, we have from the specification that $(x, x + 1) \in range(f)$; and $f(x, x+1) = (x, x+1)$, which is an identity. For the case where $x \leq 100$, we know from the specification that $f(x, y) = (x, y)$.

D.III: Find $\boxed{b}$ **that evaluates to true in** $domain(f) - range(f)$ **and false in** $range(f)$.

Find a predicate b that is false on its range and true elsewhere. Since we want y to take on the value $x+1$ or we want x to be less than or equal to 100 on the range of f, we know that $(y = x+1) \vee (x \leq 100)$ will be true on the range and hence false on $domain(f) - range(f)$. So the negative of this has our desired property: $\neg((x \leq 100) \vee (y = x + 1)) = (x >$

$100) \wedge (y \neq x+1)$. Since the loop will exit when this predicate is false, $b = (x > 100) \wedge (y \neq x+1)$, giving the partial solution

$$\textbf{while}(x > 100) \ \wedge \ (y \neq (x+1)) \ \textbf{do} \\ \{d\}$$

D.IV: Develop d so that all values are preserved for f.

Find a function $\boxed{d}$ that preserves the values needed for f. y needs to become $x+1$, so let $d = (x, y := x, x+1)$. Our solution is now:

$$\textbf{while}(x > 100) \ \wedge \ (y \neq x+1) \ \textbf{do} \\ x, y := x, x+1$$

or just

$$\textbf{while}(x > 100) \ \wedge \ (y \neq x+1) \ \textbf{do} \\ y := x+1$$

D.V: Show that the loop must terminate.

We know that b is false on the range of the **while** statement, so if we can prove that the loop terminates, the current values of x and y when the loop terminates must be a feasible solution. The loop can easily be shown to terminate after one iteration.

5.3. Multiplication – Again

As a third example, let us revisit the multiplication program previously proven in Chapter 2, Section 1.3:

```
    {B ≥ 0}
1.  MULT(A, B) ≡
2.      a := A
3.      b := B
4.      y := 0
5.      while b > 0 do {
6.          y := y + a
7.          b := b − 1}
    {y = AB}
```

We first need to give the specification. One possibility is just:

$$B \geq \mathbf{0} \rightarrow y := AB$$

However, more formally, we are interested in how $MULT$ modifies the state space for all variables, so we can write a specification as follows:

$$\exists m_1, m_2, m_3, m_4, (B \geq \mathbf{0} \rightarrow (a, b, y, A, B) := (m_1, m_2, AB, m_3, m_4))$$

This clearly shows that the specification is a relation, and all that we care about is the effect of the program on the variable y.

Because of composition of assignment statement functions, it should be clear that we need to show:

$$\boxed{MULT(A,B)} = \boxed{a := A} \circ \boxed{b := B} \circ \boxed{y := \mathbf{0}} \circ \boxed{while} = AB$$

While verification

The while loop adds ab to y and sets b to 0, therefore, we can use as a candidate function f for this loop the equation:

$$f(a, b, y) = (b > \mathbf{0} \rightarrow (a, \mathbf{0}, y + ab))|()$$

To show this, we must show that the three properties **V.I–V.III** are true.

$f = \boxed{if\ b\ then\ d} \circ f$. In this case, predicate b is $b > \mathbf{0}$ and d is assignment statements 6 and 7 in the program.

This requires four trace tables since we have two choices for the predicate in $\boxed{if}$ and two choices for the predicate in f.

1. f_1 = both are true.

Part	Condition	a	b	y
if	$b > \mathbf{0}$			
$y := y + a$				$y + a$
$b := b - \mathbf{1}$			$b - \mathbf{1}$	
f	$b - \mathbf{1} > \mathbf{0}$	a	$\mathbf{0}$	$y + a + a(b - \mathbf{1}) =$ $y + ab$

$$f_1(a,b,y) = (b > \mathbf{0} \wedge b - \mathbf{1} > \mathbf{0} \rightarrow (a, \mathbf{0}, y + ab)) =$$
$$(b > \mathbf{1} \rightarrow (a, \mathbf{0}, y + ab))$$

2. $f_2 = if$ is false, f is true.

Part	Condition	a	b	y
if	$b \leq \mathbf{0}$			
f	$b > \mathbf{0}$	a	$\mathbf{0}$	$y + ab$

$$f_2(a,b,y) = \quad (b \leq \mathbf{0} \wedge b > \mathbf{0} \rightarrow (a, \mathbf{0}, y + ab)) = null\ function$$

3. $f_3 = if$ is true, f is false.

Part	Condition	a	b	y
if	$b > \mathbf{0}$			
$y := y + a$				$y + a$
$b := b - \mathbf{1}$			$b - \mathbf{1}$	
f	$b - \mathbf{1} \leq \mathbf{0}$			

$$f_3(a,b,y) = \quad (b > \mathbf{0} \wedge b - \mathbf{1} \leq \mathbf{0} \rightarrow (a, b - \mathbf{1}, y + a)) =$$
$$(b > \mathbf{0} \wedge b \leq \mathbf{1} \rightarrow (a, b - \mathbf{1}, y + a)) =$$
$$(b = \mathbf{1} \rightarrow (a, \mathbf{0}, y + ab))$$

4. $f_4 =$ both are false.

Part	Condition	a	b	y
if	$b \leq \mathbf{0}$			
f	$b \leq \mathbf{0}$			

$$f_4(a,b,y) = \quad (b \leq \mathbf{0} \rightarrow ())$$

We then compute the required functionality for the **while** statement:

$$f = \quad f_1|f_3|f_4 =$$
$$(b > \mathbf{1} \rightarrow (a, \mathbf{0}, y + ab))|(b = \mathbf{1} \rightarrow (a, \mathbf{0}, y + ab))|(b \leq \mathbf{0} \rightarrow ()) =$$
$$(b > \mathbf{0} \rightarrow (a, \mathbf{0}, y + ab))|()$$

Now that we have defined f, we have to show that it meets the other two properties:

Property V.II: $domain(f) = domain($ while $)$

Both functions are defined for all b, so domains are equal.

Property V.III: $\neg b \rightarrow f = \neg b \rightarrow ()$.

$$\neg b \rightarrow f =$$
$$b \leq \mathbf{0} \wedge (b > \mathbf{0} \rightarrow (a, \mathbf{0}, y + ab))|()) =$$
$$((b \leq \mathbf{0} \wedge b > \mathbf{0}) \rightarrow (a, \mathbf{0}, y + ab))|(b \leq \mathbf{0} \rightarrow ()) =$$
$$(false \rightarrow (a, \mathbf{0}, y + ab))|(b \leq \mathbf{0} \rightarrow ()) =$$
$$(b \leq \mathbf{0} \rightarrow ())$$

Program verification

Now that we have identified the function of the **while** statement, we can complete the proof of $MULT$ by a trace table of the four program components:

Part	Condition	a	b	y
$a := A$		A		
$b := B$			B	
$y := \mathbf{0}$				$\mathbf{0}$
f		A	$\mathbf{0}$	$\mathbf{0} + AB = AB$

This clearly shows that $MULT(A, B) = AB$. We also need to show termination of the program, but the same argument used previously is still true here.

6. DATA ABSTRACTION DESIGN

The discussion so far has concentrated on the process of developing a correct procedure from a formal specification. However, program design also requires appropriate handling of data.

6.1. Data Abstractions

A *data abstraction* is a class of objects and a set of operators that access and modify objects in that class. Such objects are usually defined via the *type* mechanism of a given programming language, and a module is created consisting of such a type definition and its associated procedures.

Crucial to the data abstraction model are the isolation of the type definition and invocations of the procedures that operate on such objects.

Each procedure has a well-defined input-output definition. The implementor is free to modify any procedure within a module as long as its input-output functional behavior is preserved, and any use of such a procedure can only assume its functional specification. The result of this is that rather than viewing a program as a complex interaction among many objects and procedures, a program can be viewed as the interaction among a small set of data abstractions, each relatively small and well-defined.

Languages such as Ada (or C++) allow data abstractions to be built relatively easily, since the object type can be specified as the **private** part of a **package** (or **class**) specification. Only the body of the **package** has access to the type structure while other modules have access only to the names of the procedures that are contained in the module.

However, even in older languages, such as C or Pascal, data abstractions form a good model of program design. Even though not automatically supported by the language, with reasonable care, programs can be designed which adhere to the data abstraction guidelines.

6.2. Representation Functions

A procedure within a data abstraction translates a high-level description of a process into a lower-level programming language implementation. For example, suppose character strings up to some predefined maximum value are needed. Pascal, for example, only defines fixed-length strings; therefore, we must implement this as objects using primitive Pascal data types.

In procedures outside of the defining module, we would like to refer to these objects (e.g., call them **Vstrings**) and be able to operate on them, while inside the module we need to operate on their Pascal representation (arrays of characters). In the former case, we call such functions **abstract functions** that define the functional behavior of the operation, while we call the latter case **concrete functions** that give the details of the implementation.

For the **Vstring** example, we could define such a string via an abstract comment containing the functional definition:

$$\{abs : x_{abs} \ = < x_1, x_2, \ldots x_n >\}$$

while the concrete representation of a **Vstring** could be:

$$\begin{aligned}&con : x_{con} : record\\ &\quad chars : array(1..maxval)\ of\ char;\\ &\quad size : 0..maxval\\ &\quad end;\end{aligned}$$

To show that both representations are the same, we define a *representation function* which maps concrete objects into abstract objects by mapping a state into a similar state leaving all data unchanged except for those specific objects. Let r map a concrete object into its abstract representation. If **Cstrings** is the set of concrete strings (i.e., the set of variables defined by the above record description) and if **Vstrings** is the set of abstract strings, then we define a representation function r with the signature: $r : state \rightarrow state$ such that

$$r \stackrel{\triangle}{=} \{(u, v) | u = v\ except\ that\ if\ u(x) \in Cstrings, then\ v(x) \in Vstrings\ \}$$

We simply mean that u and v represent the same set of variables in the program store, except that each occurrence of a concrete variable in u is replaced by its abstract definition in v.

For each implementation of a string, we have its abstract meaning given by function r:

$$x_{abs} := < x_{con}.chars[i] | 1 \leq i \leq x_{con}.size >$$

The purpose of a procedure in an abstraction module is to implement an abstract function on this abstract data. For example, if we would like to implement an $Append$ operation, we can define $x := Append(x, y)$ as:

$$\{abs : x_1, \ldots, x_n, \ldots, x_{n+m} := x_1, \ldots x_n, y_1, \ldots, y_m\}$$

Similarly, we can define a concrete implementation of this same function as:

$$\begin{aligned}&\{con : x.chars[n+1], \ldots, x.chars[n+m], x.size :=\\ &\quad y.chars[1], \ldots, y.chars[y.size], x.size + y.size\}\end{aligned}$$

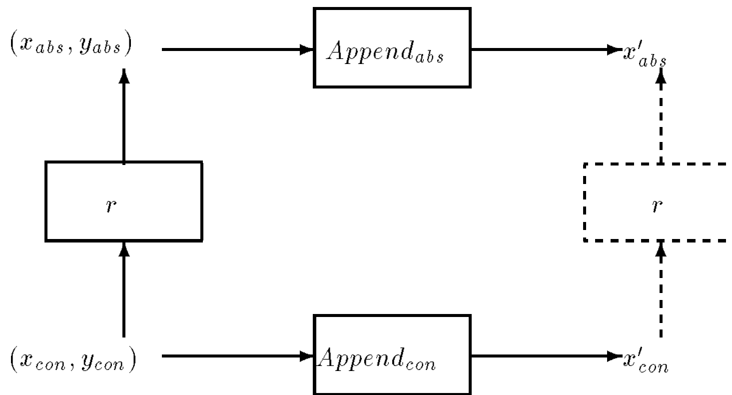


Figure 3.2. Commuting Representation Diagram

If x_{con} and y_{con} represent the concrete implementations of **Vstrings** x and y, and if x_{abs} and y_{abs} represent their abstract representation, and if $Append_{con}$ and $Append_{abs}$ represent the concrete and abstract functions, we have:

$$x'_{con} := Append_{con}(x_{con}, y_{con})$$
$$x'_{abs} := Append_{abs}(x_{abs}, y_{abs})$$

We want to know if both the concrete and abstract functions achieve the same result, that is, the abstract representation of what we get by implementing $Append_{con}$ is the same as our abstract definition of $Append$. This is just the result: Is $r(x'_{con}) = x'_{abs}$? We say that the representation diagram of Figure 3.2 *commutes* (i.e., either path from (x_{con}, y_{con}) to x'_{abs} gives the same result). We have to show that r applied to x'_{con} gives us x'_{abs} (e.g., $x_1, x_2, \ldots, x_n, y_1, \ldots, y_m$).

As given by our correctness theorem, a program (e.g., $Append_{con}$) may often compute a value larger than is necessary ($domain(Append_{abs}(x)$ in this example). Thus, we actually want to show:

$$r \circ Append_{abs} \subset Append_{con} \circ r$$

7. USING FUNCTIONAL VERIFICATION

We have seemingly developed two mechanisms for designing programs: (a) a functional model for showing the equivalence of a design and its implementation, and (b) a commuting diagram for showing correct data abstractions. However, both are complementary ideas of the same theory. For example, the concrete design comment for $Append$ in the previous section is just a concurrent assignment which we can translate into a source program via the techniques described earlier.

This leads to a strategy for developing correct programs:

1. From the requirements of a program, develop the abstract data objects that are needed.

2. For each object, develop abstract functions that may be necessary to operate on the abstract object.

3. Using the abstract object and operations as a goal, design the concrete representation of the object and corresponding representation function.

4. Design a concrete function for each corresponding abstract function.

5. Show that the representation diagram commutes. That is, the concrete function does indeed implement the abstract function.

6. Develop correct programs from each concrete function.

Note the order of Steps 2 and 3. It is important to understand the abstract functions before one designs the concrete representation, since the appropriate representation will depend greatly on the application. Consider the implementation of a **date** data object. The following are all feasible concrete representations depending upon the required abstract functions:

- **Store as character string 'MM/DD/YY.'** This is appropriate if the date is simply a unique tag associated with some data and has no other semantic meaning.

- **Store as** $< YY, DDD >$ where integer YY is the year and integer DDD is the day of year. This is quite efficient if sequential dates are needed.

• **Store as number of days since some initial date.** This is most efficient to compute distances between two days. It also avoids certain problems such as accounting for leap years in all functions, but it is cumbersome to print out in its usual format.

• **Store as** $< MM, DD, YY >$ **for integers MM, DD, and YY.** Computing dates is a bit more cumbersome, but conversion to its usual printed form is quite easy.

The importance of this technique is that it can be applied at any level of detail. Here, we obviously considered only short program segments. For larger programs, only those concepts that are critical to the success of a program need be formalized, although a long range goal would be to develop this or other techniques which can be applied to very large systems in their entirety. Its major difference from other verification techniques is that it forces the programmer or designer to consider the functionality of the program as a whole, and requires the designer to design data structures with operations that operate on those structures. Since this is central to the data abstraction model of program design, this technique is quite applicable to current thinking about programming.

8. EXERCISES

Problem 1. It is not always the case that a **while** loop can be used to implement a desired specification. Therefore, as designers it is important for us to know when a loop can and cannot be used:

Theorem: A **while** statement can be devised to implement a desired specification f if and only if

1. $range(f) \subseteq domain(f)$, and
2. f behaves as the identity function in $range(f)$.

In each of the following parts, a specification f is given, and you are to state whether or not a **while**-loop construct can be devised to implement that specification. If so, then give such a loop. If not, then justify your claim. In all cases, x and y are integer valued variables.

$(a)\quad f = (\mathbf{true} \rightarrow\ x, y := x + \mathbf{1}, y)$
$(b)\quad f = (x \geq \mathbf{0} \rightarrow\ x, y := x + \mathbf{1}, y)$
$(c)\quad f = (\mathbf{true} \rightarrow\ x, y := x + y, \mathbf{0})$
$(d)\quad f = (y \geq \mathbf{0} \rightarrow\ x, y := x + y, \mathbf{0})$
$(e)\quad f = (x \geq \mathbf{0} \wedge y \geq \mathbf{0} \rightarrow ()) \mid (\mathbf{true} \rightarrow\ x, y := y, x)$
$(f)\quad f = (\mathbf{true} \rightarrow\ x, y := max(x, y), min(x, y))$

Problem 2. Why does the Mills' style method of functional correctness not have a rule corresponding to Hoare's rule of adaptation?

Problem 3. Consider the specification

$$f = (n \geq \mathbf{0} \wedge n \geq k \rightarrow\ x, n, k := \ \frac{n!}{k! \cdot (n-k)!}\ , n, k)$$

which ignores any effect a program might have on any other program variables. Write a program to implement this specification, and demonstrate that your program satisfies f by using the Mills approach.

Problem 4. Now consider the specification

$$f = (n \geq \mathbf{0} \wedge n \geq k \rightarrow\ x, n, k := \ \frac{n!}{n! \cdot (n-k)!}\ , n, k)$$

and write a program to implement this specification, and demonstrate that your program satisfies f by using the Mills approach.

Problem 5. Prove that the following program P satisfies the specification

$$F = (n \geq \mathbf{1} \rightarrow\ t, i := n!, n + \mathbf{1})$$

Use Mills' techniques.

$P \equiv \{$
$\quad t := \mathbf{1}$
$\quad i := \mathbf{1}$
$\quad \mathbf{while}\ i \leq n\ \mathbf{do}$
$\quad\quad t := t * i$
$\quad\quad i := i + \mathbf{1}\}$

Problem 6. Prove that the following program P satisfies the specification

$$\begin{aligned} F \;=\; & (i > 0 \;\wedge\; j > 0 \;\rightarrow i, j := \mathbf{gcd}(i, j), 0) \\ & \mid\; (i \leq 0 \;\wedge\; j > 0 \;\rightarrow i, j := j, \mathbf{remainder}(i, j)) \\ & \mid\; (j \leq 0 \rightarrow \;) \end{aligned}$$

Use Mills' techniques.

$$\begin{aligned} & P \equiv \{ \\ & \quad \textbf{while } j > 0 \textbf{ do} \\ & \qquad r := \mathbf{remainder}(i, j) \\ & \qquad i := j \\ & \qquad j := r\} \end{aligned}$$

Problem 7. Define the semantics of the **repeat** ... **until** statement, and verify that the program fragment

$$\{\textbf{repeat } y := y - 1 \;;\; x := x + 2 \textbf{ until } y \leq 0\}$$

computes the function

$$(y > 0 \;\rightarrow\; x, y := x + 2 \cdot y, 0) \mid (y \leq 0 \;\rightarrow\; x, y := x + 2, y - 1)$$

Problem 8. Find the function computed by the following procedure, and verify your answer using Mills' techniques:

$$\begin{aligned} & z(a, b) : ()\textbf{procedure} \equiv \{ \\ & \quad \textbf{if } a > 0 \textbf{ then} \\ & \qquad a := a - 1 \\ & \qquad \textbf{call } z(a, b) \\ & \qquad b := b + 1\} \end{aligned}$$

Problem 9. Prove that the following program P satisfies the specification

$$F \;=\; (n \geq 0 \;\rightarrow\; y := x^n)$$

Use Mills' techniques.

```
P ≡ {
    k := n
    y := 1
    z := x
    while k > 0 do
        if odd(k)
            then {k := k − 1; y := y ∗ z}
            else {k := k/2; z := z ∗ z}}
```

Problem 10. Write a program that takes an integer $z \geq 2$ and determines whether or not (say, by setting a Boolean flag appropriately on output) z is a prime number. One method that could be used to check primality is to see whether any integers from 2 up to z will evenly divide z (you are free to assume the existence of a primitive function mod as in Pascal). Prove the correctness of your program by Mills' techniques.

Problem 11. Most formal systems for reasoning about programs restrict procedure bodies to contain *no* nonlocal variable references. Is this restriction truly necessary when using Mills' techniques? Clearly state your answer. If the restriction is not necessary, provide a convincing argument to support your claim. If the restriction is necessary, provide a simple example to illustrate why it is so.

Problem 12. Consider the program:

```
k := 0
while n > 1 do
    if prime(n) then k := k + 1
    n := n − 1
```

which assumes the existence of a Boolean-valued function called ***prime*** having the obvious behavior. Using Mills' techniques, prove that this program meets the formal specification

$$(\; 2 \leq n \leq 100 \; \rightarrow \; k := \varphi(n) \;)$$

where $\varphi(i)$ is defined to be "the number of primes in the range $[2, i]$."

Problem 13. Determine the function computed by the following code:

```
while x ≠ 0 do
    t := 0
    while t ≠ y do
        s := s + 1
        x := x + 3
        t := t + 1
        y := y + x
    x := x − 1
```

Problem 14. Write a program whose function *exactly equals* the following specification:

$$(\ 2 \le n < 5 \ \rightarrow\ x := prime(n)\ )$$

where in this case, our function $prime(i)$ returns the ith prime number (for example, $prime(1) = 2$ and $prime(7) = 17$). Verify that your program behaves exactly like the specification using Mills' techniques.

Problem 15. Determine the function computed by the following procedure, and prove it is so by Mills' techniques:

```
mystery procedure (x, n, z) : () ≡ {
    if n = 0 then
        z := 1
    else
        n := n − 1
        call mystery(x, n, z) : ()
        x := 2x
        z := 2}
```

Problem 16. Consider the functional specification

$$f\ =\ (0 \le n\ \rightarrow\ k := bits(n))$$

where ***bits***(j) is defined as "the number of 1 bits used to represent the integer j as a binary numeral." (For instance, ***bits***(1)=1, ***bits***(8)=1, and ***bits***(7)=3.)

Write a program that implements this specification. Verify the implementation using Mills' techniques.

9. SUGGESTED READINGS

The first sections of the paper

• D.D. Dunlop and V.R. Basili, "A Comparative Analysis of Functional Correctness," *ACM Computing Surveys*, Vol. 14, No. 2, June 1982, pp. 229-244.

provide an especially useful introduction to this area. The original works, of course, are also invaluable:

• R.C. Linger, H.D. Mills and B.I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979, Chapter 6.

• H.D. Mills, "The New Math of Computer Programming, *Communications of the ACM*, Vol. 18, No. 1, 1975, pp. 43-48.

The papers

• J.D. Gannon, R.B. Hamlet and H.D. Mills, "Theory of Modules," *IEEE Transactions on Software Engineering*, Vol. 13, No. 7, 1987, pp. 820-829.

• M. V. Zelkowitz, "A functional correctness model of program verification," *IEEE Computer*, Vol. 23, No. 11, 1990, pp. 30-39.

are works that clarify many of the more subtle points associated with functional correctness. Finally, even though the material can not be described as being compact, chapters 6–11 in

• H.D. Mills, V.R. Basili, J.D. Gannon and R.G. Hamlet, *Principles of Computer Programming: A Mathematical Approach*, William C. Brown, Dubuque, IA, 1987.

provide a large number of exercises for students of the functional approach.

Chapter 4

Predicate Transformers

In work related to the Hoare-style axiomatic proof, Dijkstra developed the concept of the *weakest precondition* [10]. Using the concept of a *state*, much like in the functional model of Chapter 3, the idea of the weakest precondition is defined. In addition, Dijkstra considered the concept of nondeterminacy in programming languages with the introduction of the guarded command. These simplify some of the verification properties we wish to prove.

1. GUARDED COMMANDS

Most programming languages (and algorithms) are sequential and deterministic. For example, the symbolic trace table method of Chapter 3 is strictly based upon being able to trace the execution of a program statement by statement in a prescribed order. However, there are times when nondeterminacy makes a software design easier. Unfortunately, we have not had the programming tools for constructing "nondeterministic" programs in existing languages. We first describe several constructs for introducing nondeterminacy into languages and then show how some algorithms become easier to describe using this method.

The basic concept is the *guard* which is written as: $[\!]$. If B is a predicate and S is a statement, then in a guarded command, $B \rightarrow S$ means that

statement S is "enabled" or ready for execution if guard B is *true*. We use guards in **guarded if** and **repetitive** statements:

1.1. Guarded If Statement

If B_i is a set of predicates and S_i is a set of statements, the **guarded if** is written as:

$$\textbf{if } B_1 \rightarrow S_1 \, [\!] \, B_2 \rightarrow S_2 \, [\!] \, \ldots \, [\!] \, B_n \rightarrow S_n \textbf{ fi}$$

This has the meaning that at least one of the predicates is *true*, and the execution is that corresponding statement, or:

$$\exists i \ni B_i \text{ and execution is } S_i$$

Note that this is not the same as the LISP $cond$ or the conditional design statement from Chapter 3. In those cases, the statement to execute was the one with the minimal i such that $B_i = true$. Here we arbitrarily choose any i with B_i true.

Two implications of this definition:

1. At least one of the predicates must be true; otherwise the statement is equivalent to an **abort** statement.

2. If more than one predicate is true, no indication of which statement to execute is made. Any statement having a true guard is executed. This introduces nondeterminacy into algorithms.

1.2. Repetitive statement

This is the generalization to the sequential **while** statement and is similar to the **guarded if**: If B_i is a set of predicates and S_i is a set of statements, the **do** is written as:

$$\textbf{do } B_1 \rightarrow S_1 \, [\!] \, B_2 \rightarrow S_2 \, [\!] \, \ldots \, [\!] \, B_n \rightarrow S_n \textbf{ od}$$

Execution proceeds as follows: If any guard B_i is *true*, the corresponding S_i is executed. This process repeats as long as some guard is true.

There are several implications to this definition:

1. If no guard is true initially, the **do** is equivalent to the **skip** or **null** statement.

2. As in the guarded **if**, nondeterminacy is introduced into the execution if more than one guard is true at the same time.

3. When execution exits the **do**, we know that all guards must be equal to *false*.

Using guarded commands often makes many algorithms easier to develop or understand. Consider the following two examples:

Example 1. Trivial example for $maximum$ of two numbers:

$$\textbf{if } x \geq y \ \rightarrow maximum := x \ [\!] \ y \geq x \ \rightarrow maximum := y \ \textbf{fi}$$

This simply says if x is greater, then use x; if y is greater, use y. We have an ambiguous situation when $x = y$, but either choice gives the same answer.

Example 2. Consider the binary search algorithm given as Problem 6 at the end of Chapter 2:

$$\begin{array}{l}
\textbf{PROGRAM} \equiv \{ \\
\quad i := \mathbf{0}; \\
\quad j := n; \\
\quad \textbf{while } i + \mathbf{1} \neq j \textbf{ do} \\
\quad\quad h := (i + j)/\mathbf{2}; \\
\quad\quad \textbf{if } a[h] \leq x \\
\quad\quad\quad \textbf{then } i := h \\
\quad\quad\quad \underline{e}lse \ j := h\}
\end{array}$$

We can write this using guarded commands as:

$$\begin{array}{l}
\textbf{PROGRAM} \equiv \{ \\
\quad i := \mathbf{0}; \\
\quad j := n; \\
\quad \textbf{do } i + \mathbf{1} \neq j \rightarrow \\
\quad\quad h := (i + j)/\mathbf{2}; \\
\quad\quad \textbf{if } a[h] \leq x \rightarrow i := h \\
\quad\quad\quad [\!] \ a[h] \geq x \rightarrow j := h \textbf{ fi} \\
\quad \textbf{od}\}
\end{array}$$

2. WEAKEST PRECONDITIONS

Let S be a statement in some programming language and let Q be a predicate. $wp(S, Q)$ is the set of initial states (described by a predicate) for which S terminates and Q is true on termination. Note that this differs from the earlier axiomatic model since termination is inherent in the definition of pre- and postconditions.

A program S is *correct* with respect to predicates $[P, Q]$ if $P \Rightarrow wp(S, Q)$.

2.1. Axioms

Several axioms must be satisfied by the weakest precondition wp:

1. $wp(S, false) = false$
2. $$\frac{P \Rightarrow Q}{wp(S, P) \Rightarrow wp(S, Q)}$$
3. $$\frac{wp(S, P \vee Q)}{wp(S, P) \vee wp(S, Q)}$$
4. $$\frac{wp(S, P \wedge Q)}{wp(S, P) \wedge wp(S, Q)}$$

Since we are building an axiomatic model, many of the properties from Chapter 2 still apply:

Assignment. If $x := e$ is an assignment statement and Q is a postcondition:

$$wp(x := e, Q) = Q_e^x$$

Composition. If S_1 and S_2 are statements and Q is a predicate:

$$wp(S_1; S_2, Q) \triangleq wp(S_1, wp(S_2, Q))$$

2.2. If Statements

Given **if** statement:

$$if \equiv \textbf{if}\ B_1 \rightarrow S_1 \ [\!]\ B_2 \rightarrow S_2 \ [\!]\ \ldots \ [\!]\ B_n \rightarrow S_n \ \textbf{fi}$$

we define its weakest precondition as follows:

$$
\begin{aligned}
wp(if, Q) \triangleq\ & (\exists j, 1 \leq j \leq n \ni B_j) \wedge \\
& (\forall i, 1 \leq i \leq n \ni B_i \Rightarrow wp(S_i, Q)) \\
=\ & (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge \\
& (B_1 \Rightarrow wp(S_1, Q)) \wedge \\
& (B_2 \Rightarrow wp(S_2, Q)) \wedge \\
& (B_3 \Rightarrow wp(S_3, Q)) \wedge \\
& \vdots \\
& (B_n \Rightarrow wp(S_n, Q))
\end{aligned}
$$

Using the definition of the **if** statement, we get the following theorem:

Theorem: Let $BB \triangleq (\exists j, 1 \leq j \leq n \ni B_j)$, predicates Q and R, and **if** statement

$$if \equiv \textbf{if } B_1 \rightarrow S_1 \ [\!]\ B_2 \rightarrow S_2 \ [\!]\ \ldots \ [\!]\ B_n \rightarrow S_n \textbf{ fi}$$

If $Q \Rightarrow BB$ and $(\forall i, 1 \leq j \leq n \ni (Q \wedge B_j) \Rightarrow wp(S_j, R))$
then $Q \Rightarrow wp(if, R)$

Proof:

$$
\begin{array}{ll}
1.\ Q \Rightarrow BB & Assumption \\
2.\ Show:\ Q \Rightarrow (\forall j, 1 \leq j \leq n \ni B_j \Rightarrow wp(S_j, R)) & \\
\quad (\forall j, 1 \leq j \leq n \ni (Q \wedge B_j) \Rightarrow wp(S_j, R)) & Assumption \\
\quad Q \wedge B \Rightarrow W \equiv \neg(Q \wedge B) \vee W \equiv & \\
\quad \neg Q \vee \neg B \vee W \equiv \neg Q \vee (B \Rightarrow W) \equiv & \\
\quad Q \Rightarrow (B \Rightarrow W) & \\
3.\ Q \Rightarrow BB \wedge (\forall j, 1 \leq j \leq n \ni B_j \Rightarrow wp(S_j, R)) & 1\ and\ 2 \\
4.\ Q \Rightarrow wp(if, R) & Def.\ of\ wp(if, R)
\end{array}
$$

□

Using these results, we can derive weakest preconditions for the traditional **if then** and **if then else** statements used elsewhere in this book:

If then else statement.

$$
\begin{aligned}
&wp(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, R) \stackrel{\triangle}{=} \\
&\quad (B \Rightarrow wp(S_1, R)) \wedge (\neg B \Rightarrow wp(S_2, R)) = \quad \textit{See Problem } 6 \\
&\quad (B \wedge wp(S_1, R)) \vee (\neg B \wedge wp(S_2, R))
\end{aligned}
$$

If then statement.

$$
\begin{aligned}
&wp(\textbf{if } B \textbf{ then } S_1, R) \stackrel{\triangle}{=} \\
&\quad (B \Rightarrow wp(S_1, R)) \wedge (\neg B \Rightarrow R) = \\
&\quad (B \wedge wp(S_1, R)) \vee (\neg B \wedge R)
\end{aligned}
$$

For these statements, $BB = B \vee \neg B = true$. Thus $Q \Rightarrow BB$ must be true, so we do not have to establish that to conclude $Q \Rightarrow wp(if, R)$.

Example 3: Verify the following program with respect to its specification:

$$
\begin{aligned}
&\{a \geq 0 \wedge b \geq 0 \wedge c \geq 0\} \\
&\quad \textbf{if } a < b \textbf{ then } a := b; \\
&\quad \textbf{if } a < c \textbf{ then } a := c \\
&\{a \geq b \wedge a \geq c\}
\end{aligned}
$$

We develop the proof as follows:

$$
\begin{aligned}
&wp(a := c; a \geq b \wedge a \geq c) = (c \geq b \wedge c \geq c) = c \geq b \\
&wp(\textbf{if } a < c \textbf{ then } a := c, a \geq b \wedge a \geq c) = \\
&\quad (a < c \wedge c \geq b) \vee (a \geq c \wedge a \geq b \wedge a \geq c) = \\
&\quad (a < c \wedge c \geq b) \vee (a \geq b \wedge a \geq c)) \\
&wp(a := b; (a < c \wedge c \geq b) \vee (a \geq b \wedge a \geq c)) = \\
&\quad (b < c \wedge c \geq b) \vee (b \geq b \wedge b \geq c) = \\
&\quad (c > b \vee b \geq c) = \\
&\quad true \\
&wp(\textbf{if } a < b \textbf{ then } a := b, (a < c \wedge c \geq b) \vee (a \geq b \wedge a \geq c)) = \\
&\quad (a < b) \vee (a \geq b \wedge ((a < c \wedge c \geq b) \vee (a \geq b \wedge a \geq c))) = \\
&\quad (a < b) \vee (a \geq b \wedge a < c \wedge c \geq b) \vee (a \geq b \wedge a \geq c) = \\
&\quad (a < b) \vee (a \geq b \wedge a < c) \vee (a \geq b \wedge a \geq c) = \\
&\quad (a < b) \vee (a \geq b \wedge (a < c \vee a \geq c)) = \\
&\quad (a < b) \vee (a \geq b) = true \\
&wp(\textbf{if } a < b \textbf{ then } a := b; \textbf{if } a < c \textbf{ then } a := c, a \geq b \wedge a \geq c) = \\
&\quad true
\end{aligned}
$$

Since $a \geq 0 \wedge b \geq 0 \wedge c \geq 0 \Rightarrow wp(\mathbf{if} \cdots, a \geq b \wedge a \geq c)$, the program is correct.

2.3. Do Statements

Consider the **repetitive** or **do** statement:

$$do \equiv \mathbf{do}\ B_1 \rightarrow S_1 \ [\!]\ B_2 \rightarrow S_2 \ [\!]\ \ldots \ [\!]\ B_n \rightarrow S_n\ \mathbf{od}$$

Let BB, Q, and R be defined as before. Define if as the **do** statement with **do od** replaced by **if fi** (i.e., it represents one "pass" through the loop).

We define $wp(do, R) \stackrel{\triangle}{=} (\exists k, k \geq 0 \ni H_k(R))$ where we need auxiliary functions H_i defined as follows:

$$H_0(R) \stackrel{\triangle}{=} \quad R \wedge \neg(\exists j, 1 \leq j \leq n \ni B_j) = R \wedge \neg BB$$
$$H_k(R) \stackrel{\triangle}{=} \quad wp(if, H_{k-1}(R)) \vee H_0(R)\ for\ k > 0$$

The function H_k represents the number of passes through the loop. H_0 represents a loop that has no enabled guards, hence is equivalent to the null statement. H_k represents k passes through the loop, where each pass is equivalent to the if statement defined above. Writing a general expression for H_k in this model is equivalent to the loop invariant of the Hoare axiomatic model.

In order to use this definition, since the weakest precondition depends upon termination, we need to know that the **do** terminates. In the axiomatic model we used property P such that $P(i)$ was always positive, yet decreasing in the loop. This was handled outside of the formal Hoare axioms.

In this case, the termination property equivalent to the $P(i)$ of the axiomatic model is the value of k in function H_k. We give the following theorem which permits us to use this definition.

Iteration Theorem: Given a repetitive **do**, predicate P, integer function t, and new identifier t_1:

If 1. $\forall i, 1 \leq i \leq n, P \wedge B_i \Rightarrow wp(S_i, P)$
2. $P \wedge BB \Rightarrow t > 0$
3. $\forall i, 1 \leq i \leq n, P \wedge B_i \Rightarrow wp(t_1 := t; S_i, t < t_1)$

then $P \Rightarrow wp(do, P \wedge \neg BB)$

Proof:

1. Show $(\forall i, P \wedge B_i \Rightarrow wp(S_i, P)) \Rightarrow (P \wedge BB \Rightarrow wp(if, P))$.

$$\begin{array}{ll}
(\forall i, P \wedge B_i \Rightarrow wp(S_i, P)) = P \wedge (\forall i, B_i \Rightarrow wp(S_i, P)) & \\
(P \wedge B_1) \vee (P \wedge B_2) \vee \cdots = P \wedge BB & Assumption\ 1 \\
P \wedge BB \wedge P \wedge (\forall i, B_i \Rightarrow wp(S_i, P)) & \\
\Rightarrow BB \wedge (\forall i, B_i \Rightarrow wp(S_i, P)) = wp(if, P) &
\end{array}$$

2. Show $(\forall i, P \wedge B_i \Rightarrow wp(t_1 := t; S_i, t < t_1)) \Rightarrow (\forall t_0, P \wedge BB \wedge t \leq t_0 + 1 \Rightarrow wp(if, t \leq t_0))$.

$$\begin{array}{ll}
P \wedge B_i = (P \wedge B_1) \vee (P \wedge B_2) \vee \cdots \vee (P \wedge B_n) = & \\
\quad P \wedge (B_1 \vee B_2 \vee \cdots \vee B_n) = P \wedge BB & \\
wp(t_1 := t; S_i, t < t_1) = wp(S_i, t < t_1)_t^{t_1} & Def\ of\ assignment \\
(\forall i, P \wedge B_i \Rightarrow wp(t_1 := t; S_i, t < t_1)) = & \\
\quad P \wedge (\forall i, B_i \Rightarrow wp(S_i, t < t_1)_t^{t_1}) & \\
P \wedge BB \wedge P \wedge (\forall i, B_i \Rightarrow wp(S_i, t < t_1)_t^{t_1}) & \\
\quad \Rightarrow BB \wedge (\forall i, B_i \Rightarrow wp(S_i, t < t_1)_t^{t_1}) & \\
\quad \Rightarrow wp(if, t < t_1)_t^{t_1} & \\
Thus\ P \wedge BB \Rightarrow wp(if, t < t_1)_t^{t_1} &
\end{array}$$

The following holds for all values of t_0:

$$\begin{array}{ll}
P \wedge BB \wedge t \leq t_0 + 1 \Rightarrow wp(if, t < t_1)_t^{t_1} \wedge t \leq t_0 + 1 & \\
Add\ t \leq t_0 + 1\ to\ both\ sides: & \\
\quad P \wedge BB \wedge t \leq t_0 + 1 & \\
\qquad \Rightarrow (wp(if, t < t_1)_t^{t_1} \wedge t_1 \leq t_0 + 1)_t^{t_1} & Text\ subst. \\
\quad P \wedge BB \wedge t \leq t_0 + 1 & \\
\qquad \Rightarrow (wp(if, t < t_1)_t^{t_1} \wedge wp(if, t_1 \leq t_0 + 1))_t^{t_1} &
\end{array}$$

if does not refer to t_1 or t_0. Thus $wp(if, t_1 \leq t_0 + 1) = t_1 \leq t_0 + 1 \wedge BB$. Since BB appears on the left of the implication, we have not added anything that cannot be proven on the right side.

$$\begin{array}{l}
P \wedge BB \wedge t \leq t_0 + 1 \Rightarrow wp(if, t \leq t_1 \wedge t_1 \leq t_0 + 1)_t^{t_1} \\
P \wedge BB \wedge t \leq t_0 + 1 \Rightarrow wp(if, t \leq t_0)_t^{t_0} \\
\quad t < t_1 \wedge t_1 \leq t_0 + 1 \Rightarrow t \leq t_0 \\
P \wedge BB \wedge t \leq t_0 + 1 \Rightarrow wp(t_1 := t; if, t \leq t_0) \\
P \wedge BB \wedge t \leq t_0 + 1 \Rightarrow wp(if, t \leq t_0) wp(t_1 := t; if, t \leq t_0) = \\
wp(if, t \leq t_0)
\end{array}$$

3. Show $P \wedge t \leq k \Rightarrow H_k(P \wedge \neg BB)$ for $k \geq 0$ by induction.

Base case: $k = 0$.

$$
\begin{array}{ll}
P \wedge BB \Rightarrow t > 0 & Assumption\ 2 \\
\neg P \vee \neg BB \vee t > 0 \equiv \neg P \vee \neg(t \leq 0) \vee \neg BB \equiv & \\
\quad \neg(P \wedge t \leq 0) \vee \neg BB \equiv P \wedge t \leq 0 \Rightarrow \neg BB & \\
P \wedge t \leq 0 \Rightarrow P \wedge \neg BB = P \wedge t \leq 0 & \\
\quad \Rightarrow H_0(P \wedge \neg BB) &
\end{array}
$$

Recursive case: Assume $P \wedge t \leq k \Rightarrow H_k(P \wedge \neg BB)$*. Show:* $P \wedge t \leq k + 1 \Rightarrow H_{k+1}(P \wedge \neg BB)$.

$$
\begin{array}{ll}
P \wedge t \leq k + 1 \equiv & \\
\quad (P \wedge BB \wedge t \leq k + 1) \vee (P \wedge \neg BB \wedge t \leq k + 1) & \\
P \wedge BB \wedge t \leq k + 1 \Rightarrow wp(if, P \wedge t \leq k) & \\
\quad P \wedge BB \Rightarrow wp(if, P) & Step\ (b) \\
\quad P \wedge BB \wedge t \leq k + 1 \Rightarrow wp(if, t \leq k + 1) & \\
\quad wp(if, P) \wedge wp(if, t \leq k + 1) = & \\
\quad\quad wp(if, P \wedge t \leq k + 1) & \\
P \wedge BB \wedge t \leq k + 1 \Rightarrow wp(if, H_k(P \wedge \neg BB)) & \\
\quad P \wedge t \leq k + 1 \Rightarrow H_k(P \wedge \neg BB) & Inductive\ Hyp. \\
P \wedge \neg BB \wedge t \leq k + 1 \Rightarrow H_0(P \wedge \neg BB) & \\
\quad H_0(P \wedge \neg BB) = P \wedge \neg BB & \\
P \wedge t \leq k + 1 \Rightarrow wp(if, H_k(P \wedge \neg BB)) & \\
\quad \vee H_0(P \wedge \neg BB) = H_{k+1}(P \wedge \neg BB) &
\end{array}
$$

Thus $P \wedge t \leq k + 1 \Rightarrow H_k(P \wedge \neg BB)$.

The proof assumed the existence of an upper bound k for t in any state: $(\exists k, 0 \leq k, t \leq k)$ is true in any state.

$$
\begin{array}{l}
P = P \wedge (\exists k, 0 \leq k, t \leq k) = (\exists k, 0 \leq k, P \wedge t \leq k) \Rightarrow \\
(\exists k, 0 \leq k, H_k(P \wedge \neg BB)) = wp(do, P \wedge \neg BB)
\end{array}
$$

□

3. USE OF WEAKEST PRECONDITIONS

Using this theory, let us revisit some of the programs we already discussed in Chapter 2 in order to see how the proofs change.

3.1. Example: Integer Division

We previously discussed this program in Chaper 2, Section 1.1:

$$
\begin{array}{l}
\{x \geq 0 \wedge y > 0\} \\
\textbf{PROGRAM} \equiv \{ \\
\quad q := 0; \\
\quad r := x; \\
\quad \textbf{while } y \leq r \textbf{ do} \\
\quad\quad r := r - y; \\
\quad\quad q := 1 + q\} \\
\{(0 \leq r < y) \ \wedge \ x = r + yq\}
\end{array}
$$

The postcondition to this program is given as $P \stackrel{\triangle}{=} x = qy + r \wedge 0 \leq r < y$ and we have already shown in Chapter 2 that the invariant to the **while** loop in this program is $x = qy + r \ \wedge \ 0 \leq r$. We will assume the same invariant and show that it works in this case also.

However, we also need the integer function t for computing H_k by the Iteration Theorem. We will use $t \equiv r$ as our candidate function.

Consider the three steps on proving the weakest precondition for a **do** statement by the Iteration Theorem:

$$
\begin{array}{l}
1. \ \mathit{Show} \ P \wedge BB \Rightarrow wp(S_i, P) \\
\quad wp(r := r - y; q := 1 + q, P) = \\
\quad\quad 0 \leq r - y \wedge 0 < y \wedge x = (1 + q)y + r - y = \\
\quad\quad 0 \leq r - y \wedge 0 < y \wedge x = qy + r \\
\quad P \wedge r \geq y \Rightarrow 0 \leq r - y \wedge 0 < y \wedge x = qy + r
\end{array}
$$

$$
\begin{array}{l}
2. \ \mathit{Show} \ P \wedge BB \Rightarrow (t > 0) \\
\quad r \geq y \wedge y > 0 \Rightarrow r > 0
\end{array}
$$

$$
\begin{array}{l}
3. \ \mathit{Show} \ P \wedge BB \Rightarrow wp(t_1 := t; S_i, t < t_1) \\
\quad\quad wp(t_1 := r; r := r - y; q := 1 + q, r < t_1) = \\
\quad\quad wp(t_1 := r, r - y < t_1) = r - y < r \\
\quad wp(t_1 := r; r - y < t_1) = wp(t_1 := t; S_i, t < t_1)
\end{array}
$$

So $P \Rightarrow wp(do, P \wedge \neg r \geq y)$.

$$P \wedge \neg r \geq y \Rightarrow 0 \leq r < y \wedge x = qy + r \Rightarrow$$
$$wp(do, P \wedge \neg r \geq y) \Rightarrow wp(do, 0 \leq r < y \wedge x = qy + r) \quad Axiom\ 2$$

So we can conclude that our invariant is the weakest precondition to the loop:

$$P \Rightarrow wp(do, 0 \leq r < y \wedge x = qy + r)$$

We complete the proof, by computing the weakest precondition of the two initialization statements:

$$wp(q := 0; r := x, P) =$$
$$wp(q := 0; 0 \leq x \wedge 0 < y \wedge x = qy + r) =$$
$$0 \leq x \wedge 0 < y \wedge x = 0y + x = 0 \leq x \wedge 0 < y$$

This results in: $0 \leq x \wedge 0 < y \Rightarrow wp(PROGRAM, x = qy + r \wedge 0 \leq r < y)$.

3.2. Still More Multiplication

Let us repeat the multiplication program $MULT$ that we proved earlier in Chapter 2, Chapter 1.3:

```
    {B ≥ 0}
1.  MULT(A, B) ≡ {
2.      a := A
3.      b := B
4.      y := 0
5.      while b > 0 do
6.          y := y + a
7.          b := b − 1}
    {y = AB}
```

We will redo the proof, and will show that except for notational differences, it is quite similar to the Hoare-axiomatic proof done earlier.

We already know that the invariant I for the loop is $y + ab = AB \wedge b \geq 0$. By Axiom 2 after we establish $I \wedge \neg b > 0 \Rightarrow y = AB$, we can conclude

$wp(S, I \wedge \neg b > 0) \Rightarrow wp(S, y = AB)$. Next we use the Iteration Theorem to demonstrate $I \Rightarrow wp(\mathbf{while} b > 0 \mathbf{do} \cdots, I \wedge \neg b > 0)$.

The function t is just the value of b. If we let S refer to the body of the **while** loop, by using the iteration theorem, we need to show:

$$\begin{aligned}
&1.\ I \wedge BB \Rightarrow wp(S, I)\\
&2.\ I \wedge BB \Rightarrow b > 0\\
&3.\ I \wedge BB \Rightarrow wp(t_1 := b; S, b < t_1)
\end{aligned}$$

1. Show $I \wedge BB \Rightarrow wp(S, I)$.

$$\begin{aligned}
&wp(y := y + a; b := b - 1, I) \equiv\\
&wp(y := y + a; b := b - 1, y + ab = AB \wedge b \geq 0) \equiv\\
&y + a + a(b - 1) = AB \wedge (b - 1) \geq 0 \equiv\\
&y + ab = AB \wedge b > 0
\end{aligned}$$

$$\begin{aligned}
&I \wedge BB \equiv\\
&(y + ab = AB \wedge b \geq 0) \wedge b > 0 \equiv\\
&y + ab = AB \wedge b > 0
\end{aligned}$$

2. Show $I \wedge BB \Rightarrow b > 0$.

$$\begin{aligned}
I \wedge BB &\equiv (y + ab = AB \wedge b \geq 0) \wedge b > 0\\
&\equiv y + ab = AB \wedge b > 0
\end{aligned}$$

3. Show $I \wedge BB \Rightarrow wp(t_1 := b; S, b < t_1)$.

$$\begin{aligned}
&wp(t_1 := b; S, b < t_1) \equiv\\
&wp(t_1 := b; y := y + a; b := b - 1, b < t_1) \equiv\\
&wp(t_1 := b; y := y + a, b - 1 < t_1) \equiv\\
&wp(t_1 := b; b - 1 < t_1) \equiv\\
&b - 1 < b \equiv true
\end{aligned}$$

Computing the weakest precondition of the three initialization statements with respect to I completes the proof:

$$\begin{aligned}
&wp(a := A; b := B; y := 0, I) \equiv\\
&wp(a := A; b := B; y := 0, y + ab = AB \wedge b \geq 0) \equiv\\
&wp(a := A; b := B, 0 + ab = AB \wedge b \geq 0) \equiv\\
&wp(a := A, aB = AB \wedge B \geq 0) \equiv\\
&AB = AB \wedge B \geq 0 \equiv\\
&B \geq 0
\end{aligned}$$

Thus $B \geq 0 \Rightarrow wp(MULT(A,B), y = AB)$

However, Problem 2 of Chapter 2 demonstrates that $A = 0$ is also a valid precondition to this program. Since the weakest precondition is the *largest* set that fulfills the postcondition, why didn't we find it in this case?

The reason is that we have simply shown that $B \geq 0 \Rightarrow wp(\cdots)$ and not that $B \geq 0 = wp(\cdots)$. In order to show equality, let's repeat the proof using the actual definition of $wp(do, R)$ using functions H_k.

Define the following:

$$\begin{aligned}
R &\triangleq \quad y = AB \\
S &\triangleq \quad y := y + a; b := b - 1 \\
BB &\triangleq \quad b > 0
\end{aligned}$$

$$\begin{aligned}
H_0(R) = \quad & R \wedge \neg BB \equiv (y = AB \wedge b \leq 0) \\
H_1(R) = \quad & wp(if, H_0(R)) \vee H_0(R) \equiv \\
& wp(\textbf{if } b > 0 \textbf{then begin } y := y + a; b := b - 1 \textbf{ end}, \\
& \quad y = AB \wedge b \leq 0) \vee (y = AB \wedge b \leq 0) \equiv \\
& ((b > 0 \wedge wp(y := y + a; b := b - 1, y = AB \wedge b \leq 0)) \\
& \quad \vee (b \leq 0 \wedge (y = AB \wedge b \leq 0)) \vee (y = AB \wedge b \leq 0) \equiv \\
& ((b > 0 \wedge y + a = AB \wedge b - 1 \leq 0) \vee (b \leq 0 \wedge (y = AB) \equiv \\
& (y + a = AB \wedge b = 1) \vee (b \leq 0 \wedge y = AB) \\
H_2(R) = \quad & wp(if, H_1(R)) \vee H_0(R) \equiv \\
& wp(\textbf{if } b > 0 \textbf{then begin } y := y + a; b := b - 1 \textbf{ end}, \\
& \quad (y + a = AB \wedge b = 1) \vee (b \leq 0 \wedge y = AB)) \\
& \quad \vee (y = AB \wedge b \leq 0) \equiv \\
& ((b > 0 \wedge wp(y := y + a; b := b - 1, (y + a = AB \wedge b = 1) \vee \\
& \qquad (b \leq 0 \wedge y = AB)) \\
& \quad \vee (b \leq 0 \wedge ((y + a = AB \wedge b = 1) \vee (b \leq 0 \wedge y = AB))) \\
& \quad \vee (y + ab = AB \wedge b \leq 0) \equiv \\
& ((b > 0 \wedge y + a + a = AB \wedge b - 1 = 1) \vee (b > 0 \wedge b - 1 \leq 0 \\
& \qquad \wedge y + a = AB)) \\
& \quad \vee (false) \vee (b \leq 0 \wedge y = AB))) \\
& \quad \vee (y + ab = AB \wedge b \leq 0) \equiv \\
& (y + a + a = AB \wedge b = 2) \vee (b = 1 \wedge y + a = AB) \vee \\
& \quad (y + ab = AB \wedge b \leq 0) \\
H_3(R) = \quad & wp(if, H_2(R)) \vee H_0(R) \\
& \vdots
\end{aligned}$$

We are now in a position to guess $H_k(R)$

$$H_k(R) \equiv (\exists k > 0, \bigvee_{i=1}^{k}(b = i \wedge y + ia = AB)) \vee (y = AB \wedge b \leq 0)$$

We can finish computing the weakest precondition for the program.

$$\begin{array}{ll} wp(y := 0, wp(while, \cdots)) = & ((\exists k > 0, \bigvee_{i=1}^{k}(b = i \wedge 0 + ia = AB))\vee \\ & \quad (0 = AB \wedge b \leq 0)) \\ wp(b := B, wp(\cdots)) = & ((\exists k > 0, \bigvee_{i=1}^{k}(B = i \wedge ia = AB))\vee \\ & \quad (0 = AB \wedge B \leq 0)) \\ wp(a := A, wp(\cdots)) = & ((\exists k > 0, \bigvee_{i=1}^{k}(B = i \wedge iA = AB))\vee \\ & \quad (0 = AB \wedge B \leq 0)) \end{array}$$

We have therefore computed the weakest precondition for the given output to be exactly:

$$B > 0 \vee (0 = AB \wedge B \leq 0)$$

But we can write this as:

$$B > 0 \vee (0 = AB \wedge B = 0) \vee (0 = AB \wedge B < 0)$$

which is equivalent to:

$$B > 0 \vee B = 0 \vee (A = 0 \wedge B < 0)$$

or

$$B \geq 0 \vee (A = 0 \wedge B < 0)$$

4. EXERCISES

Problem 1. Determine a sufficient precondition for the following program and postcondition, then show that this predicate implies ***wp***.

$$
\begin{array}{l}
\quad i := 1 \\
\quad m := a[1] \\
\quad \textbf{while } i \neq 100 \textbf{ do} \\
\qquad \textbf{if } m < a[i+1] \textbf{ then } m := a[i+1] \\
\qquad i := i + 1 \\
\{(\exists j \ni 1 \le j \le 100, m = a[j]) \land (\forall k \ni 1 \le k \le 100, m \ge a[k])\}
\end{array}
$$

Problem 2. Determine a sufficient precondition for the following program and postcondition, then show that this predicate implies ***wp***.

$$
\begin{array}{l}
\quad k := a \\
\quad \textbf{while } k < b \textbf{ do} \\
\qquad k := k + 1 \\
\qquad x[k-1] := x[k] \\
\{a \le k \le b \ \land \ (\forall i \ni a \le i < b, x[i] = \overleftarrow{x}[i+1])\}
\end{array}
$$

Give the precondition if the postcondition were written as:

$$
\{a \le k \le b \ \land \ (\forall i \ni a \le i < b, x[i] = x[i+1])\}
$$

Problem 3. Using predicate transformers verify the following program:

$$
\begin{array}{l}
\{n \ge 0\} \\
\quad i := 1 \\
\quad p := 1 \\
\quad \textbf{while } i \neq n \textbf{ do} \\
\qquad \textbf{if } b[i] \neq b[i-p] \\
\qquad\quad \textbf{then } i := i + 1 \\
\qquad\quad \textbf{else } i := i + 1; p := p + 1 \\
\{(\exists k \ni 0 \le k \le n - p, b[k] = b[k + p - 1]) \land \\
\quad (\forall k \ni 0 \le k \le n - p - 1, b[k] \neq b[k + p])\}
\end{array}
$$

Problem 4. Using the method of predicate transformers, demonstrate the correctness of the following array reversal program:

$$
\begin{array}{l}
\{i = 1 \ \wedge \ n > 0\} \\
\textbf{while } i \leq n/2 \textbf{ do} \\
\quad z := a[i] \\
\quad a[i] := a[n - i + 1] \\
\quad a[n - i + 1] := z \\
\quad i := i + 1 \\
\{\forall j \ni 1 \leq j \leq n, a[j] = \overleftarrow{a}[n - j + 1]\}
\end{array}
$$

Problem 5. Consider the following program:

$$
\begin{array}{l}
\{i = 1\} \\
\textbf{while } i < n \textbf{ do} \\
\quad a[i + 1] := a[i] + a[i + 1] \\
\quad i := i + 1 \\
\{\forall j \ni 1 \leq j \leq n, a[j] = \sum_{k=1}^{j} \overleftarrow{a}[k]\}
\end{array}
$$

Verify the total (strong) correctness of the program with respect to the given pre- and postconditions using the method of predicate transformers.

Problem 6. From the definition of the weakest precondition for the traditional **if then else statement**, show the following is true:

$$
\begin{array}{l}
(B \Rightarrow wp(S_1, R)) \wedge (\neg B \Rightarrow wp(S_2, R)) = \\
(B \wedge wp(S_1, R)) \vee (\neg B \wedge wp(S_2, R))
\end{array}
$$

5. SUGGESTED READINGS

• E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, Vol. 18, No. 8, 1975, pp. 453-458.

• S. K. Basu and R. T. Yeh, "Strong Verification of Programs," *IEEE Transactions on Software Engineering*, Vol. 1, No. 3, 1975, pp. 339-346.

• E. W. Dijkstra, "A Discipline of Programming," Prentice-Hall, Englewood Cliffs, NJ, 1976.

• D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.

Chapter 5

Algebraic Specifications

Up to this point, we have dealt primarily with integer data types in the application programs. But to continue to do so would be very restrictive to programmers. In fact, not only do we want to find ways to reason about programs having many different primitive types (such as strings, characters, or floating point numbers), but we want to build an inference system that allows programs to be verified even when the application programmer is free to enrich the environment with new and special purpose types.

1. MORE ABOUT DATA ABSTRACTIONS

Being able to add new data types on a program by program basis is a powerful capability. This concept, which evolved in the 1970s, is known as *data abstraction* (or, looked at another way, *encapsulation*), and it is a major "divide and conquer" technique available to help programmers keep development costs down. The idea is to identify commonly used activities or information, and then, abstracting from the situation, explicitly describe that information structure in a concise and centralized way. Once the structure has been isolated, then one programmer can go off and implement the data-handling routines, another can work on the application program *using* the data, and both will be assured that the union of their efforts will function as intended.

We previous discussed data abstractions in Chapter 3. Here we extend the concept and discuss why we have this concept of "type" in the first place. Associating a "type" name with some piece of data is a declaration by the programmer concerning how that data is intended to be used. When the only thing we can declare about a datum is that it is an integer or string, then there is still much room for misuse of that information. However, when a designer abstracts some collection of desirable properties and declares this collection to be a new "type," then it is easier to infer more about the intended usage of the data. The programmer can say stronger things about the program components. In general, the way a designer will perform this abstraction activity will vary from methodology to methodology, and this is (currently) well beyond what we study in this course. However, once a new type has been defined it is essential that we be able to

1. verify that implementations of the new type preserve properties that characterize it, and

2. reason about application programs that use the new type.

For our purposes, a **data type** consists a set of values (its domain) and a set of operations on the domain. Defining the domain is usually straightforward: Small, finite domains can be enumerated. Others can be defined constructively, by first identifying a small number of base elements, and then describing how to generate the remaining elements of the domain. For instance, the $string$ domain can be constructed by saying:

- A single letter is a $string$.
- Any $string$ followed by a single letter is a $string$.
- Nothing else is a $string$.

This third assertion helps us get a bound on what can be a $string$.

Once the domain is specified, we are faced with describing the operations on it. The first step is to enumerate these operators' names, and identify in some way what types of values they both operate on (completely) and also return. This step is referred to as prescribing the *syntax*, and the way we give syntax is usually dependent on the way we intend to describe the operator's functionality (or *semantics*).

There are several ways to give semantics. One is quite informal: to give a textual description of the behavior. Due to the ambiguities of natural language, this is generally unsatisfactory. It allows ambiguities, does not lend itself to automatic checking, and requires more space to accomplish the task than other methods. Two other techniques for specifying abstract data types (ADTs) are more formal. These are *operational* specifications and *algebraic* specifications, and are described in detail in the following sections. In each case, we first concentrate on what the specifications look like (along with related issues that come up in the approach), then move on to consider how implementations of that data type are verified. We conclude discussion on each approach with detailed examples of how application programs using the ADTs are themselves verified.

2. OPERATIONAL SPECIFICATIONS

An operational specification describes an operation in terms of another type with well-defined mathematical properties, e.g., sequences, sets, or tuples. The VDL model of Chapter 1 was one general class of operational specification. As another specific example, consider specifying operations to make rational numbers from integers and to add two rational numbers using fractions as the implementation of rational numbers. Each procedure defines one of the operations using **pre** and **post** conditions that define the semantics of the operation. The postcondition is the essential operational semantics of that particular operation.

The meaning of $AddRat$ is defined using the "+" operator on fractions.

$\mathbf{procedure}\ MakeRat(\mathbf{var}\ Result : Rational; N, D : integer);$
$\quad \{\mathbf{pre} : D \neq \mathbf{0}$
$\quad \mathbf{post} : Result = N/D\}$
$\mathbf{procedure}\ AddRat(\mathbf{var}\ Result, Op\mathbf{1}, Op\mathbf{2} : Rational);$
$\quad \{\mathbf{pre} : true$
$\quad \mathbf{post} : Result = Op\mathbf{1} + Op\mathbf{2}\}$

Specifying rational numbers using fractions does not seem like giving an implementation because there is no difference in abstraction between the two types. A better example of operational specifications is the specification of type stack in terms of type sequence[62]. For this,

we need a definition for sequences and the operation which manipulate them.

$< s_1, ..., s_k >$	denotes the sequence of elements specified
$<>$	empty sequence
$s \sim x$	is the sequence resulting from concatenating element x at the end of s
$s_1 \& s_2$	is the sequence resulting from concatenating sequence s_2 after s_1
$length(s)$	the number of elements in s
$first(s)$	the leftmost element of s
$last(s)$	the rightmost element of s
$tail(s)$	the sequence resulting from deleting the first element of s
$lead(s)$	the sequence resulting from deleting the last element of s
$seq(V, n, m)$	$< V[n], V[n+1], ..., V[m] >$; also $seq(V, n, m) = seq(V, n, m-1) \sim V[m]$

Note that first, last, tail, and lead are undefined for $<>$.

Assuming we have this notation for sequences, we define type stack as follows.

const $MaxDepth := \ldots$
type $Stack := \ldots$
$\{$**invariant** $: 0 \leq length(S) \leq MaxDepth\}$
procedure $NewStack($**var**$S : Stack)$;
    $\{$**pre** $: true$
    post $: S = <>\}$
procedure $Push($**var**$S : Stack; E : StackElt)$;
    $\{$**pre** $: length(S) < MaxDepth$
    post $: S = S' \sim E\}$
procedure $Pop($**var**$S : Stack)$;
    $\{$**pre** $: true$
    post $: (length(S') = 0 \wedge S = S') \vee (length(S') > 0 \wedge S = lead(S'))\}$
function $Top($**var**$S : Stack) : StackElt$;
    $\{$**pre** $: length(S) > 0$
    post $: Top = last(S)\}$
function $Empty($**var**$S : Stack) : boolean$;
    $\{$**pre** $: true$
    post $: Empty = (length(S) = 0)\}$

These specifications are relatively easy to construct because they seem like programming, although they have been criticized since they suggest implementations too strongly. Thus, one might argue that operational specifications should be very inefficient implementations so that no one is tempted to use the specification as an implementation.

3. ALGEBRAIC SPECIFICATION OF ADTS

An alternative to operational specification of ADTs is to define them *algebraically*. In this approach, we define the semantics of ADT operators in terms of how they *inter*act with one another, instead of how they act in terms of a concrete type. It may be a bit harder to come up with good axioms in algebraic specification as opposed to operational specifications — after all, in operational specifications we deal with implementations of single operations.

Algebraic specifications are divided into two parts: syntax and semantics. The syntactic description is often referred to as the *signature* of the algebra, and each "auxiliary" data type used to give this signature is called a *sort*. For instance, the sorts for STACK above are ***integers*** and ***booleans***.

$$\begin{array}{ll}
push: & STACK \times INTEGERS \rightarrow STACK \\
pop: & STACK \rightarrow STACK \\
top: & STACK \rightarrow INTEGERS \cup \{undefined\} \\
empty: & STACK \rightarrow BOOLEAN \\
newstack: & \rightarrow STACK
\end{array}$$

The domain of STACK (as opposed to the domain of any of the above operators) may now be described. This can actually be more difficult than it sounds. Informally, we know that we want our type STACK to encompass all the stacks of integers that could arise in some program. Formally, we need a way to describe them. Since we have have listed the operators, we can now identify a subset of the operators as being *constructors* for the type. That is, we want to identify the least number of operators necessary to be able to construct all possible stacks that could come up. Intuitively, we can believe that ***newstack*** and ***push*** are constructors, since any stack that was formed by using a ***pop*** somewhere along the way can be formed by using only a smaller number of calls to ***push***.

Axioms describe the meanings of operators in terms of how they interact with one another. For instance, our ADT STACK can be specified as:

1. $pop(newstack) =$ $newstack$
2. $pop(push(S, I)) =$ S
3. $top(newstack) =$ $undefined$
4. $top(push(S, I)) =$ I
5. $empty(newstack) =$ $true$
6. $empty(push(S, I)) =$ $false$

where ***S*** and ***I*** are considered to be universally quantified as an instance of stack and integer, respectively. Consider Axiom 1 above: Remember we are not declaring that there is some special instance of a stack that is distinguished by being a "newstack." Instead, this axiom is read "the effect of composing a call to ***pop*** and a call to my function ***newstack*** is just the same as having called ***newstack*** directly." Likewise, Axiom 2 is read "the effect of composing a our ***pop*** function with a call to ***push*** on some stack ***S*** and an integer ***I*** yields a stack which is just the same as the stack ***S*** we started with."

This choice of axioms defines an abstract type that we call STACK, and seems to capture the basic semantics of the stack of our intuition. Note what this specification does *not* capture, for instance, boundedness. The above specification allows us to generate arbitrarily deep stacks, which might not correspond to what our underlying implementations should be expected to support. To capture boundedness, we might also define a "hidden function" (i.e. an operator not normally exported for users to access, but which allows us to capture a desired property in our specification) such as ***size***:***STACK***→INTEGERS, then use the following algebra in place of our earlier specification:

1. $pop(newstack) =$ $newstack$
2. $pop(push(S, I)) =$ $if\ size(S) = MAX\ then\ pop(S)\ else\ S$
3. $top(newstack) =$ $undefined$
4. $top(push(S, I)) =$ $if\ size(S) = MAX\ then\ top(S)\ else\ I$
5. $empty(newstack) =$ $true$
6. $empty(push(S, I)) =$ $false$
7. $size(newstack) =$ 0
8. $size(push(S, I)) =$ $if\ size(S) = MAX\ then\ size(S)$
   $else\ size(S) + 1$

where MAX is some agreed-upon parameter.

In general, a list of axioms appears as a set of equations, with each left-hand side containing a composition of operators, and each right-hand side containing a description of how the composition behaves in terms of the type's operators and simple "if-then-else" constructs. Iteration or other auxiliary variables are not used.

These axioms are often thought of as "rewrite rules." Informally, this is because we can take one instance of the ADT and appeal to the list to find a simpler way to express that instance. However, there is a more mathematical criteria for rewrite rules, which will be explored later. In the meantime, the key is that all axioms should take one composition of functions and show how it can be expressed in only one of the arguments or functions, that is, we rewrite the composition in a simpler way.

Notice also that this functional approach to building ADTs treats instances of the type as being "immutable." (In contrast, and from an implementation point of view, we usually think of a stack as being one object which accumulates side effects based on the operations on the stack.) According to the specification above, all the ***push*** operation guarantees is that it returns an instance of STACK which is constructed from another instance of STACK and some INTEGER. This situation, together with our ability to view the axioms as rewrite rules, allows us to think of instances of an algebraically specified ADT as being sentences in a language, a very useful perspective. For example, one sentence might be formed by calling ***newstack***, then applying a ***push***, then another ***push***, then a ***pop***, then applying the query ***empty***, for example,

$$empty(pop(push(push(newstack, 42), 17)))$$

Using Axiom 2 above (from unbounded stacks), we can simplify the outermost composition of ***pop*** and ***push***, and rewrite the sentence as:

$$empty(push(newstack, 42))$$

At this point we can appeal to axiom (6) and simplify the string to be simply ***false***. Note how we were able to squeeze the call to ***pop*** out, and express the argument to ***empty*** using a sentence which only contains calls to either ***push*** or ***newstack***. This illustrates our point in the earlier discussion concerning constructors. We will often refer to

an instance of the ADT which is built only through use of constructor operations as being in *normal* (or *canonical*) *form*. In general, it is very useful to look at intermediate expressions as being strings in a language to which transformations can be applied. This will assist us in our reasoning about programs using algebraically specified ADTs.

3.1. Developing Algebraic Axioms

For the axiomatization of type T to be sufficiently complete, it must assign a value to each term of type T [24]. Since the constructors produce all the values of the domain of the type being defined, then we need only specify how non-constructor operations affect each constructor. Thus if $c1_1$ and c_2 are constructors operations of type T and f is a non-constructor operation whose signature is $r : T \rightarrow T1$, we write axioms to define $f(c_1)$ and $f(c_2)$. For example, for type stack $push$ and $newstack$ are constructors, while pop, top and $empty$ are not constructors. Thus, we wrote axioms to define the meanings of:

$$\begin{array}{l} pop(newstack) \\ pop(push(S,i)) \\ top(newstack) \\ top(push(S,i)) \\ empty(newstack) \\ empty(push(S,i)) \end{array}$$

Consider another pair of examples. Let us define the ADT "SET of integers," having syntax

$$\begin{array}{ll} newset: & \rightarrow SET \\ insert: & SET \times INTEGER \rightarrow SET \\ delete: & SET \times INTEGER \rightarrow SET \\ member: & SET \times INTEGER \rightarrow BOOLEAN \end{array}$$

having semantics

$$\begin{array}{ll} member(newset,i) = & false \\ member(insert(S,i),j) = & if\ i=j\ then\ true\ else\ member(S,j) \\ delete(newset,i) = & newset \\ delete(insert(S,i),j) = & if\ i=j\ then\ delete(S,j) \\ & else\ insert(delete(S,j),i) \end{array}$$

Alternately, we can define the ADT "BAG of integers" as:

$$\begin{array}{ll} newbag: & \rightarrow BAG \\ insert: & BAG \times INTEGER \rightarrow BAG \\ delete: & BAG \times INTEGER \rightarrow BAG \\ member: & BAG \times INTEGER \rightarrow BOOLEAN \end{array}$$

having semantics

$$\begin{array}{ll} member(newbag, i) = & false \\ member(insert(B, i), j) = & if\ i = j\ then\ true\ else\ member(S, j) \\ delete(newbag, i) = & newbag \\ delete(insert(B, i), j) = & if\ i = j\ then\ B \\ & else\ insert(delete(B, j), i)) \end{array}$$

It would be very useful for you to go through a few simple examples using these two close but different sets of axioms. Build a few sentences in the "language" of the type, and verify for yourself that, for instance, you can add several copies of the same integer to a set, and that after removing it you will be able to ask "member" of that set and integer and get the right answer. In the next subsections, we give a more complete development of a set of axioms.

We have more choices for axioms when defining operations which have more than one argument of the type being defined. The examples below illustrate several different sets of axioms defining add for the type natural numbers generated by 0 and $succ$. Each of the sets of axioms are sufficiently complete.

$$\begin{array}{ll} add(\mathbf{0}, X) = & X \\ add(succ(Y), Z) = & succ(add(Y, Z)) \end{array}$$

$$\begin{array}{ll} add(\mathbf{0}, \mathbf{0}) = & \mathbf{0} \\ add(succ(X), \mathbf{0}) = & succ(X) \\ add(Y, succ(Z)) = & succ(add(Y, Z)) \end{array}$$

$$\begin{array}{ll} add(\mathbf{0}, \mathbf{0}) = & \mathbf{0} \\ add(\mathbf{0}, succ(W)) = & succ(W) \\ add(succ(X), \mathbf{0}) = & succ(X) \\ add(succ(Y), succ(Z)) = & succ(succ(add(Y, Z))) \end{array}$$

Huet and Hullot describe an algorithm for determining if the set of arguments of an operation's axioms is sufficiently complete. [33] For each operation, a set of n-tuples representing the operation's arguments is constructed. Initially all the arguments in the first positions of the axioms for a particular operation are examined to determine that they "cover" the set of constructors, either by containing an instance of each constructor or by containing a variable which can represent any constructor. For example, the second set of axioms defining add have the following set of argument two-tuples: $\{ <0,\ 0>, <succ(X),\ 0)>, <Y,\ succ(Z)> \}$. We consider the arguments in the first positions: $0,\ succ(X)$, and Y, and conclude that all constructors are present and that $succ$'s argument covers all its constructors because it is a variable. If all the constructors are present, we consider each set of n-1-tuples formed from that subset of the axioms having the first constructor or a variable in their first positions, followed by the set of n-1-tuples formed from the subset of axioms having the second constructor or a variable in their first positions, etc. until we have processed all the constructors. Again, by example, we consider the sets of 1-tuples $\{<0>, <succ(Z)> \}$ (whose first arguments were 0 or Y) and $\{<0>, <succ(Z)>\}$ (whose first arguments were $succ(X)$ or Y). Since all the constructors are present in each of these sets and $succ$'s argument covers all its constructors, the axioms are sufficiently complete.

Huet and Hullot's algorithm is given below.

function complete(F: $\{f_1,...,f_m\}$ where $f_i = < s_i^1, ..., s_i^k >$): boolean;
  for every f_i do if f_i contains duplicate variables then
    return(false);
  if k = 0 and F = $\{ <> \}$ then return(true);
  for i in $1..m$ do
    if s_i^1 is a variable and complete($\{< s_i^2, ..., s_i^k >| \ s_i^1$
      is a variable$\}$) then return(true);
  for every constructor c(p_1,...,p_n) do
    if there exists one s_i^1 with leading function symbol c and
    complete($\{< p_1, ..., p_n, s_i^2, ..., s_i^k >| \ s_i^1$ = c(p_1,...,p_n)$\}$
        $\cup \{< X_1, ..., X_n, s_i^2, ..., s_i^k >|$
          s_i^1 is a variable and X_j's are fresh variables$\}$)
      then skip
      else return(false);
  return(true)

3.2. Hints For writing algebraic axioms

The process of writing algebraic specifications is similar to that of programming.[1] The main difference is that in programming we are interested in the efficiency of the solutions, whereas in specifying we are interested in expressing the functionality and nothing else (i.e., avoid implementation bias). This means that in writing specifications we want to express only the function to be computed, not the method of computation. Besides, algebraic specification languages do not have the concept of side effect (e.g., variable assignment) every time a new value is created, so algebraic specifications resemble so-called functional programs.

Determine functionality

You should determine the use of the data type, what operations create instances of the data type, what operations modify instances,[2] what operations obtain values from the data type, etc. To determine these functions you must think about the uses of the data type. This might be hard, especially if this is a "general-purpose" type and you cannot anticipate all its uses.

Write signatures

Once you have decided which is the set of functions, you write the signatures of them. In this step you can provide a lot of insight on the abstract type if you use appropriate, meaningful names. Note, however that the final word on meaning is not the name: *This is the whole point of using formal specifications*.

Choose constructors

Initially all functions that yield the defined type are candidates for constructors. In principle, you can have them all as constructors, but that will introduce many problems both in the consistency of your axioms and also in the number of axioms (equations) that you have to provide.

[1] We are indebted to Pablo Straub of the Catholic University of Chile for contributing this part.

[2] In the specification no modification is allowed, but modification is expressed by creation of a new instance that is very similar to the original.

Hence you want a "minimal" set of constructors. This set of constructors will have at least one function that has no parameter of the type defined (usually it will be a function with no parameters at all).

If you can figure out how to express the semantics of a function in terms of others, then this function is not a constructor. Besides, the way you express the semantics of the function will dictate the axioms you will write later.

Sometimes which operations are constructors is evident from the specification; sometimes there are explicit choices to be made. For example, consider a specification for a dequeue, a list with insertion and deletion on both ends. We have two possible sets of constructors: {AddFront, New} and {AddBack, New}. Which one to use is a matter of taste, but you should decide on one of them.

Write axioms

The first part is to determine which axioms you need to write. Constructors do not need axioms: you will only write axioms for nonconstructors. For each nonconstructor function, you need to cover all cases of the parameters. As a rule of thumb, if there are n constructors you will need n equations for each nonconstructor.

This rule of thumb is not complete, though, because sometimes there is more than one parameter of the defined type, so you may need more axioms. For example if you are defining a function with 3 parameters of the type defined and you have 2 constructors, you may need $2^3 = 8$ equations to cover all constructor combinations. Besides, sometimes you need to consider nesting of constructors to discriminate different cases. Fortunately, this rarely occurs in practice, and you will usually be able to cover all cases with just a few equations.

Now that you know all cases to consider, you write the left-hand side of all axioms. There are no general methods to write the right-hand side, though. While writing the axioms you will notice that some left hand sides are meaningless (e.g., extract information from an empty data structure). From this you define your exceptions.

Example: Finite integer functions

We want to design a data structure to represent finite integer functions. A finite function is a function whose domain is finite, that is, the function is defined in a finite number of points. Finite functions can be represented by a lookup table whose entries are ordered pairs. In this case these ordered pairs will be pairs of integers.

Note that even though here we are defining the data structure for the case of integers, the specification that we will develop can be parameterized for finite functions with any domain and range.

Determine functionality

We want to be able to manipulate finite mappings by creating simple mappings, modify mappings at a specific point (define the value, make the value undefined), and make the union of mappings. The union of two mappings is not commutative, because if both mapping are defined at a particular point, the definition from the second mapping takes precedence. We need to be able to handle empty mappings and also have a function to create a single-entry mapping. Obviously, given an integer, we want to know the value of the function at that point. Besides, given a mapping, we want to get its domain.

Writing signatures

Given the requirements expressed before, define the following functions with their corresponding signatures.

EmptyMapping:		$\to$ mapping
SingleMapping:	integer $\times$ integer	$\to$ mapping
Define:	mapping $\times$ integer $\times$ integer	$\to$ mapping
Undefine:	mapping $\times$ integer	$\to$ mapping
Union:	mapping $\times$ mapping	$\to$ mapping
Apply:	mapping $\times$ integer	$\to$ integer
Domain:	mapping	$\to$ set[integer]

Choosing constructors

Candidates for constructors are:

$$EmptyMapping, SingleMapping, Define, Undefine, Union$$

Of them, either $EmptyMapping$ or $SingleMapping$ must be chosen as constructor, because these are the only ones that do not have a parameter of type mapping. $SingleMapping$ can be easily expressed by defining an $EmptyMapping$. Further thought can lead us to choose $EmptyMapping$ and $Define$ as the constructors.

Writing axioms

First, consider which axioms we need. Our rule of thumb tells us that we need two axioms for each nonconstructor, with the caveat that $Union$ might require four axioms. Besides, since $SingleMapping$ has no parameters of type mapping, we can define it with only one axiom.

The left-hand sides are then:

$$SingleMapping(i, v) =$$

$$Undefine(Empty, i) =$$
$$Undefine(Define(m, i_1, v), i_2) =$$

$$Apply(Empty, i) =$$
$$Apply(Define(m, i_1, v), i_2) =$$

$$Domain(Empty) =$$
$$Domain(Define(m, i, v)) =$$

We delay the left-hand sides for the Union operation because we do not know yet whether we will be able to do it with two or four equations.

Now we define the semantics of all operations except Union.

$$\begin{array}{ll}
SingleMapping(i, v) = & Define(Empty, i, v) \\
Undefine(Empty, i) = & Empty \\
Undefine(Define(m, i_1, v), i_2) = & if\ i_1 = i_2\ then\ Undefine(m, i_2) \\
 & else\ Define(Undefine(m, i_2), i_1, v) \\
Apply(Empty, i) = & undefined \\
Apply(Define(m, i_1, v), i_2) = & if\ i_1 = i_2\ then\ v\ else\ Apply(m, i_2) \\
Domain(Empty) = & NullSet \\
Domain(Define(m, i, v)) = & AddMember(Domain(m), i)
\end{array}$$

Now let us consider function $Union$. Obviously union with the $Empty$ mapping does not modify the mapping. So as long as one of the mappings is empty the result is trivial. What happens with two nonempty

mappings? We want the value of the second mapping to take precedence. Here we present 4 equations to consider all possible cases.

$$
\begin{aligned}
&Union(Empty, Empty) = Empty\\
&Union(Define(m,i,v), Empty) = Define(m,i,v)\\
&Union(Empty, Define(m,i,v)) = Define(m,i,v)\\
&Union(Define(m_1,i_1,v_1), Define(m_2,i_2,v_2)) =\\
&\quad Define(Union(Define(m_1,i_1,v_1), m_2), i_2, v_2)
\end{aligned}
$$

By looking at the complex specification of the $Union$ function we note that we can use only two equations, because the first parameter is never used in recursion: It is always taken *in toto*. If the second parameter is $Empty$ we always return the first; if the second parameter is not $Empty$ we add all of its definitions to the first one.

The new equations are these

$$
\begin{aligned}
Union(m, Empty) = &\quad m\\
Union(m_1, Define(m_2,i,v)) = &\quad Define(Union(m_1,m_2), i, v)
\end{aligned}
$$

3.3. Consistency

Axioms sets are *consistent* if we always get the same final result, independent of the order in which we apply the axioms to simplify terms. As an example (adapted from [27]), consider adding the following axiom to the definition of BAGs above:

$$member(delete(S,i),j) = if\ i = j\ then\ false\ else\ member(S,j)$$

Then consider the simplification of string:

$$member(delete(insert(insert(newbag,3),3),3),3)$$

By first applying our new axiom, this can be rewritten as

$$if\ 3 = 3\ then\ false\ else\ member(\ldots)$$

or ***false***. However, by our original ***delete(insert())*** axiom, this can become

$$member(if\ 3 = 3\ then\ insert(newbag,3)\ else\ delete(\ldots)$$

or

$$member(insert(newbag, 3), 3)$$

or

$$if\ 3 = 3\ then\ true\ else\ member(\ldots)$$

or ***true***, a contradiction. In general, questions of consistency can be answered by using the Knuth–Bendix algorithm [38], which is described in detail later in this chapter.

3.4. Term Equality

Other interesting predicaments arise. Consider two strings for SETs,

insert(insert(newset, 3), 4)

and

insert(insert(newset, 4), 3)

Intuitively we want the sets represented by these two strings to be equal. After all, we see each as being a set of two elements, 3 and 4, and we know a property of sets is that there is no order property. However, it is impossible to prove this using only the axioms given to define the SET semantics.

In general, questions of equality are very difficult to deal with. In so-called "initial algebras," two sentences are considered equal only if they can be proven so as a consequence of axioms. Some researchers, such as Guttag [17], say that two sentences are equal *unless* one of the operations mapping out of the type of interest carries them to distinct values. In effect, the question of equality is then punted into the equality interpretation of the ADT's sorts. Regardless, when you discuss equality in the manipulation of abstract data types, it is essential to be absolutely clear as to "which equals" you are talking about.

4. DATA TYPE INDUCTION

Data type induction is an important technique in verifying properties of algebraic specifications. This induction works as follows, and is closely related to natural induction of the positive integers.

Given the set of operations defined on some ADT X, we can separate them into three sets:

1. $\{f_i\}$ are those functions which return objects of type X but do not contain arguments of type X. These generating functions create the primitive objects of type X.

2. $\{g_i\}$ are those functions which return objects of type X and do contain an argument of type X. These generating function creates non-primitive objects of type X.

3. $\{h_i\}$ are all other functions, generally taking arguments of type X and returning values of some other type. These represent uses of the type.

For the stack ADT, we get:

$$\begin{aligned} f = & \quad \{newstack\} \\ g = & \quad \{push, pop\} \\ h = & \quad \{empty, top, size\} \end{aligned}$$

In general there will be one f member, one or two g members, and a small number of h members, but there are no prohibitions on more than these.

Let $P(x)$ be some predicate concerning $x \in X$ for ADT X. Under what conditions will P be true for every member of type X? Similar to natural induction, we want to show that primitive objects are true under predicate P and that we can use these primitive values to show that by constructing new values of the type, the property remains true.

We can define *data type induction* as follows:

- Given ADT X with functions f_i, g_i and h_i defined as above and predicate $P(x)$ for $x \in X$.
- Show that $P(f_i)$ is valid (base case).
- Assume $P(x)$ is valid. Show that this implies $P(g_i(x))$ is valid (i.e., $P(x) \vdash P(g_i(x))$).
- We can then conclude $\forall x \in X \ P(x)$.

For the stack ADT, this would mean that we show $P(newstack)$ and $P(x) \vdash P(g_i(x))$ (i.e., both $P(push(x,i))$ and $P(pop(x))$.) However, we do not need to show this for pop since pop is not needed to generate all

stacks. The *normal-form lemma* gives us this. This lemma is a formal assertion of what we have already informally claimed earlier, namely that all instances of a STACK can be built using only calls to $newstack$ and $push$; that is, each stack has a normal form.

Lemma: For every $S \in$ STACK, either

1. $S =$ ***newstack***, or
2. $\exists S_1 \in$ STACK and $I \in$ INTEGER such that $S = push(S_1, I)$

(The uniqueness of this normal form requires a separate proof, and is left as an exercise.).

This lemma simply says that we can replace any pop operation by an equivalent set of $newstack$ or $push$ operations. The proof of this lemma is quite complex and beyond the scope of this book. It depends upon the following argument:

1. The STACK ADT is primitive recursive and each object is constructible by a finite application of the constructors.
2. "Count" the number of function applications in constructing stack S.
3. For each instance of $pop(S_i)$, consider the definition of S_i. If it is of the form $newstack$, then $pop(newstack) = newstack$. If it is of the form $pop(push(S_j, j))$, then by axiom, this is just S_j. If it is of the form $pop(pop(S_j))$ apply the same argument to S_j until you get a $pop(newstack)$ or $pop(push(\ldots))$.
4. In all of the above cases, the "count" of function applications by applying these reductions will be less. Thus, the process must terminate.
5. This can be repeated as long as a pop operation remains. So all pop operations can be removed from any stack description.

This explanation was generally intuitive. See the Guttag and Horning reference [24] at the end of the chapter for complete details on the proof of the normal form lemma.

This normal form lemma is not essential for verifying properties of an ADT. However, proving such a lemma is often a way to reduce the amount of work associated with verifications because arguments which

previously needed to be made for ***all*** operators returning the type of interest can then be replaced by an argument for only the ***constructors*** of the type.

4.1. Example: Data Type Induction Proof

Using data type induction, we can prove that pushing a value on a stack increases its size by proving the theorem $size(push(S, X)) > size(S)$. The axioms for stacks are shown below.

$$\begin{array}{ll} axiom\ top_1: & top(newstack) = undefined \\ axiom\ top_2: & top(push(S, I)) = I \\ axiom\ size_1: & size(newstack) = 0 \\ axiom\ size_2: & size(push(S, I)) = size(S) + 1 \\ axiom\ pop_1: & pop(newstack) = newstack \\ axiom\ pop_2: & pop(push(S, I)) = S \end{array}$$

We start the proof of $size(push(S, X)) > size(S)$ by using the axioms until no more simplifications are possible.

$$\begin{array}{ll} size(push(S, X)) > size(S) & \\ size(S) + 1 > size(S) & by\ size_2 \end{array}$$

At this point, we resort to data type induction. First we replace S by $newstack$ and show the theorem holds.

$$\begin{array}{ll} Begin\ induction\ on\ S = newstack & \\ size(newstack) + 1 > size(newstack) & replace\ S\ with\ newstack \\ 0 + 1 > 0 & by\ size_1 \\ true & \end{array}$$

Next, we assume that the theorem holds for S', i.e, $size(S') + 1 > size(S')$, and show the theorem holds for $S = push(S', X)$.

$$\begin{array}{ll} Induction\ on\ S = push(S', X) & \\ Inductive\ hyp.: size(S') + 1 > size(S') & \\ size(push(S', X)) + 1 > size(push(S', X)) & push(S', X)\ replaces\ S \\ size(S') + 1 + 1 > size(S') + 1 & by\ size_2 \\ size(S') + 1 > size(S') & subtract\ 1\ from\ each\ side \\ true & by\ inductive\ hypothesis \end{array}$$

5. VERIFYING ADT IMPLEMENTATIONS

5.1. Verifying Operational Specifications

Proving that operations are implemented correctly is difficult because of the abstraction gap between the objects in the specifications (sequences) and those in the implementation (arrays and integers). We add two new pieces of documentation to bridge this gap: a representation mapping (R) that maps implementation objects to specification objects, and implementation-level input and output assertions for each operation. For the stack example, these are:

$\mathbf{const}\ MaxDepth := \ldots$
$\mathbf{type}\ Stack := \mathbf{record}\ Sp : \mathbf{0}..MaxDepth;$
$\quad V : \mathbf{array}[\mathbf{1}..MaxDepth]\ \mathbf{of} \ldots \mathbf{end}$
$\{\mathbf{invariant} : \mathbf{0} \leq S.Sp \leq MaxDepth\}$
$\{\mathbf{representation\ mapping} : R(S.V, S.Sp) = seq(S.V, \mathbf{1}, S.Sp)\}$
$\mathbf{procedure}\ NewStack(\mathbf{var} S : Stack);$
$\quad \{\mathbf{in} : true$
$\quad \mathbf{out} : S.Sp = \mathbf{0}\}$
$\mathbf{procedure}\ Push(\mathbf{var} S : Stack; E : StackElt);$
$\quad \{\mathbf{in} : S.Sp < MaxDepth$
$\quad \mathbf{out} : S.V = alpha(S'.V; S'.Sp + \mathbf{1} : E) \wedge S.Sp = S'.Sp + \mathbf{1}\}$
$\mathbf{procedure}\ Pop(\mathbf{var} S : Stack);$
$\quad \{\mathbf{in} : true$
$\quad \mathbf{out} : (S'.Sp = \mathbf{0} \wedge S.Sp = S'.Sp) \vee$
$\quad\quad (S'.Sp > \mathbf{0} \wedge S.Sp = S'.Sp - \mathbf{1})\}$
$\mathbf{function} Top(\mathbf{var} S : Stack) : StackElt;$
$\quad \{\mathbf{in} : S.Sp > \mathbf{0}$
$\quad \mathbf{out} : Top = S.V[S.Sp]\}$
$\mathbf{function} Empty(\mathbf{var} S : Stack) : boolean;$
$\quad \{\mathbf{in} : true$
$\quad \mathbf{out} Empty = (S.Sp = \mathbf{0})\}$

The verification steps are:

• Prove that any initialized concrete object is a well-formed abstract object, that is, if any implementation object X satisfies the concrete invariant, its corresponding abstract value (formed by applying the representation mapping) satisfies the abstract invariant.
$CI(X) \Rightarrow AI(R(X))$

• For each operation, show that its implementation is consistent with its in and out conditions:

$$in(X) \wedge CI(X)\{operation\ body\}out(X) \wedge CI(X)$$

This proof is identical to those presented in Chapter 2. However, if $post(X)$ contains ghost variables, we may assertions to $in(X)$ equating input values with their respective ghost variables.

• For each operation, show its abstract precondition guarantees that its concrete precondition is true, and that its concrete postcondition guarantees that its abstract postcondition is true:

$$\begin{array}{ll} a. & CI(X) \wedge pre(R(X)) \Rightarrow in(X) \\ b. & CI(X) \wedge pre(R(X')) \wedge out(X) \Rightarrow post(R(X)) \end{array}$$

Example verifications

1. $CI(X) \Rightarrow AI(R(X))$

$$0 \le S.Sp \le MaxDepth \Rightarrow 0 \le length(seq(S.V, 1, S.Sp)) \le MaxDepth$$
$$0 \le S.Sp \le MaxDepth \Rightarrow 0 \le S.Sp \le MaxDepth$$

2. $\{in(X) \wedge CI(X)\}$ Push's body $\{out(X) \wedge C(X)\}$

```
procedure Push(var S : Stack; E : StackElt);
    {in : S.Sp < MaxDepth
    out : S.V = alpha(S'.V; S'.Sp + 1 : E) ∧ S.Sp = S'.Sp + 1}
    begin
        S.Sp := S.Sp + 1;
        S.V[S.Sp] := E
    end
```

Since the post-condition contains ghost variables $S.V$ and $S'.Sp$, we add assertions to the pre-condition equating $S.V$ and $S.V$, and $S.Sp$ and $S'.Sp$ to obtain the pre-condition:

$$\textbf{in}: S.V = S'.V\ land\ S.Sp = S'.Sp\ \wedge\ S.Sp < MaxDepth$$

First we use the array assignment axiom:

$$Q \triangleq \{\alpha(S.V; S.Sp : X) = \alpha(S'.V; S'.Sp + \mathbf{1} : X) \wedge S.Sp = S'.Sp + \mathbf{1})\} S.V[S.Sp] := E$$
$$post \triangleq \{(S.V = \alpha(S'.V; S'.Sp + \mathbf{1} : E) \wedge S.Sp = S'.Sp + \mathbf{1})\}$$

Next, we use the assignment axiom:

$$P \triangleq \{\alpha(S.V; S.Sp + \mathbf{1} : X) = \alpha(S'.V; S'.Sp + \mathbf{1} : X) \wedge S.Sp + \mathbf{1} = S'.Sp + \mathbf{1})\} S.Sp := S.Sp + \mathbf{1}$$
$$Q \triangleq \{\alpha(S.V; S.Sp : X) = \alpha(S'.V; S'.Sp + \mathbf{1} : X) \wedge S.Sp = S'.Sp + \mathbf{1})\}$$

By the rule of composition:

$$\frac{\{P\} S.Sp := S.Sp + \mathbf{1} Q, Q\{S.V[S.Sp] := E\{post\}}{\{P\} S.Sp := S.Sp + \mathbf{1}; S.V[S.Sp] := E\{post\}}$$

Finally, we invoke the rule of consequence:

$$\frac{pre \Rightarrow P, \{P\} S.Sp := S.Sp + \mathbf{1}; S.V[S.Sp] := E\{post\}}{\{pre\} S.Sp := S.Sp + \mathbf{1}; S.V[S.Sp] := E\{post\}}$$

3a. $CI(X) \wedge pre(R(X)) \Rightarrow in(X)$

$$\mathbf{0} \leq S.Sp \leq MaxDepth \wedge length(seq(S.V, \mathbf{1}, S.Sp)) < MaxDepth \Rightarrow S.Sp < MaxDepth$$

$$\mathbf{0} \leq S.Sp \leq MaxDepth \wedge length(seq(S.V, \mathbf{1}, S.Sp)) < MaxDepth \Rightarrow S.Sp < MaxDepth$$

Rewriting $length$ by its definition yields:

$$\mathbf{0} \leq S.Sp \leq MaxDepth \wedge S.Sp < MaxDepth \Rightarrow S.Sp < MaxDepth$$

3b. $CI(X) \wedge pre(R(X')) \wedge in(X) \Rightarrow post(R(X))$

$$\begin{array}{l} \mathbf{0} \leq S.Sp \leq MaxDepth \wedge length(seq(S'.V, \mathbf{1}, S'.Sp)) < MaxDepth \wedge \\ S.V = \alpha(S'.V; S'.Sp + \mathbf{1} : E) \wedge S.Sp = S'.Sp + \mathbf{1} \Rightarrow \\ \quad seq(S.V, \mathbf{1}, S.Sp) = seq(S'.V, \mathbf{1}, S'.Sp) \sim E \end{array}$$

Rewriting length by its definition yields:

$$\begin{array}{l} \mathbf{0} \leq S.Sp \leq MaxDepth \wedge S'.Sp < MaxDepth \wedge \\ S.V = \alpha(S'.V; S'.Sp + \mathbf{1} : E) \wedge S.Sp = S'.Sp + \mathbf{1} \Rightarrow \\ \quad seq(S.V, \mathbf{1}, S.Sp) = seq(S'.V, \mathbf{1}, S'.Sp) \sim E \end{array}$$

Replacing S.V and S.Sp on the right side of the equation yields:

$$\begin{array}{l} \mathbf{0} \leq S.Sp \leq MaxDepth \wedge S'.Sp < MaxDepth \wedge \\ S.V = \alpha(S'.V; S'.Sp + \mathbf{1} : E) \wedge S.Sp = S'.Sp + \mathbf{1} \Rightarrow \\ \quad seq(\alpha(S'.V; S'.Sp + \mathbf{1} : E), \mathbf{1}, S'.Sp + \mathbf{1}) = seq(S'.V, \mathbf{1}, S'.Sp) \sim E \end{array}$$

From the definition of sequence, $seq(V, n, m) = seq(V, n, m - \mathbf{1}) \sim V[m]$. Thus

$$\begin{array}{l} seq(\alpha(S'.V; S'.Sp + \mathbf{1} : E), \mathbf{1}, S'.Sp + \mathbf{1}) = \\ seq(\alpha(S'.V; S'.Sp + \mathbf{1} : E), \mathbf{1}, S'.Sp) \sim \\ \quad \alpha(S'.V; S'.Sp + \mathbf{1} : E)[S'.Sp + \mathbf{1}] \end{array}$$

By the definition of the α notation:

$$\begin{array}{l} \alpha(S'.V; S'.Sp + \mathbf{1} : E)[S'.Sp + \mathbf{1}] = E \\ seq(\alpha(S'.V; S'.Sp + \mathbf{1} : E), \mathbf{1}, S'.Sp) = seq(S'.V, \mathbf{1}, S'.Sp) \end{array}$$

Thus the previous formula can be written as:

$$seq(\alpha(S'.V; S'.Sp + \mathbf{1} : E), \mathbf{1}, S'.Sp + \mathbf{1}) = seq(S'.V, \mathbf{1}, S'.Sp) \sim E$$

Finally, the right side of the formula reduces to $true$.

$$\begin{array}{l} \mathbf{0} \leq S.Sp \leq MaxDepth \wedge S'.Sp < MaxDepth \wedge \\ S.V = \alpha(S'.V; S'.Sp + \mathbf{1} : E) \wedge S.Sp = S'.Sp + \mathbf{1} \Rightarrow \\ \quad seq(S'.V, \mathbf{1}, S'.Sp) \sim E = seq(S'.V, \mathbf{1}, S'.Sp) \sim E \end{array}$$

5.2. Verifying Algebraic Specifications

In operationally specified ADTs, the actions of operators were described in terms of a distinct, concrete type; hence, their implementation is easily discussed in terms of only that type. In contrast, the operators of an algebraically specified ADT are described by how they interrelate with the other operators. There is no second (let alone concrete) type used in this case. Therefore, to give an implementation of an algebraically specified ADT, it is essential that we first describe the properties of a second data type that will be used to implement the first. In keeping with the spirit of this abstraction activity, we usually give an algebraic specification for this second type as well.

We often refer to the type we are trying to implement as a "higher level" data type, and to the type used in the implementation as a "lower level" data type. However, these phrases are vague, and they only result from the way we naturally think about the implementation process. In fact, the lower level type might itself be an elaborate ADT with rich semantics. It, in turn, might be implemented in terms of yet a lower-level type, and so on. A hierarchy of data types can often result in this manner. The goal of course is for one of these lower level types to ultimately be directly representable and implemented in hardware (for instance, we will shortly show how to implement the STACKs specified earlier in terms of an algebraically specified array, and we would expect that this array could be directly implemented in our host language on most computers).

However, there is one glaring problem associated with this type hierarchy, and that is how to check whether a lower-level data type is faithfully implemented on a real machine. In the example we are about to give (STACKs in terms of arrays), our specification of array assumes that we have an unbounded array, and further, that uninitialized entries in the array can be detected. These properties are rarely true of machine implementations. Hence, our techniques are only as reliable as is our implementation of the base types. Nonetheless, if we do develop a specification which can be reliably implemented on a machine, then all the benefits of a verified implementation can be enjoyed. One of the most important of these benefits is that the machine-specific properties that must be captured are very *localized*, that is, developers will be free to use instances of our STACK data type, and it will be transparent whether they are implemented using arrays (on some hardware) or actual stacks (on a stack machine).

Once we have the specification for both the higher- and lower-level types, we can describe how the one is to be implemented in terms of the other. This is done through an *implementation program*, which, just as with the original specification of the individual ADTs, looks like a list of axioms. However, whereas before the axioms showed how the operators from any one type lnterrelated, the implementation program shows how the higher-level type behaves in terms of the lower-level type.

In order to complete this implementation program, there is one additional, and quite essential, function which must be provided: the *representation function*. Remember that the operators from each individual type are functions that map into different ranges. It would be absurd to make an implementation program that then equates the range of one ADT's operator with the range of a different ADT's operator. The program only makes sense when there is a representation function which shows how to map an instance of the lower-level type into an instance of the higher-level type. Then, more precisely, we can show how the operators of the higher-level type behave *in terms of the range of the representation function applied to operators of the lower level type*.

Another subtle requirement that is placed on us is to come up with an *equivalence interpretation*. The implementation program is a list of equations, as described above. But remember that, in general, questions of equality are very difficult to resolve. If we are to have any hope of being able to verify properties of this program, then we must have a reliable way to answer questions of equality *in terms of the lower-level type*. The emphasis is for an important distinction: An implementation and its associated equivalence interpretation do not resolve the problem of checking equality within just the higher level type system. They will only allow us to check whether similar operations on instances of the lower-level type will produce (through the representation function) identical sentences in the language of the higher-level type.

We are now ready to summarize what it means to verify an implementation of some ADT. The most obvious requirement is that the axioms for our type of interest are preserved through the implementation. Since the only way we mathematically characterize our ADT is through how they interrelate with one another, the only place we can turn to check the reasonableness of an implementation is exactly this body of rules. These must be checked. A less obvious proof obligation is to verify that our equality interpretation is in fact an equivalence relation. Finally,

we must prove a representation invariant, that is, insure that all elements of our ADT domain can in fact be represented as the image of the representation function applied to some instances of the lower level type.

5.3. Example: Verifying an Implementation of Stacks

Using a technique developed by Guttag, Horowitz, and Musser [25], we now illustrate the idea of implementing one ADT in terms of another and then verifying the correctness of that implementation by showing how the ADT stack defined previously may be implemented in terms of arrays. To do this, our first step must be to formulate an axiomatization of array. Its syntax will be:

$$\begin{array}{ll} newarray: & \rightarrow array \\ assign: & array \times integer \times integer \rightarrow array \\ access: & array \times integer \rightarrow integer \cup \{undefined\} \end{array}$$

and its semantics will be simply:

$$\begin{array}{ll} access(newarray, I) = & undefined \\ access(assign(array, I_1, val), I_2) = & if\ I_1 = I_2\ then\ val \\ & \quad else\ (access(array, I_2)) \end{array}$$

Note that this is a somewhat simplistic view of what arrays are. In particular, this is an unbounded array, which is useful only for purposes of introduction. As we discussed in the previous section, it will be important for us to ultimately give our implementation of STACKs in terms of a data type which is in turn more closely implemented directly by the underlying computer.

We choose our representation function to be:

$$STAK(array, integer) \rightarrow STACK$$

and hence we may give our implementation program:

$$\begin{array}{ll} newstack = & STAK(newarray, \mathbf{0}) \\ push(STAK(arr, t), x) = & STAK(assign(arr, t + \mathbf{1}, x), t + \mathbf{1}) \\ pop(STAK(arr, t)) = & if\ t = \mathbf{0}\ then\ STAK(arr, \mathbf{0}) \\ & \quad else\ (STAK(arr, t - \mathbf{1})) \\ top(STAK(arr, t)) = & access(arr, t) \\ empty(STAK(arr, t)) = & (t = \mathbf{0}) \end{array}$$

Our representation invariant can now be declared to be the predicate

$$P(S) \;=\; \exists A \in ARRAY,\; \exists i \in INTEGERS, i \geq 0,\; S = STAK(A, i)$$

where S is a free variable corresponding to an element of STACK.

In order to insure that the correspondence between an element of STACK and a pair $(array, i)$ is reasonable, we must show each of:

$$P(newstack)$$
$$P(S) \Rightarrow P(push(S, x))$$
$$P(S) \Rightarrow P(pop(S))$$

Since we have already proven a normal form lemma for stacks, we only have to prove the first two of these assertions. So:

• Need to show $P(newstack)$, that is:

$$\exists A \in ARRAY, \exists i \in INTEGERS, newstack = STAK(A, i)$$

Based on our implementation, all we need do to show this is simply choose A to be $newarray$, and i to be 0.

• Need to show $P(S) \Rightarrow P(push(S, x))$. We know $S = STAK(A, i) \wedge i \geq 0$ from $P(S)$, and need to show $\exists A'$ and i' such that $push(S, x) = STAK(A', i')$. But $push(S, x) = STAK(assign(A, i+1, x), i+1)$. We can show the desired assertion simply by choosing A' to be $assign(A, i+1, x)$ and i' to be $i + 1$, with $i' \geq 0$.

Equality

Our equality interpretation is chosen as follows:

$$STAK(A_1, t_1) = STAK(A_2, t_2)$$

if and only if

$$(t_1 = t_2) \;\wedge\; \forall k(1 \leq k \leq t_1, access(A_1, k) = access(A_2, k))$$

Our proof obligation here is to show that this is a "reasonable" interpretation, that is, it has the properties of being an equivalence relation:

1. $reflexivity:\quad x = x$
2. $symmetry:\quad x = y \;\Rightarrow\; y = x$
3. $transitivity:\quad x = y \;\wedge\; y = z \;\Rightarrow\; x = z$

and, so that we may manipulate expressions in our word algebra safely, we must show that the relation preserves substitutivity, i.e.

$$x = y \Rightarrow P(x) = P(y)$$

where P can be any of our operators. So, in turn:

• **reflexivity:** We need to show that $S = S$ for a stack S and our equivalence interpretation. We know S is in the range of STAK, based on our representation invariant, that is, $S = STAK(A,t)$. But $A = A$ and $t = t$ based on our usual interpretation of equality (over integers and arrays), hence $STAK(A,t) = STAK(A,t)$ in our desired equality interpretation.

• **symmetry:** We need to show that $S_1 = S_2$ implies $S_2 = S_1$. Since $S_1 = S_2$ under our equivalence interpretation, the arrays and integers which map into these stacks under STAK are themselves equal. Since the equality of integers and arrays is symmetric, then $S_2 = S_1$ is also true in our equality interpretation.

• **transitivity:** Precisely the same argument as given for symmetry will show transitivity holds.

• **substitutivity:** We need to show each of:

$$\begin{array}{l} S_1 = S_2 \Rightarrow push(S_1, x) = push(S_2, x) \\ S_1 = S_2 \Rightarrow pop(S_1) = pop(S_2) \\ S_1 = S_2 \Rightarrow top(S_1) = top(S_2) \\ S_1 = S_2 \Rightarrow empty(S_1) = empty(S_2) \end{array}$$

We will show this proof for the push operation, and leave the remaining three as exercises. Note that all operators must be shown in this step — the substitutive property must hold for all operators, even those which map out of the type of interest, hence having proven a normal form lemma does not relieve us of work in this step of the proof.

Let $S_1 = STAK(A_1, i_1)$ and $S_2 = STAK(A_2, i_2)$, and assume that $S_1 = S_2$ to show that

$$push(STAK(A_1, i_1), x) = push(STAK(A_2, i_2), x)$$

Use the implementation of push on the right-hand side of the above expression to obtain

$$STAK(assign(A_1, i_1 + 1, x), i_1 + 1) = STAK(assign(A_2, i_2 + 1, x), i_2 + 1)$$

In terms of our equality interpretation, we must show

$$(i_1 = i_2) \ \wedge \ \forall k(1 \leq k \leq i_1, access(A_1, k) = access(A_2, k)) \ \Rightarrow$$
$$(i_1 + 1 = i_2 + 1) \ \wedge \forall k(1 \leq k \leq i_1 + 1,$$
$$access(assign(A_1, i_1 + 1, x), k) = access(assign(A_2, i_2 + 1, x), k)$$

The first clause of the consequent follows directly from the first clause of the antecedent. The second clause of the consequent is supported by simplifying as follows:

$$access(assign(A_1, i_1 + 1, x), k) = \ if \ k = i_1 + 1) \ then \ x \ else \ access(A_1, k)$$

due to the implementation program. Likewise,

$$access(assign(A_2, i_2 + 1, x), k) = \ if \ k = i_2 + 1) \ then \ x \ else \ access(A_2, k)$$

Hence, because $i_1 = i_2$, the clause we are examining simplifies to

$$\forall k(1 \leq k \leq i_1 + 1, \ if \ k = i_1 + 1 \ then \ x = x)$$
$$else \ (access(A_1, k) = access(A_2, k)))$$

For the range $i \leq k \leq i_1$, this is implied by clause two of the antecedent. When $k = i_1 + 1$, we know the expression is true due to the symmetry of "$=$" over integers.

Finally, we are at the point where we can verify that the axioms of our type STACK still hold through our implementation. The expressions which must be checked are:

1. pop(newstack) $=$ ***newstack***
2. pop(push(S, I)) $=$ ***S***
3. top(newstack) $=$ ***undefined***
4. top(push(S, I)) $=$ ***I***
5. empty(newstack) $=$ ***true***
6. empty(push(S, I)) $=$ ***false***

We will only prove two of these here, leaving the remaining as exercises.

First, we prove Item 4 above, showing

$$top(push(S, x)) = x$$

through the implementation. Using the representation invariant, assume that $S = STAK(A, i)$. Then

$$top(push(STAK(A, i), x)) =$$
$$top(STAK(assign(A, i + 1, x), i + 1)) =$$
$$access(assign(A, i + 1, x), i + 1) = x$$

Thus, the left side reduces to x and we have $x = x$.

Next we prove Item 2 above, showing

$$pop(push(S, x)) = S$$

Again, assume $S = STAK(A, i)$.

$$\begin{array}{l} pop(push(STAK(A, i), x)) = \\ pop(STAK(assign(A, i + \mathbf{1}, x), i + \mathbf{1})) = \\ if\ i + \mathbf{1} = \mathbf{0}\ then\ STAK(assign(A, i + \mathbf{1}, x), \mathbf{0})) \\ \quad else\ STAK(assign(A, i + \mathbf{1}, x), i) \end{array}$$

Now we must use data type induction to show that $P(STAK(A, i))$ implies $i \geq \mathbf{0}$ so that we need consider only one case above.

$$\begin{array}{l} P(newstack) = P(STAK(newarray, \mathbf{0})) = \mathbf{0} \geq \mathbf{0}. \\ P(S) \Rightarrow P(push(S, x)) \\ \quad P(STAK(A, i)) \Rightarrow P(push(STAK(A, i), x)) \\ \quad i \geq \mathbf{0} \Rightarrow P(STAK(assign(A, i + \mathbf{1}, x), i + \mathbf{1})) \\ \quad i \geq \mathbf{0} \Rightarrow i + \mathbf{1} \geq \mathbf{0} \end{array}$$

Because of our normal-form lemma, we do not need to show $P(S) \Rightarrow P(pop(S))$ separately.

Since $i \geq \mathbf{0}$, $i + \mathbf{1} \neq \mathbf{0}$, so we need only consider

$$STAK(assign(A, i + \mathbf{1}, x), i) = STAK(A, i)$$

Using the equality interpretation:

$$(i = i) \wedge \forall k(\mathbf{1} \leq k \leq i, access(assign(A, i + \mathbf{1}, x), k) = access(A, k))$$

Using equality of integers and the access implementation, this simplifies to

$$\forall k(\mathbf{1} \leq k \leq i, if\ i + \mathbf{1} = k\ then\ x\ else\ access(A, k) = access(A, k))$$

Since $i + \mathbf{1} \neq k, \forall k(\mathbf{1} \leq k \leq i, access(A, k) = access(A, k))$, and Item 2 is proved.

5.4. Verifying Applications With ADTs

As stated earlier in this chapter, it is important not only that we be able to verify implementations of abstract data types, but also that we be able to reason about programs which themselves use ADTs. Below is a simple use of our ADT "stack" to effect a "swap" between the variables a and b. Note that we do not need to initialize the variable s to be $newstack$. This program will perform as intended whether or not there happen to be other items already pushed on the stack when the program begins execution.

$$
\begin{array}{l}
\{true\} \\
\quad s := push(s, a) \\
\quad s := push(s, b) \\
\quad a := top(s) \\
\quad s := pop(s) \\
\quad b := top(s) \\
\{a = \overleftarrow{b} \wedge b = \overleftarrow{a}\}
\end{array}
$$

Let us now use Hoare-style inference techniques to verify the partial correctness of this program. Our inference system consists of both our existing first order predicate calculus, as enhanced to reason about assignment statements, plus the axioms for dealing with this new type "stack." As usual, we start with the postcondition and work back towards the precondition, using the axiom of assignment:

$$
\begin{array}{l}
\{a = \overleftarrow{b} \wedge top(s) = \overleftarrow{a}\}\ b := top(s)\ \ \{a = \overleftarrow{b} \wedge b = \overleftarrow{a}\} \\
\{a = \overleftarrow{b} \wedge top(pop(s)) = \overleftarrow{a}\} s := pop(s) \{a = \overleftarrow{b} \wedge top(s) = \overleftarrow{a}\} \\
\{top(s) = \overleftarrow{b} \wedge top(pop(s)) = \overleftarrow{a}\} a := top(s) \{a = \overleftarrow{b} \wedge top(pop(s)) = \overleftarrow{a}\} \\
\{top(push(s, b)) = \overleftarrow{b} \wedge top(pop(push(s, b))) = \overleftarrow{a}\} \\
\quad s := push(s, b) \{top(s) = \overleftarrow{b} \wedge top(pop(s)) = \overleftarrow{a}\} \\
\{top(push(push(s, a), b)) = \overleftarrow{b} \wedge top(pop(push(push(s, a), b))) = \overleftarrow{a}\} \\
\quad s := push(s, a) \\
\{top(push(s, b)) = \overleftarrow{b} \wedge top(pop(push(s, b))) = \overleftarrow{a}\}
\end{array}
$$

To identify that the above expression which we have derived is in fact equal to the program's precondition, simply apply the axioms for stack.

$$
\begin{array}{l}
\{1 \leq n \ \wedge \ \forall j \ni 1 \leq j \leq n, a[j] = \overleftarrow{a}[j]\} \\
\quad S := newstack; i := 1 \\
\quad \textbf{while } i \leq n \textbf{ do} \\
\qquad S := push(S, a[i]); i := i + 1 \\
\quad i := 1 \\
\quad \textbf{while } i \leq n \textbf{ do} \\
\qquad a[i] := top(S); S := pop(S); i := i + 1 \\
\{\forall j \ni 1 \leq j \leq n, a[j] = \overleftarrow{a}[n - j + 1]\}
\end{array}
$$

Figure 5.1. Program to reverse an array, using the ADT "stack."

5.5. Example: Reversing an Array Using a Stack

In Figure 5.1 is our old friend, the "array reversal program" implemented using our abstract data type STACK. We would probably never implement an array reversal in this way, but it is instructive to see the same sort of example in many different contexts. The method chosen is straightforward: All elements are pushed onto the stack in order, then, again in order, are popped off back into the array. A verification that the array is indeed reversed proceeds as follows: The precondition directly implies the program initialization section, as seen in Step 1, which also shows how our choice for the first loop invariant is supported by the initialization. The first loop maintains its invariant, as shown in Step 2. After the first loop, the invariant and $i > n$, followed by execution of $i := 1$, implies the expression which is our choice for the second loop's invariant. This is shown in Step 3. The invariant for the second loop is maintained, as shown in Step 4. Finally, the second invariant and $i > n$, after the second loop, clearly implies our postcondition, and the proof is complete.

In order to describe the remaining details, we accept that the first loop's invariant will be

$$J \triangleq S = \mathbf{PUSH}(i - 1) \ \wedge \ i \leq n + 1 \ \wedge \ \forall j \ni 1 \leq j \leq n, a[j] = \overleftarrow{a}[j]$$

and the second loop's invariant will be

$$I \triangleq \left\{ \begin{array}{l} \quad \forall j \ni 1 \leq j < i, a[j] = \overleftarrow{a}[n - j + 1] \\ \wedge\, S = \mathbf{PUSH}(n - i + 1) \ \wedge i \leq n + 1 \end{array} \right.$$

where we define the notation

$$\mathbf{PUSH}(j) := \begin{cases} \ (j > \mathbf{0} \rightarrow & push(\mathbf{PUSH}(j-\mathbf{1}), \overleftarrow{a}[j])) \\ \mid (j = \mathbf{0} \rightarrow & newstack) \end{cases}$$

Step One: Initialization

By use of our axiom of assignment we know

$$\underbrace{\{S = \mathbf{PUSH}(\mathbf{0}) \wedge \mathbf{1} \leq n + \mathbf{1} \wedge \forall j \ni \mathbf{1} \leq j \leq n, a[j] = \overleftarrow{a}[j]\}}_{Z} \ i := \mathbf{1} \ \{J\}$$

as well as

$$\{\mathbf{true} \wedge \mathbf{precondition}\} \ S := newstack \ \{Z\}$$

which can be composed for our initialization.

Step Two: First loop maintains invariant

By twice applying our axiom of assignment, we know

$$\underbrace{\{S = \mathbf{PUSH}(i) \ \wedge \ i \leq n \ \wedge \ \forall j \ni \mathbf{1} \leq j \leq n, a[j] = \overleftarrow{a}[j]\}}_{J'} \ i := i + \mathbf{1} \ \{J\}$$

and

$$\underbrace{\{push(S, a[i]) = \mathbf{PUSH}(i) \ \wedge \ i \leq n \ \wedge \ \forall j \ni \mathbf{1} \leq j \leq n, a[j] = \overleftarrow{a}[j] \ \}}_{J''}$$
$$S := push(S, a[i]) \ \{J'\}$$

After composing these, we observe that $J \wedge (i \leq n)$ implies clause two of J'' directly; it implies clause one by simple use of our notation; and it implies clause three directly. Hence, the invariant is maintained.

Step Three: Output of first loop implies input to second loop

The result of the first loop is

$$\begin{aligned} S &= \mathbf{PUSH}(i-\mathbf{1}) \wedge (i \le n+\mathbf{1}) \wedge \neg(i \le n) \\ &\Rightarrow S = \mathbf{PUSH}(n) \\ &\Rightarrow S = \mathbf{PUSH}(n) \wedge \mathbf{true} \\ &\Rightarrow S = \mathbf{PUSH}(n) \wedge \mathbf{1} = \mathbf{1} \end{aligned}$$

which can be used in a rule of consequence with the following application of the axiom of assignment

$$\{S = \mathbf{PUSH}(n) \wedge \mathbf{1} = \mathbf{1}\}\ i := \mathbf{1}\ \{S = \mathbf{PUSH}(n) \wedge i = \mathbf{1}\}$$

This expression implies the first clause of I vacuously; it implies the second clause by simple rearrangement of notation and repeated use of our algebraic rules for the ADT; and it implies the third clause of I directly, assuming we were clever enough to have pulled the requirement $n \ge 1$ down through the program so we could use it at this point. The third clause of J allows us to substitute $\overleftarrow{a}$ for our uses of a in the expression involving S. Hence, we have established I at the start of the second loop.

Step Four: Second loop maintains Invariant

Repeatedly using our axiom of assignment,

$$\underbrace{\begin{array}{c}\{\forall j, \mathbf{1} \le j < i+\mathbf{1}, a[j] = \overleftarrow{a}[n-j+\mathbf{1}] \\ \wedge\ S = \mathbf{PUSH}(n-(i+\mathbf{1})+\mathbf{1})\ \wedge\ i+\mathbf{1} \le n+\mathbf{1}\}\end{array}}_{I'} \quad i := i+\mathbf{1}\ \{I\}$$

$$\underbrace{\begin{array}{c}\{\forall j \ni \mathbf{1} \le j < i+\mathbf{1}, a[j] = \overleftarrow{a}[n-j+\mathbf{1}] \\ \wedge\ pop(S) = \mathbf{PUSH}(n-(i+\mathbf{1})+\mathbf{1})\ \wedge\ i \le n\}\end{array}}_{I''} \quad S := pop(S)\ \{I'\}$$

$$\underbrace{\begin{array}{c}\{\forall j \ni \mathbf{1} \le j < i+\mathbf{1}, \alpha(a, i, top(S))[j] = \overleftarrow{a}[n-j+\mathbf{1}] \\ \wedge\ pop(S) = \mathbf{PUSH}(n-(i+\mathbf{1})+\mathbf{1})\ \wedge\ i \le n\}\end{array}}_{I'''}$$

$$a[i] := top(S)\{I''\}$$

then composing these expressions we obtain:

$$\{I'''\}\quad \mathit{body\ of\ while}\quad \{I\}$$

We claim that $I \wedge (i \leq n) \Rightarrow I'''$ hence showing that the second **while** loop invariant is maintained: For j in the range $1 \leq j < i$, clause one of I''' follows directly from the first clause of I. For $j = i$, I''' reduces as follows:

$$I \Rightarrow \; top(S) \; = \; top(\mathbf{PUSH}(n - i + \mathbf{1}))$$
$$= top(push(\mathbf{PUSH}(n - i + \mathbf{1} - \mathbf{1}), \overleftarrow{a}[n - i + \mathbf{1}])) \; = \overleftarrow{a}[n - i + \mathbf{1}]$$

The second clause of I''' is supported as follows:

$$\begin{aligned} I \Rightarrow \; & S = \mathbf{PUSH}(n - i + \mathbf{1}) \; \Rightarrow \; pop(S) \\ & = \; pop(\mathbf{PUSH}(n - i + \mathbf{1})) \\ & = \; pop(push(\mathbf{PUSH}(n - i + \mathbf{1} - \mathbf{1}), \overleftarrow{a}[n - i + \mathbf{1}])) \\ & = \; \mathbf{PUSH}(n - i + \mathbf{1} - \mathbf{1}) \\ & = \; \mathbf{PUSH}(n - (i + \mathbf{1}) + \mathbf{1}) \end{aligned}$$

The third clause of I''' is directly supported by the loop conditional $i \leq n$.

6. INDUCTIONLESS INDUCTION

Previously we have used data type induction to prove assertions concerning our algebraically specified ADTs. There is a second technique which is useful, called *inductionless induction* (also called *structural induction*). It is useful in general, but we will use it here as an essential technique in showing that sets of axioms are consistent (as discussed earlier in this chapter).

The technique is based upon the technique known as unification and on the Knuth-Bendix algorithm for showing consistency among a set of axioms. We first describe unification and then give this algorithm.

6.1. Knuth–Bendix Algorithm

The idea behind this algorithm is fairly simple. The use of rewrite rules separates the set of valid expressions into discrete equivalence classes. If the rules are noetherian and confluent, then applying rewrite rules

for any equation causes the process to terminate with a unique normal-form value. Thus for two expressions, we can determine whether they are in the same class by computing the normal-form value for each.

Using a process called *unification*, described below, we can combine two rewrite rules into a new rule showing the equivalence of two new expressions. If these come from two distinct equivalence classes (computed by forming each's normal-form value), then we have shown that these two classes really are the same and we can collapse them together. We call these two expressions *critical pairs* and can add them as a new rewrite rule in our system.

As long as we add critical pairs, we collapse discrete sets together. However, if we end up by collapsing the equivalence classes described by *true* and *false*, then we have shown the inconsistency of our rewrite rules.

Unification

Unification is an algorithm that forms the basis of Logic programming languages, theorem provers, and type checking systems in functional languages. To unify two expressions U, V means to find substitutions for the variables occurring in U and V so that the two expressions become identical.

For example, unifying $f(X, Y)$ with $f(g(Y), Z)$ is done by binding: X to $g(Y)$, and Y to Z.

Unification can be seen as an algorithm for solving a system of equations. In setting up the algorithm to unify expressions $U = V$, a list, L, of unifications yet-to-be-done is set up with $(U = V)$ as its only member. Each variable, X, occurring in U and V, is set to point to an equation, $EQ(X)$, of the form:

$$Variable - List = Expression : Occurrences$$

where $Occurrences$ keeps track of the number of times any variable on the left-hand side occurs on the right-hand side of any other equation. This will keep track of cyclic references.

The Knuth–Bendix algorithm is basically a search technique, where each axiom (along with cleverly chosen transformations of that axiom)

is compared to all the remaining axioms in order to check whether an inconsistency has arisen. There are three possible outcomes:

1. The algorithm terminates (possibly after generating new rules none of which are inconsistent). In this case, the set of axioms is consistent. In Section 6.2 we use this fact to give another proof technique (inductionless induction) along with the already given data type induction.

2. An inconsistency is discovered. This usually presents itself by deriving the rule $true = false$ from the existing set of rules.

3. The algorithm does not terminate and an infinite set of rules is generated. In this case, the Knuth-Bendix algorithm cannot determine the consistency of the set of rules.

Huet's version [32] of the Knuth–Bendix algorithm is shown below. In order to use the algorithm, we must be able to orient the equations of the form $x = y$ into rewrite rules $x \rightarrow y$. Huet and Oppen describe a method for weighing words that permits equations to be ordered so that rewrite rules reduce the weights of equations.

Weighing words

The weight of a word w is defined as

$$weight(w) = weight_0 \times \sum_{j=1}^{V} occurs(v_j, w) + \sum_{j=1}^{O} weight_j \times occurs(f_j, w)$$

where $weight_0$ is the minimum weight of a nullary operator, v_j is a variable, $occurs(n, w)$ is the number of occurrences of symbol n in word w, and $weight_j$ is the weight of operator f_j. The symbols O and V represent the number of operators and number of variables respectively.

For example, consider natural numbers defined by the axioms:

$$\begin{array}{ll} add(\mathbf{0}, X) = & X \\ add(succ(X), Y) = & succ(add(X, Y)) \end{array}$$

and the equation describing the associativity of addition

add(add(X,Y),Z) = ***add(X,add(Y,Z))***

Generally in defining operators' weights, f is assigned a higher weight than g if f is used to define g. Also unary operators are usually assigned $weight_0$. The arithmetic operators have the following weights:

add	0
succ	0
0	1

Since 0 is the only nullary operator, $weight_0$ is 1. However, ***succ*** is not assigned a similar weight. Some sample words and their weights are shown below:

Word	Weight	
X	$1 \times (1) + ((0 \times 0) + (0 \times 0) + (1 \times 0)) =$	1
add(0,X)	$1 \times (1) + ((0 \times 1) + (0 \times 0) + (1 \times 1)) =$	2
add(***succ***(X),Y)	$1 \times (2) + ((0 \times 1) + (0 \times 1) + (1 \times 0)) =$	2
add(***succ***(X,Y))	$1 \times (2) + ((0 \times 1) + (0 \times 1) + (1 \times 0)) =$	2
add(***add***(X,Y),Z)	$1 \times (1+1+1) + ((0 \times 2) + (0 \times 0) + (1 \times 0)) =$	3
add(X,***add***(Y,Z))	$1 \times (1+1+1) + ((0 \times 2) + (0 \times 0) + (1 \times 0)) =$	3

Ordering equations

The relation used for ordering words $a > b$ is defined as follows: $a > b$ iff either

1. $weight(a) > weight(b)$ and $1 \leq i \leq V \;\Rightarrow\; occurs(v_i, a) \geq occurs(v_i, b)$; or

2. $weight(a) = weight(b)$ and $1 \leq i \leq V \;\Rightarrow\; occurs(v_i, a) = occurs(v_i, b)$, and either

(a) $a = f(v_k)$ and $b = v_k$ for some operator f, or

(b) $a = f_j(a_1, \ldots, a_j)$ and $b = f_k(b_1, \ldots, b_k)$ and either $j > k$, or both $j = k$ and $1 \leq t \leq j \;\Rightarrow\; a_1 = b_1, \ldots, a_{t-1} = b_{t-1}, a_t > b_t$.

Several examples appear below:

Words	Reasons
add(0,X) > X	2>1
add(***succ***(X),Y) > ***succ***(***add***(X,Y))	2=2, each has 2 variables, first has fewer args
add(***add***(X,Y),Z) > ***add***(X,***add***(Y,Z))	3=3, each has 3 variables, ***add***(a,b) > a
add(***succ***(X),X) ? ***add***(0,X)	2=2, but the number of variables differs

In the algorithm, the following operations use the notion of order:

Operation	Meaning
orderable($s = t$)	($s > t$) or ($t > s$)
order($s = t$)	$(s > t \rightarrow (s = t)) \mid (t = s)$

Here, ***orderable*** is a Boolean-valued predicate that checks whether the two words *can* be compared according to the above criteria. ***Order*** is a function which then would return the rewrite rule based on the original equation.

Critical pairs

Critical pairs arise when (with renaming of variables) the left side of one rewrite rule occurs as an argument in the left side of another rule. This is an application of the unification property described earlier. Thus equations that could be reduced by the second rule could also be reduced by the first. After applying unification, if the normal forms for each expression in the rewrite rule are different, then we have a critical pair. The critical pair produced is an equation asserting the equality of the two expressions.

Some notation must first be developed in order to discuss critical pairs further:

Notation	Meaning
t/u	The subterm u of t
t_y^u	Replace subterm u of t with y
σ	A substitution $\{u := y\}$ applied to a term
$normalize(x)$	Compute normal form for expression x

Superposition algorithm

Assume rewrite rules $u \rightarrow v$ and $x \rightarrow y$ with the property that u and x have no variables with common names. If z is a nonvariable occurrence in u such that u/z and x are unifiable with minimal unifier σ, then the pair of terms $\langle \sigma(u_y^z), \sigma(v) \rangle$ is potentially a critical pair. If $normalize(\sigma(u_y^z)) \neq normalize(\sigma(v))$, then they are a critical pair, and they represent an additional rule (or theorem) derivable from our axioms: $normalize(\sigma(u_y^z)) = normalize(\sigma(v))$. (Actually, we need the terms in the correct order, or $order(normalize(\sigma(u_y^z))) = order(normalize(\sigma(v)))$.)

For example, the rules:

$$add(\mathbf{0}, W) \rightarrow W$$
$$add(add(X, Y), Z) \rightarrow add(X, add(Y, Z))$$

with unifier:

$$\sigma(X := \mathbf{0}, Y := W)$$

produce the critical pair:

$$\langle add(W, Z), add(\mathbf{0}, add(W, Z)) \rangle$$

Using this notation, the Knuth-Bendix algorithm is given in Figure 5.2. An example of the use of this algorithm is given in the next section.

6.2. Application of Knuth Bendix to induction

The idea of inductionless induction was introduced by Musser [46] [47] when he proved the following theorem:

Theorem: Let T be a collection of types with axiom set A and assume that T is fully specified by (i.e., sufficiently complete with respect to) A. If E is a set of equations, each in the language generated by T, and A ∪ E is consistent, then each equation in E is in the inductive theory of T.

Thus to prove an equation $x = y$ by inductionless induction, add it to a set of equations containing a sufficiently complete set of axioms and execute the Knuth–Bendix algorithm. If the algorithm terminates,

```
R := {};
i := 0;
done := false;
while ¬done do begin
    done := true;
    while E ≠ {} do begin        "Order E's equations and put them in R."
        "Find non − joinable critical pair"
        (s = t) := choseOneFrom(E);
        E := E − {s = t};
        s' := normalize(s, R); t' := normalize(t, R)
        if s' ≠ t' then begin "Order equation."
            if ¬orderable(s' = t') then return(fail);
            (v → w) := order(s' = t');
            "Normalize rewriting system."
            for each ⟨x → y, positionXY, markXY⟩in R do begin
                x' := normalize(x, {v → w});
                if x ≠ x' then begin
                    R := R − {⟨x → y, positionXY, markXY⟩};
                    E := E ∪ {x' = y}
                    end
                else begin
                    y' := normalize(y, R ∪ {v → w});
                    if y ≠ y'then
                        R := (R − {⟨x → y, positionXY, markXY⟩})
                        ∪{⟨x → y', positionXY, markXY⟩}
                    end
                end; "... for each"
            i := i + 1;
            R := R ∪ {⟨v → w, i, unmarked⟩}
            end"... order equation."
    end; "... while E ≠ {}"
    "Find an unmarked rule."
    for each⟨x → y, positionXY, markXY⟩in R do
        if markXY = unmarked then begin
            "Compute critical pairs."
            done := false;
            for each⟨u → v, positionUV, markUV⟩in R do
                if positionUV ≤ positionXY then
                    E := E ∪ criticalpairs(x → y, u → v);
            R := (R − {⟨x → y, positionXY, markXY⟩})∪
                {⟨x → y, positionXY, marked⟩}
            end;
    end; "... while not done."
return(success).
```

Figure 5.2. Knuth–Bendix Algorithm

then $x = y$ is consistent with the previous axioms; if it derives $true = false$, then the new rule is inconsistent with the set of axioms.

As an example (adapted from [17]) of both inductionless induction and the use of Knuth–Bendix, we now use this method to show how the axiom of associativity of addition is consistent with the algebraic axioms of addition given earlier. Initially, the set of equations E contains the following members:

$$
\begin{array}{l}
add(\mathbf{0}, U) = U \\
add(succ(V), W) = succ(add(V, W)) \\
add(add(X, Y), Z) = add(X, add(Y, Z))
\end{array}
$$

The **while** statement guarded by $E \neq \{\}$ removes these equations from E one at a time, orients them according to the weighing scheme described above, normalizes them with respect to each other, and adds them to R (the list of rewrite rules):

$$
\begin{array}{l}
\langle add(\mathbf{0}, U) \rightarrow U, \mathbf{1}, unmarked\rangle \\
\langle add(succ(V), W) \rightarrow succ(add(V, W)), \mathbf{2}, unmarked\rangle \\
\langle add(add(X, Y), Z) \rightarrow add(X, add(Y, Z)), \mathbf{3}, unmarked\rangle
\end{array}
$$

After E is empty, the **for** statement selects unmarked rules from R, computes critical pairs for lower numbered members of R, adds the critical pairs to E, and replaces the unmarked rules with marked rules. For our example, no critical pairs are computed until the third element of R is selected.

$$
\begin{array}{l}
\langle add(\mathbf{0}, U) \rightarrow U, \mathbf{1}, marked\rangle \\
\langle add(succ(V), W) \rightarrow succ(add(V, W)), \mathbf{2}, marked\rangle \\
\langle add(add(X, Y), Z) \rightarrow add(X, add(Y, Z)), \mathbf{3}, unmarked\rangle
\end{array}
$$

The following critical pairs are computed and added to E:

Unifier	**Equation**
$\{X := \mathbf{0}, Y := U\}$	$add(U, Z) = add(\mathbf{0}, add(U, Z))$
$\{X := succ(V), Y := W\}$	$add(succ(add(V, W)), Z) = add(succ(V), add(W, Z))$

The final element of R is also marked so no unmarked members of R remain. Since *done* has value *false* the outer *while* statement is executed again with E no longer empty. When one of the equations of E is selected and normalized with respect to R, E is empty and no new rewrite rules are added to R. Thus the algorithm terminates.

$add(U, Z) = add(\mathbf{0}, add(U, Z))$:

$$normalize(add(U, Z)) = add(U, Z)$$
$$normalize(add(\mathbf{0}, add(U, Z))) = add(U, Z) \;\; (by\ 1)$$

$add(succ(add(V, W)), Z) = add(succ(V), add(W, Z))$:

$$\begin{aligned}
&normalize(add(succ(add(V, W)), Z)) = \\
&\quad succ(add(add(V, W), Z)) = \;\; (by\ 2) \\
&\quad succ(add(V, add(W, Z))) = \;\; (by\ 3) \\
&normalize(add(succ(V), add(W, Z)) = \\
&\quad succ(add(V, add(W, Z))) \;\; (by\ 2)
\end{aligned}$$

Notice that equations like the commutativity of addition "***add***(X,Y) = ***add***(Y,X)" cannot be proved using this technique because the terms cannot be ordered by weight. However, such equations can be proved by data type induction. (This is given as an exercise.)

6.3. Example Using Knuth–Bendix

Consider the following specification for sequences of integers. The symbols S, S', T, and T' will represent sequences, and x and x' will represent integers. We will define the specifications for five functions:

1. $null$ returns the null sequence.
2. $S + x$ appends integer x onto the end of sequence S.
3. $x|S$ places x at the start of sequence S.
4. $S\ cat\ T$ concatenates two sequences S and T into one sequence.
5. $isnull(S)$ returns $true$ if sequence S is null; $false$ otherwise.

The signatures for these functions are:

$null$	$\rightarrow sequence$
$+$	$sequence \times integer \rightarrow sequence$
$\|$	$integer \times sequence \rightarrow sequence$
cat	$sequence \times sequence \rightarrow sequence$
$isnull$	$sequence \rightarrow boolean$

The ten axioms needed to define these sequences are:

$$
\begin{array}{ll}
Ax_1 & (S = S) = true \\
Ax_2 & (null = S + x) = false \\
Ax_3 & (S + x = null) = false \\
Ax_4 & (S + x = S' + x') = ((S = S') \wedge (x = x')) \\
Ax_5 & x|null = null + x \\
Ax_6 & x|(S + x') = (x|S) + x' \\
Ax_7 & null\ cat\ S = S \\
Ax_8 & (S + x')\ cat\ S' = S\ cat\ (x'|S') \\
Ax_9 & isnull(null) = true \\
Ax_{10} & isnull(S + x) = false
\end{array}
$$

Example: Data type induction

Let us review the previous method of data type induction by proving that

$$S\ cat\ (S' + x') = (S\ cat\ S') + x'$$

.

Let $P(S)$ be the predicate: $\forall S'(S\ cat\ (S' + x') = (S\ cat\ S') + x')$. We will assume a normal form lemma, such as with stacks, that states that the constructors for $sequences$ are $null$ and $+$. Both cat and $|$ can be obtained by applications of the first two functions. To use data type induction, we need to show that:

1. $P(S)$, and

2. $P(S) \Rightarrow P(S + x)$

(1) $P(null) = \forall S'(null\ cat\ (S' + x') = (null\ cat\ S') + x')$

Proof:

$$
\begin{array}{ll}
null\ cat\ (S' + x') \ = \ (null\ cat\ S') + x' & Hypothesis \\
(S' + x') \ = \ (null\ cat\ S') + x' & Ax_7 \\
(S' + x') \ = \ (S' + x') & Ax_7 \\
true & Ax_1 \\
\square &
\end{array}
$$

(2) Show $P(S) \Rightarrow P(S + x)$.

Proof:

$$
\begin{array}{ll}
Assume\ P(S) \ = \ S\ cat\ (S' + x') \ = & \\
\quad (S\ cat\ S') + x' & \\
Show\ P(S + x) \ = \ (S + x)\ cat\ (S' + x') \ = & \\
\quad ((S + x)\ cat\ S') + x' & \\
(S + x)\ cat\ (S' + x') \ = \ ((S + x)\ cat\ S') + x' & \\
S\ cat\ (x|(S' + x')) \ = \ ((S + x)\ cat\ S') + x' & Ax_8 \\
S\ cat\ ((x|S') + x') \ = \ ((S + x)\ cat\ S') + x' & Ax_6 \\
(S\ cat\ (x|S')) + x' \ = \ ((S + x)\ cat\ S') + x' & Inductive\ Hypothesis \\
(S\ cat\ (x|S')) + x' \ = \ (S\ cat\ (x|S')) + x' & Ax_8 \\
true & Ax_1 \\
\square &
\end{array}
$$

Thus we have shown $P(S)$ for all $S \in sequences$.

Using Knuth–Bendix

There are four conditions that can result from applying the Knuth–Bendix algorithm to a set of axioms. In one case, given axiom $\alpha = \beta$, we are unable to define a weight function w such that $w(\alpha) > w(\beta)$. In this case, the algorithm does not apply and we cannot use it to determine the consistency of the axioms.

In the case where we can order the axioms, the algorithm has three results:

1. The algorithm cannot develop any more critical pairs. In this case, we have shown that the axioms are consistent. If an original set of axioms is complete and we add a new axiom, then this new axiom must be consistent with the complete set of axioms, and hence must be a theorem provable from the original axioms.

2. The algorithm derives the critical pair $true = false$. This is a termination condition which shows that the original axioms are inconsistent.

3. The algorithm generates a nonterminating set of critical pairs. In this case, the algorithm is inconclusive as to the correctness of the set of axioms.

The following examples all demonstrate these results.

Assume the definition of $sequences$ given above with its ten axioms. Add to these axioms the following:

1. $S\ cat\ (S' + x') \ = \ (S\ cat\ S') + x'$. This rule causes the Knuth–Bendix algorithm to terminate. As was already shown above, this statement is consistent with the given axioms.

2. $S\ cat\ null \ = \ S$ and Rule 1 above. These two are consistent and the algorithm terminates.

3. $S\ cat\ null \ = \ S$ without Rule 1. In this case, as will be shown below, the algorithm does not terminate. Without the above "associative" rule, this second rule cannot be proven within the framework of the given ten axioms, even though we know it is true by first proving the companion associative rule.

4. $S\ cat\ null \ = \ null$. Intuitively, this should be $false$. As will be shown, the algorithm demonstrates that this rule is inconsistent with the axioms.

Nontermination example

Consider the ten axioms and the new rule $S\ cat\ null \ = \ S$ (from [47]). Applying Knuth–Bendix gives the following steps:

$T\ cat\ null \ = \ T$	$Substitute\ for\ new\ rule$
$(S + x')\ cat\ S' \ = \ S\ cat\ (x' \vert S')$	Ax_8
$\langle S\ cat\ (x' \vert null), S + x' \rangle$	$Possible\ Critical\ Pair$ $\sigma(T := S + x', S' := null)$
$\langle S\ cat\ (null + x), S + x' \rangle$	$Normalize\ expressions\ with\ Ax_5$
$S\ cat\ (null + x) \ = \ S + x$	$Critical\ Pair -$ $New\ rule\ added$

At this point, we can repeat the process. Take the final axiom (critical pair 2, above) and repeat the same five steps using for σ in critical pair 1 the sequence: $S := null, S := (null + x), S := ((null + x) + x', \ldots$. This gives the infinite sequence of critical pairs:

$$S\ cat\ (null + x') \ = \ S + x'$$
$$S\ cat\ ((null + x) + x') \ = \ (S + x) + x'$$
$$S\ cat\ (((null + x) + x') + x'' \ = \ ((S + x) + x') + x''$$
$$\vdots$$

and the algorithm does not terminate.

Inconsistent axiom example

Consider the rule $S\ cat\ null \ = \ null$. We will show it is inconsistent as follows:

$T\ cat\ null \ = \ null$	$Substitute\ for\ new\ rule$
$(S + x')\ cat\ S' \ = \ S\ cat\ (x' \vert S')$	Ax_8
$\langle S\ cat\ (x' \vert null), null\rangle$	$Possible\ CriticalPair-$ $\sigma(T := S + x', S' := null)$
$S\ cat\ (null + x) \ = \ null$	$Critical\ Pair\ 1-$ $Normalize\ with\ Ax_5$
$null\ cat\ T \ = \ T$	$Substitute\ for\ Ax_7$
$\langle null + x, null\rangle$	$Possible\ Critical\ Pair-$ $\sigma(T := null + x, S := null)$
$null + x \ = \ null$	$Critical\ Pair\ 2$
$isnull(S + x) \ = \ false$	Ax_{10}
$isnull(null) \ = \ false$	$\sigma(S := null)$
$isnull(null) \ = \ true$	Ax_9
$true \ = \ false$	$Critical\ Pair$

Since we have added the rule $true \ = \ false$, the initial rule is inconsistent with our axioms.

7. EXERCISES

Problem 1. Redo the algebraic specification of a stack in Section 3 so that the definition is modified to replace the top element of the stack with the new element when the stack is full, rather than ignore the new element, as is presently given.

Problem 2. Consider the set of natural numbers with operations: 0, succ, and add.

add(X,0)	=	X
add(X,succ(Y))	=	succ(add(X,Y))

• Give an appropriate, complete and non-redundant definition of the predicate even(X).

• Prove the following theorem: even(X) $\wedge$ even(Y) $\Rightarrow$ even(add(X,Y))

Problem 3. Consider the type "set of integers" defined by the following axioms.

in(emptySet,X)	=	false
in(insert(S,I),J)	=	I=J $\vee$ in(S,J)
delete(emptySet,X)	=	emptySet
delete(insert(S,I),J)	=	if I=J then delete(S,J) else insert(delete(S,J),I)
isEmptySet(emptySet)	=	true
isEmptySet(insert(S,X))	=	false

• Give a correct, complete, and non-redundant list of axioms for set equality.

• Prove the theorem in(delete(S,X),X) = false.

Problem 4. Consider a data type list with operations:

append(null,X)	=	X
append(cons(X,Y),Z)	=	cons(X,append(Y,Z))

• Add axioms for the operation rev, which reverses the order of the elements in the list.

• Show rev(rev(X)) = X.

Problem 5. Consider the following definition of type list in which the operation add appends values to the end of a list.

```
front(newlist)            =   0
front(add(newlist,A))     =   A
front(add(add(L,A),B))    =   front(add(L,A))

tail(newlist)             =   newlist
tail(add(newlist,A))      =   newlist
tail(add(add(L,A),B))     =   add(tail(add(L,A)),B)

length(newlist)           =   0
length(add(L,A))          =   succ(length(L))
```

• Give axioms to define the operations sorted and perm having the following intuitive definitions. You may define hidden functions if necessary.

– sorted: list $\rightarrow$ boolean. sorted returns true if the list elements appear in ascending order with the smallest value at the front. Otherwise sorted returns false.

– perm: list $\times$ list $\rightarrow$ boolean. perm returns true if one of its arguments is a permutation of the other. Otherwise perm returns false.

Problem 6. Consider the operations member and sinsert, defined as follows:

```
member(newlist,X)      =   false
member(add(L,X),Y)     =   ((X=Y) or member(L,Y))
sinsert(newlist,A)     =   add(newlist,A)
sinsert(add(L,A),B)    =   if B >= A then add(add(L,A),B)
                                 else add(sinsert(L,B),A))
```

Prove the theorem: member(sinsert(L,X),X) = true. You may assume the existence of a normal form lemma showing that all list objects can be constructed from the operations newlist and add.

Problem 7. Give algebraic specifications for a library of books (whose titles are natural numbers) with the following operations:

init:	$\rightarrow$ library	open for business
donate:	library $\times$ nat $\rightarrow$ library	add a volume to library
remove:	library $\times$ nat $\rightarrow$ library	destroy a book & forget it was donated
check:	library $\times$ nat $\rightarrow$ library	borrow a book
return:	library $\times$ nat $\rightarrow$ library	bring a book back
given:	library $\times$ nat $\rightarrow$ bool	has a book been donated?
avail:	library $\times$ nat $\rightarrow$ bool	is book available for checkout?
out:	library $\times$ nat $\rightarrow$ bool	is book already checked out?

Using data type induction, show: given(L,B) $\supset$ (avail(L,B) $\vee$ out(L,B)).

Problem 8. The low-water-mark problem is defined for a system composed of processes and objects, each with its own security level. Three security levels are linearly ordered: classified $\leq$ secret $\leq$ topsecret. Processes have fixed security levels, but objects' security levels may be changed by operations. When a process writes a value in an object, the security level of the object drops to that of the process. When a process resets an object, the security level of the object becomes top secret and its value becomes undefined. The low-water-mark idea is that the security level of an object may decrease, but not increase unless a reset operation is executed for the object.

The Bell and LaPadula simple security and * properties (colloquially known as "no read up, no write down") require enforcement of the following restrictions:

- For a process to read an object's value, the security level of the process must be greater than or equal to that of the object.

- For a process to write an object's value, the security level of the process must be less than or equal to that of the object.

Using the following operations, give an algebraic specification for the data type state which enforces the Bell and LaPadula properties. Operations attempting to "read up" should return the undefined value. Operations attempting to "write down" should be ignored. Reset operations enforce the same security-level restrictions as write operations.

newstate:	$\rightarrow$ state
reset:	process $\times$ object $\times$ state $\rightarrow$ state
write:	process $\times$ object $\times$ nat $\times$ state $\rightarrow$ state
read:	process $\times$ object $\times$ state $\rightarrow$ nat $\cup$ {undefi ned}
name:	object $\rightarrow$ id
level:	object $\times$ state $\rightarrow$ (classified, secret, topsecret)
level:	process $\rightarrow$ (classified, secret, topsecret)

Problem 9. Consider the list data type with operations:

$$\begin{array}{ll} newlist: & \rightarrow list \\ addelt: & list \times nat \rightarrow list \\ tail: & list \rightarrow list \\ head: & list \rightarrow nat \\ last: & list \rightarrow nat \\ length: & list \rightarrow nat \end{array}$$

with their usual meanings. Give three different sets of axioms defining lists in which elements are added to the right end of the list, in which elements are added to the left end of the list, and in which elements are added so that the list remains sorted in ascending order. To make things a bit easier, you may define operations so that applying them to empty lists yields 0.

Problem 10. If we added the operation join to lists in which elements are added to the right end of the list, we might write the axioms:

$$\begin{array}{ll} Signature: & \\ join: & list \times list \rightarrow list \\ Axioms: & \\ join_1: & join(L, newlist) = L \\ join_2: & join(L, addelt(M, X)) = addelt(join(L, M), X) \end{array}$$

Show that $join(newlist, L) = L$ is not an axiom, but a theorem following from the given axioms.

Problem 11. Consider the following implementation of the data type SetType with operations NewSet, AddElt, and Member. Give a repre-

sentation function for the type and write comments for each operation giving pre, post, in and out.

```
const
   SetMax = 256;
type
   EltType =  integer;
   SetIndex = 1..SetMax;
   SetType =  record
                 Next: 0..SetMax;
                 V: array [SetIndex] of EltType
              end;

procedure NewSet(var S: SetType);
begin
   S.Next := 0
end;

procedure AddElt(var S: SetType; E: EltType);
var i: integer;
begin
   S.V[S.Next+1] := E;
   i := 1;
   while (i<=S.Next+1) and (S.V[i]<>E) do i := i+1;
   if i = S.Next+1 then S.Next := S.Next+1
end;

function Member(var S: SetType; E: EltType): boolean;
var I: SetIndex;
    B: boolean;
begin
   B := false;
   for I := 1 to S.Next do B := B or (E = S.V[I]);
   Member := B
end;
```

Problem 12. A bag or multiset is a set that permits duplicate values. Consider the following implementation of the data type "bag of integers." A set cannot be used to represent a bag, as there is no way to indicate the presence in a set of more than one occurrence of an element. Two good choices would be to represent a bag as a sequence of

elements, since a sequence can contain duplicated elements, or a set of pairs representing (element, number of occurrences). Which choice makes the specification task easier?

```
const Max = ...;
type T = integer;
     Bag = array [0..Max] of T;
procedure BagInit(var B: Bag);
   var i: integer;
   begin
      for i := 0 to Max do B[i] := 0
   end;
procedure BagAdd(var B: Bag; X: T);
   begin
      if (0 <= X) and (X <= Max) then B[X] := B[X] + 1
   end;
procedure BagDel(var B: Bag; X: T);
   begin
      if (0 <= X) and (X <= Max) and (B[X] > 0)
         then B[X] := B[X] - 1
   end;
function BagMember(var B: Bag; X: T): boolean;
   begin
      if (0<=X) and (X<=Max) then BagMember := B[X]>0
      else BagMember := false
   end;
```

Problem 13. Give a representation function for the type and abstract and concrete comments for data type IntSet.

```
type IntSet = array [1..Max] of boolean;

procedure IntSetInit(var S: IntSet);
var i: integer;
begin
   for i := 1 to Max do S[i] := false
end;

procedure IntSetAdd(var S: IntSet; X: integer);
begin
   if (1<=X) and (X<=Max) then S[X] := true
```

```
end
```

Problem 14. The following code implements type "queue of EltType" as a circular list with a contiguous representation.

```
const Max = 2;
type
   EltType = 0..maxint;
   QueueIndex = 0..Max;
   Queue = record
             M: array [QueueIndex] of EltType;
             H, T: QueueIndex
           end;

procedure NewQ(var Q: Queue);
begin
   Q.H := 0; Q.T := 0;
end;

procedure EnQ(var Q: Queue; Elt: EltType);
var Temp: QueueIndex;
begin
   Temp := (Q.T + 1) mod (Max + 1);
   if Temp <> Q.H then begin
      Q.T := Temp;
      Q.M[Q.T] := Elt
      end;
end;

procedure DeQ(var Q: Queue; var Result: EltType);
begin
   if Q.H <> Q.T then begin
      Q.H := (Q.H + 1) mod (Max + 1);
      Result := Q.M[Q.H];
      end
   else Result := 0
end;
```

State a representation function for the type. State non-trivial concrete and abstract invariants for the type and show that the concrete

invariant implies the abstract invariant. Give pre, post, in, and out comments for NewQ and demonstrate their consistency.

8. SUGGESTED READINGS

The algebraic specification techniques are well-described in the papers:

• J. V. Guttag and J. J. Horning, "The algebraic specification of abstract data types," *Acta Informatica*, Vol. 10, 1978, pp. 27-52.

• J. V. Guttag, "Notes on Type Abstraction (Version 2)," *IEEE Transactions on Software Engineering*, Vol. 6, No. 1, 1980, pp. 13-23.

• J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract Data Types and Software Validation ," *Communications of the ACM*, Vol. 21, No. 12, 1978, pp. 1048-1064.

• J. V. Guttag, E. Horowitz, and D.R. Musser, "The Design of Data Type Specifications ," R.T. Yeh (Ed.), *Current Trends in Programming Methodology 4: Data Structuring*, Prentice-Hall, Englewood Cliffs NJ, 1978, pp. 60-79.

A paper which summarizes the verification process for algebraically specified data types is the following. This paper presents a straightforward technique for expressing algebraic axioms as a pair of pre- and postconditions suitable for Hoare-style proofs.

• L. Flon and J. Misra, "A Unified Approach to the Specification and Verification of Abstract Data Types," *Proceedings of the IEEE Conference on Specifications of Reliable Software*, Cambridge, MA, 1979, pp. 162-169.

Papers which describe proof techniques for algebraic data types include the following:

• D. R. Musser, "Abstract data type specifications in the AFFIRM system," *IEEE Specifications of Reliable Software Conference*, Cambridge MA, 1979, pp. 47-57.

• D. R. Musser, “On Proving Inductive Properties of Abstract Data Types,” *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, 1980, pp. 154-162.

• G. Huet, “Confluent Reductions: Abstract Properties and Applications to Term-Rewriting Systems,” *Journal of the ACM*, Vol. 27, 1980, pp. 797-821.

• G. Huet and J.-M. Hullot, “Proofs by Induction in Equational Theories with Constructors ,” *Journal of Computer and System Science*, Vol. 25, 1982, pp. 239-266.

The first two Musser papers describe the application of the Knuth–Bendix algorithm to the problem of proving properties of ADTs – in this case by describing the AFFIRM system, which Musser built.

Chapter 6

Denotational Semantics

Denotational semantics is a form of an operational semantic model of a program's execution based upon function composition. The basic model is an outgrowth of simple type theory and the λcalculus. First the λcalculus will be described, and then denotational semantics will be explained as an extension.

Aside from its role in explaining semantic content of a program, the λcalculus is the basis for the VDM specification methodology, which has been used to specify several very large real systems.

1. THE LAMBDA CALCULUS

The λcalculus is a formal functional model used by Church to develop a theory of numbers. It is relevant to programming language design since it represents a typeless (hence simpler) model in the theory of denotational semantics.

λexpressions are defined recursively as:

1. If x is a variable name, then x is a λexpression.

2. If M is a λ-expression, then $\lambda x M$ is a λexpression.

3. If F and A are λexpressions, then (FA) is a λexpression. F is the *operator* and A is the *operand*.

The following are all λexpressions:

$$\begin{array}{lll} x & \lambda xx & \lambda xy \\ \lambda x(xy) & (\lambda x(xx)\lambda x(xx)) & \lambda x\lambda yx \end{array}$$

Variables may be bound or free. In λxM, x is the *binding* variable and occurrences of x in M are *bound*. A variable is *free* if it is not bound.

Any bound variable may have its name changed. Thus the λexpression λxx is equivalent to λyy. $\lambda x\lambda xx$ is equivalent to $\lambda x\lambda yy$ since the variable x is not free in the original λexpression λxx.

Informally, bound variables are "parameters" to the function described by the λexpression, free variables are global. There is no concept of local variable.

This analogy shows that the λexpression is a simple approximation to the procedure or subroutine concept in most algorithmic programming languages like Pascal, C, Ada or Fortran. λexpressions are almost directly representable in LISP and λsubstitutions are a direct model of Algol-like procedures.

λexpressions have only one operation – *reduction*. If (FA) is a λexpression, and $F = \lambda xM$, then A may be substituted for all free occurrences of x in M. This is written as: $(\lambda xMA) \rightarrow F'$.

Examples:

$$\begin{array}{llll} (\lambda xxy) & \rightarrow y & & \\ (\lambda x(xy)y) & \rightarrow (yy) & & \\ (\lambda x(xy)\,\lambda xx) & \rightarrow (\lambda xxy) & \rightarrow y & \\ (\lambda x(xx)\,\lambda x(xx)) & \rightarrow (\lambda x(xx)\,\lambda x(xx)) & \rightarrow & \ldots \end{array}$$

Note that the above examples are all unambiguous. The first expression (λxxy) has only one interpretation. It must be of the form $(F\ A)$ or $(\lambda xM\ A)$. So A is either y or xy. But xy is not a proper λexpression, so A is just y.

Also note that the third expression does not terminate. If we have $F = (\lambda x M\ A)$, substitution of A for x in M results in F again. This is a nonterminating reduction. This leads us to:

Church-Rosser property. If two different reductions of a λexpression terminate, then they are members of the same value class.

The λcalculus was originally developed as a logical model of the computational process. We can use such expressions to model our understanding of arithmetic. First λexpressions will be used to model the predicate calculus, and then using predicate calculus, we can model integers.

1.1. Boolean Values as λExpressions

Objects will be modeled by functions.

True T will be defined as: $\lambda x \lambda y x$. (Of a pair, choose the first.)

False F will be defined as: $\lambda x \lambda y y$. (Of a pair, choose the second.)

We have defined these objects so the following properties are true:

$((T\ P)Q) \rightarrow P$

Proof:

$$((T\ P)Q) \rightarrow ((\lambda x \lambda y x\ P)Q) \rightarrow (\lambda y P\ Q) \rightarrow P$$

□

$((F\ P)Q) \rightarrow Q$

Proof:

$$((F\ P)Q) \rightarrow ((\lambda x \lambda y y\ P)Q) \rightarrow (\lambda y y\ Q) \rightarrow Q$$

□

Given these constants T and F, we can define the boolean function:

$$not \triangleq \lambda x((xF)T)$$

Given these definitions, we need to show that our interpretation of them is consistent with our axioms and rules of predicate logic. In this case, we need to show that not, when applied to T returns F, and not when applied to F returns T.

$$(not\ T) = (\lambda x((x\ F)T)T) \rightarrow ((T\ F)T) \rightarrow F$$
$$(not\ F) = (\lambda x((x\ F)T)F) \rightarrow ((F\ F)T) \rightarrow T$$

Note that this is the desired behavior. Nothing is said if the argument to not is neither T nor F.

Similarly, we can define and and or as:

$$and \triangleq \quad \lambda x \lambda y((xy)F)$$
$$or \triangleq \quad \lambda x \lambda y((xT)y)$$

1.2. Integers

From these logical functions, we can now develop the integers:

$$\mathbf{0} \triangleq \quad \lambda f \lambda cc$$
$$\mathbf{1} \triangleq \quad \lambda f \lambda c(fc)$$
$$\mathbf{2} \triangleq \quad \lambda f \lambda c(f(fc))$$
$$\mathbf{3} \triangleq \quad \lambda f \lambda c(f(f(fc)))$$

c plays the role of the "zero" element, and f is the "successor" function (addition of $+1$).

From these definitions of the integers, we can extend them to include our usual arithmetic operations.

Let N be the $N\underline{th}$ integer. The λexpression $(N\ a)$ is $\lambda c(a \ldots (a\ c) \ldots)$. Computing $((N\ a)b)$, we get $(a \ldots (a\ b) \ldots)$.

Consider $((M\ a)((N\ a)b))$ by applying constant $((N\ a)b)$ to λexpression $(M\ a)$. The substitution of $((N\ a)b)$ for c in $(M\ a)$ yields $(a \ldots (a\ b) \ldots)$ where there are now $(M + N)$ as in the list. We have just developed addition. Therefore:

Addition:

$$[M+N] \triangleq \lambda a \lambda b((M\ a)((N\ a)b))$$
$$+ \triangleq \lambda M \lambda N \lambda a \lambda b((M\ a)((N\ a)b))$$

Similarly,

Multiplication:

$$[M \times N] \triangleq \lambda a(M(N\ a))$$

Exponentiation:

$$[M^N] \triangleq (N\ M)$$

While this development can continue until we have developed all of the recursive functions of mathematics, our primary goal is to consider the semantics of programming languages. So we will leave the λcalculus at this point and start to look at datatypes.

2. Datatypes

λexpressions either reduce to constants ($value$) or to other λexpressions. Thus all λexpressions are solutions to the functional equation:

$$value = constant + (value \rightarrow value)$$

But there exists no set solving this, since the cardinality of functions from $(value \rightarrow value)$ is greater than the cardinality of $value$s.

Assume we have infinite objects with some finite approximation as a representation. Object a approximates object b (written $a\ ap\ b$) if b is at least as accurate as a.

Examples:

1. $a\,ap\,b$ if whenever a is defined, b is defined and gives the same answer.

2. Let $[x_1, x_2]$ be a line interval. $[x_1, x_2]\ ap\ [y_1, y_2]$ if $x_1 \leq y_1 \leq y_2 \leq x_2$. That is $[y_1, y_2]$ is contained in $[x_1, x_2]$.

Axiom 1: A datatype is a partially ordered set according to relation ap.

We are interested in the least upper bound of approximations. $F = \{f_1, \ldots, f_n\}$ where $f_1 \; ap \; f_2 \; ap \; \ldots \; ap \; f_n$. But such limits may not always exist. For example, $lub = \cap i$ for $i \in I$ where I is the set of intervals.

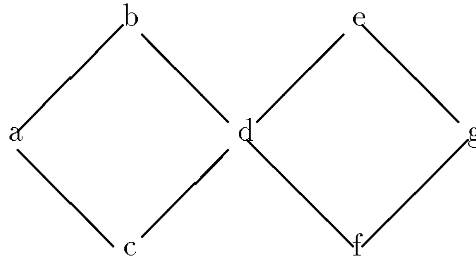
Since we require that such limits exist for every subset of a datatype, we replace Axiom 1 by:

Axiom 1': A *datatype* is a complete lattice under partial ordering ap. Objects of this lattice are called *domains*.

What we mean by this, is for two objects a and b, $a \, ap \, b$ or $b \, ap \, a$ or $a \neg ap \, b$. Also, if $a \; ap \; b$ and $b \; ap \; c$ then $a \; ap \; c$.

Examples:

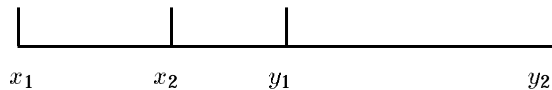
1. Consider the following lattice with 7 objects:



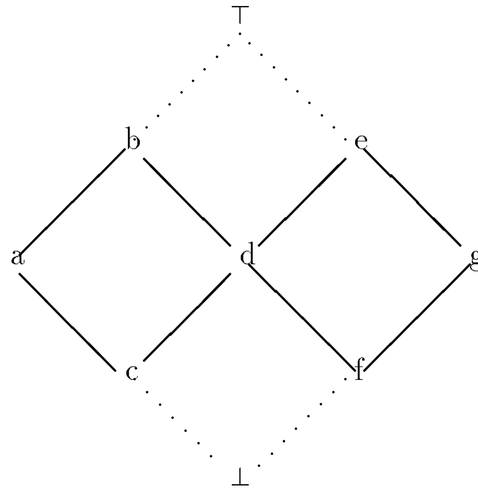
If $i \; ap \; j$ means that i is "lower" than j, we have:

$$\begin{array}{llll} a \; ap \; b & d \; ap \; b & d \; ap \; e & g \; ap \; e \\ c \; ap \; a & c \; ap \; d & f \; ap \; d & f \; ap \; g \end{array}$$

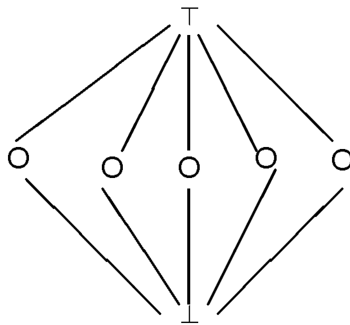
2. Consider $[x_1, x_2]$ and $[y_1, y_2]$ on the line interval:



For set $\{[x_1, x_2], [y_1, y_2]\}$, there is no lub. But one must exist by Axiom 1'. So we hypothesize the existence of $\top$(pronounced "top") and $\bot$(pronounced "bottom") as follows:



3. For discrete sets, we call the following a *primitive domain*:



The simplest primitive domain will consist only of $\top$ and $\bot$ and we call this a *simple domain*.

2.1. Continuous Functions

Axiom 2: Mappings between domains are monotonic. That is, if $f : D \rightarrow D'$, then $(x\ ap\ y \rightarrow f(x)\ ap'\ f(y))$.

Consider a function f which maps some domain into the simple domain: $f : D \rightarrow \mathcal{S}$. Let:

$$f(x) = \top\ if\ x \in D$$
$$f(x) = \bot\ if\ \neg x \in D$$

A set $X \subset D$ is *well defined* iff $\exists p$ and

$$p : D \rightarrow \mathcal{S} \ such \ that \ X = \{w | w \in D \ and \, p(w) = \top\}.$$

p is called a *permissible function*. Note that these are the recursively enumerable sets. However, not all such sets are monotonic and violate Axiom 2. To show this, consider: $NOT : \mathcal{S} \rightarrow \mathcal{S}$ such that:

$$NOT \triangleq (\mathcal{S}\ x)\mathcal{S} : (x = \top \rightarrow \bot) \mid (\top).$$

NOT is not monotonic since $\bot \ ap \ \top$ but

$$(NOT(\bot) = \top) \ \wedge \ (NOT(\top) = \bot) \ \wedge \ (\neg(NOT(\bot) \ ap \ NOT(\top))).$$

This shows that NOT is not permissible. If NOT were permissible, then

$$NOT(p) = (D\ x)\mathcal{S} : NOT(p(x))$$

demonstrating

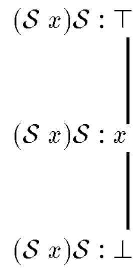
$$\{x | w \in D \ and \ NOT(p(x)) = \top\}$$

would be well defined. But complements of recursively enumerable sets are not necessarily recursively enumerable. Thus we want to restrict ourselves to monotonic functions.

2.2. Continuity

Axiom 3: Mappings under domains are continuous.

This means: $f(lub(X)) = lub\{f(x) | x \in X\}$. Continuous functions between domains also form a domain: $\mathcal{S} \rightarrow \mathcal{S}$. We get the lattice:



Exercise: What happened to: $(\mathcal{S}\ x)\mathcal{S} : NOT(x)$?

2.3. Recursive Functions

Extend integers to a complete lattice:

n_1	n_2	$n_1 \times n_2$
i_1	i_2	$i_1 \times i_2$
$\bot$	i	$\bot$
$\top$	i	$\top$
i	$\bot$	$\bot$
i	$\top$	$\top$
$\top$	$\top$	$\top$
$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$

As an example to analyze, let's look at the factorial function. We will define it as follows:

$$FACT \triangleq (num\ k)num : (k = \mathbf{0} \rightarrow \mathbf{1}) \mid (k \times FACT(k - \mathbf{1})).$$

This equation can be rewritten as: Find the function $FACT$ that solves the equation: $FACT = F(FACT)$, where F is a function from $(num \rightarrow num)$ to $(num \rightarrow num)$, that is, given a function f from $(num \rightarrow num)$ as an argument, $F(f)$ is a function from $(num \rightarrow num)$.

We can then define F as:

$$F \triangleq \quad ((num \rightarrow num)FACT)(num \rightarrow num): \\ (num\ k)num : (k = 0 \rightarrow 1)|(k \times FACT(k-1))$$

This might be made clearer (or totally obscure) by saying that the $FACT$ function is a solution to $x = F(x)$ for F defined as:

$$F \triangleq \quad ((num \rightarrow num)x)(num \rightarrow num): \\ (num\ k)num : \ (k = 0 \ \rightarrow 1)|(k \times x(k-1))$$

F takes argument $(num \rightarrow num)$ (i.e., a function) and produces a $(num \rightarrow num)$ (i.e., another function) such that the function argument takes a num parameter and applies the transformation $(k = 0 \rightarrow 1)|(k \times x(k-1))$. We want the function x such that $x = F(x)$, then this x is called a *fixed point* of F. Consider $f : D \rightarrow D$. x is a *fixed point* of f iff $x = f(x)$.

A fixed point x is a *least fixed point* if for all fixed points $x_1, x_2, \ldots x_n$,

$$x\ ap\ x_1, x\ ap\ x_2, \ldots, x\ ap\ x_n.$$

That is, x is the "largest" fixed point that includes all the other fixed points.

A recursive function has many fixed points. For example, let:

$$g \triangleq (num\ x)num : (x = \bot \rightarrow \bot) \mid (x = \top \rightarrow \top) \mid (g(x))$$

As for solutions, define g_n, for any n, as follows:

$$g_n \triangleq (num\ x)num : (x = \bot \rightarrow \bot) \mid (x = \top \rightarrow \top) \mid (n)$$

For any x, you get $n = g_n(x)$, so all g_n are fixed points of g.

It can be shown that any continuous function on a complete lattice has a least fixed point. Moreover, if $f : D \rightarrow D$ is continuous, the least fixed point $x = f(x)$ is given by:

$$x \triangleq lub\{f_n(\bot)|n = 0, \ldots k\}$$

where

$$f_n(\bot) \equiv f(f \dots f(\bot) \dots)$$
$$f_0(\bot) \equiv \bot$$

2.4. Evaluation of FACT Function

So where are we? Consider the function $FACT$ defined previously. The least fixed point is derived from the previous equations to be:

$$\begin{aligned}
F_0 &= \bot \\
F_1 &= F(\bot) = (num\ k)num : (k = \mathbf{0} \rightarrow \mathbf{1}) \mid (k \times \bot(k - \mathbf{1})) \\
F_2 &= F(F(\bot)) = (num\ k)num : (k = \mathbf{0} \rightarrow \mathbf{1}) \mid (k \times F(\bot)(k - \mathbf{1})) \\
&= (num\ k)num : (k = \mathbf{0} \rightarrow \mathbf{1}) \mid (k \times F_1(k - \mathbf{1})) \\
F_i &= (num\ k)num : (k = \mathbf{0} \rightarrow \mathbf{1}) \mid (k \times F_{i-1}(k - \mathbf{1}))
\end{aligned}$$

$i = 0$	$F_0 = \bot \ \ for\ all\ k$
$i = 1$	$F_1 = (k = \mathbf{0} \rightarrow \mathbf{1}) \mid (k \times F_0(k - \mathbf{1}))$
	$k = \bot \ F_1(k) = \bot$
	$k = \mathbf{0} \ \ F_1(k) = \mathbf{1}$
	$k = \mathbf{1} \ \ F_1(k) = \mathbf{1} \times F_0(\mathbf{0}) = \bot$
	$k \geq \mathbf{2} \ \ F_1(k) = \bot$
$i = 2$	$F_2 = (k = \mathbf{0} \rightarrow \mathbf{1}) \mid (k \times F_1(k - \mathbf{1}))$
	$k = \bot \ F_2(k) = \bot$
	$k = \mathbf{0} \ \ F_2(k) = \mathbf{1}$
	$k = \mathbf{1} \ \ F_2(k) = \mathbf{1} \times F_1(\mathbf{0}) = \mathbf{1}$
	$k = \mathbf{2} \ \ F_2(k) = \mathbf{1} \times F_1(\mathbf{1}) = \bot$
$\vdots$	$\vdots$

So each F_i defines $FACT$ for arguments $\mathbf{0} \leq k < i$, and for all $k \geq i$, we have $F_i(k) = \bot$.

3. PROGRAMMING SEMANTICS

The use of domains and fixed points allows us to describe programming language semantics. The model is a form of operational semantics,

since we are "tracing" through the execution of each statement type in order to determine its effect on a higher-level "interpreter."

However, unlike traditional interpreters that you may be familiar with, we will not consider a "machine" that symbolically executes the program. In this case we will consider a program to be a function, much like the functional correctness model considered earlier. Each statement of the language will be a function, so like the earlier model, we will model the execution of successive statements by determining what the composition function of the two must be.

In the earlier model, we would compute the composition function and give techniques to determine if this composition was the same as an externally given specification of these same statements. In this present chapter, we will simply determine what the composition function must be by describing the transformations that occur to the functions themselves.

Consider the language we have been using in these notes. We can write down a domain equation from each of these BNF rules:

$Stmt =$	$(Id \times Exp)$	Domain of assignments
	$+(Stmt \times Stmt)$	Domain of sequences
	$+(Exp \times Stmt \times Stmt)$	Domain of conditionals
	$+(Exp \times Stmt)$	Domain of iterations

Since **begin** $< stmt >$ **end** only groups statements and does not alter the semantics of the internal $< stmt >$, it was ignored by the domain equation above.

3.1. The Simple Model

As with any interpreter, we need to understand the underlying "machine." We will assume that each identifier refers to a specific location in memory. Aliasing and parameter passing are not allowed, and there is no such concept as a pointer variable. We will extend this model in the next section to include these features.

Therefore for each identifier, there is a unique location in memory containing its value. Stated another way, we can model our concept of memory as a function that returns for each identifier, its unique value. We will call such a function a *store*.

We first must identify three important domains:

1. **Values of storable values.** These are the results of computations and in our example will be the results of expression evaluation. These are the values that identifiers can take.

2. **Eval of expression values.** These are the results of any computations but may not necessarily be storable into an identifier. In our example, we will assume that these can be Boolean or number. Number will be modeled by the primitive domain num given earlier and Boolean will be the domain $bool$ given by:

$$\bot, false, true, \top$$

3. **Denotable values.** These include objects that can be named by identifiers (e.g., procedures, labels, arrays). We will not be using these initially, but the next section introduces this concept.

The storage for a program is a function that maps each location (or id) into its appropriate value. The effect of executing a statement is to create a slightly modified storage. Thus we have a new function (similar to the previous one) that now maps each id into a value – possibly the same or different one. Thus we are viewing execution as the composition of the value storage function followed by the function that describes the execution of a particular statement.

We view a $store$ as a mapping from identifiers to storable values, or a domain of type $id \rightarrow value$, where id is the primitive domain of identifiers in the language. We will call this the *program state*.

In defining the semantics for a programming language, there are three basic functions we need to represent:[1]

1. We need to describe the semantics of a *program*. In this case, the syntax of a program defines a program which has the effect of mapping a number to another number. Or in other words, we are looking for a function $\mathcal{M}$ with signature:

$$\mathcal{M} : prog \rightarrow [num \rightarrow num]$$

This is the same notion as the function $\boxed{\text{PROGRAM} \ldots}$ from Chapter 3.

[1]At this point, it may be useful to review the intuitive representation of these ideas in the introduction to the functional correctness method (Chapter 3).

2. We need to describe a statement in the language. Each statement maps a given state to another state, or:

$$\mathcal{C} : stmt \rightarrow [state \rightarrow state]$$

This simply states that each syntactic statement (domain of $\mathcal{C}$) is a function from $state$ to $state$. Therefore each unique statement will take a program state (or mapping from identifier to value) and produce a new mapping from identifier to value that is the result of all previous statements, including this new statement.

3. The statement function $\mathcal{C}$ depends upon the value of various expressions, so we need to understand the effects of expression evaluation in our language. Each expression is a syntactic entity involving identifiers and each one accesses the storage to produce a value. Thus, from the set of syntactic expressions, a set of storable values produces an expression value. This gives the signature for expression evaluation as:

$$\mathcal{E} : exp \rightarrow [state \rightarrow eval]$$

As a simple example, if exp is the expression $a + b$, then the evaluation function is: $\mathcal{E}(a + b) : state \rightarrow eval$, and applying this function to the state s gives us the function:

$$\mathcal{E}(a + b)(s) \;=\; a(s) + b(s)$$

We will usually write $\mathcal{E}(a+b)$ as $\mathcal{E}\{\!\{a+b\}\!\}$ by putting syntactic arguments in set braces rather than parentheses. It should also be clear, that this notation is the same as $\boxed{expr}$ when we described functional correctness earlier.

We can now model the domains for our programming language:

$state \stackrel{\triangle}{=} id \rightarrow value$	$program\ states$
id	$identifiers$
$value \stackrel{\triangle}{=} eval$	$values$
$eval \stackrel{\triangle}{=} num + bool$	$expression\ values$
num	$integers$
$bool$	$booleans$
exp	$expressions$
$stmt$	$statements$

To define our language, we need to define a function of type $state \rightarrow state$ for each of the syntactic statement types. For ease in readability we will use a λ-like notation for certain constructs. The term:

$$let\ x := a\ in\ body$$

will be used to mean

$$((x) : body)(a)$$

and has a meaning similar to the λexpression:

$$(\lambda x\ body\ a)$$

Recursive definitions will be denoted as:

$$f = rec\ F(f)$$

and will denote the least fixed point of F.

Some auxiliary functions we will use include:

• The conditional design function we used previously in Chapter 3 is extended to a complete lattice: $(b \rightarrow v_1) \mid (v_2)$ as follows:

b	$(b \rightarrow v_1)\mid(v_2)$
$\top$	$\top$
$true$	v_1
$false$	v_2
$\bot$	$\bot$

• If x is of domain type $D \rightarrow D'$, then the expression $x[v/e]$ is defined as:

$$x[v/e] \triangleq \quad (D\ d)D' : (d = v \rightarrow e) \mid (x(d))$$

This has the intuitive meaning of changing the v component of x to e and represents the basic model of an "assignment."

• If y is a domain of type $y = \ldots + x + \ldots$ then the functions of inspection, projection, and injection are defined as:

Inspection: For any y the inspection of y into Y, written ${}_y\mathtt{E}Y$ is defined as:[2]

$${}_y\mathtt{E}Y = true\ if\ y \in Y\ and\ false\ otherwise$$

[2]The actual definitions are more complex than given here, but this suffices for this informal introduction to this subject.

Projection: For any y the projection of y into Y, written $y|Y$ is defined as:

$$y|X \;=\; y\; if\; y \in X,\; or \perp if\; \neg y \in X$$

Injection: For any Y, the injection of Y into X is defined as $Y \subset X$.

Statement semantics

begin: No real semantics added by the **begin** ... **end** sequence.

$$\mathcal{C}\{\!\{begin\; stmt\; end\}\!\} \;=\; \mathcal{C}\{\!\{stmt\}\!\}$$

composition: Result is transformed by the first statement first, then the second.

$$\mathcal{C}\{\!\{stmt_1\;;\;stmt_2\}\!\} \;=\; (state\; s)state : \mathcal{C}\{\!\{stmt_2\}\!\}(\mathcal{C}\{\!\{stmt_1\}\!\}(s))$$

assignment: In this case create a new storage map resulting from the new denotable value. Note that this is just the array assignment axiom we already discussed in Chapter 2, Section 3.

$$\begin{aligned}\mathcal{C}\{\!\{id\; := \; exp\}\!\} \;=\; & (state\; s)state : ((value\; v)state : s[id/v]) \\ & ((\mathcal{E}\{\!\{exp\}\!\}(s)) \mid value)\end{aligned}$$

if: This simply determines which of two functions to evaluate by first evaluating $\mathcal{E}$ on the expression, applying this to the Boolean function, which then evaluates $stmt_1$ or $stmt_2$ as appropriate.

$$\begin{aligned}&\mathcal{C}\{\!\{if\; exp\; then\; stmt_1\; else\; stmt_2\}\!\} \;= \\ &\qquad (state\; s)state : ((bool\; b)state \rightarrow state : \\ &\qquad (b \rightarrow \mathcal{C}\{\!\{stmt_1\}\!\})|(\mathcal{C}\{\!\{stmt_2\}\!\})) \\ &\qquad ((\mathcal{E}\{\!\{exp\}\!\}(s)) \mid bool)(s)\end{aligned}$$

while: This recursive definition is similar to the functional correctness model of Chapter 3.

$$\begin{aligned}&\mathcal{C}\{\!\{while\; exp\; do\; stmt\}\!\} = \\ &\qquad rec(state\; s)state : ((bool\; b)state \rightarrow state : \\ &\qquad (b \rightarrow \mathcal{C}\{\!\{stmt\}\!\} \circ \mathcal{C}\{\!\{while\; exp\; do\; stmt\}\!\})|((state\; s')state : s')) \\ &\qquad ((\mathcal{E}\{\!\{exp\}\!\}(s)) \mid bool)(s)\end{aligned}$$

These definitions correspond to the intuitive meanings we usually associate with these statements.

3.2. Pointers and Aliasing

Our initial model assumed that every identifier referenced a unique value. However, in almost every language, the notion of a reference exists in some form. Even if no explicit pointer exists, the notion is used to describe mechanisms like call by reference or call by name parameter passing to procedures. Allowing parameters also permits *aliasing* of variables, or the ability to address the same memory location with more than one identifier name. This can happen if we pass the same variable as two different parameters, or address a parameter and the external argument directly.

In order to handle this more explicitly, the notion of a “program store” must be extended to the domain of denotable values. For this simple introduction, we will use the domain loc as denotable values.

We add the concept of an *environment* as a mapping from identifiers to these denotable values, or the domain: $env \;=\; id \rightarrow loc$. The “program state” now becomes a “machine store,” or a member of the domain: $store \;=\; loc \rightarrow value$.

If E is an environment, then given any identifier id, $E(id)$ becomes a function returning a loc. For languages without pointers, we can assume that loc will be a constant, so we get the set of environments: $E : id \rightarrow loc$. However, the addition of loc allows us to define identifiers whose location in our store changes.

We need to redefine our semantics to determine the effects of each statement relative to an environment. The previous $\mathcal{C} : stmt \rightarrow [state \rightarrow state]$ now becomes $\mathcal{C}' : stmt \rightarrow [env \rightarrow [store \rightarrow store]]$. The expression evaluation function $\mathcal{E}$ now becomes $\mathcal{E}'$ to interpret the meanings of expressions relative to both an environment and a store.

The modified semantics become:

begin: Same as before:

$$\mathcal{C}'\{\!\{begin\ stmt\ end\}\!\} \;=\; \mathcal{C}'\{\!\{stmt\}\!\}$$

composition: Result is transform by the first statement first, then the second.

$$\begin{aligned}&\mathcal{C}'\{\!\{stmt_1\ ;\ stmt_2\}\!\} = \\ &\qquad (env\ e; store\ s) store : \mathcal{C}'\{\!\{stmt_2\}\!\}(e)(\mathcal{C}'\{\!\{stmt_1\}\!\}(e)(s))\end{aligned}$$

assignment: Create a new storage map.

$$\mathcal{C}'\{\!\{id \ := \ exp\}\!\} = \\ (env\ e; store\ s)store : ((loc\ a; value\ v)store : s[a/v]) \\ (e(id))((\mathcal{E}'\{\!\{exp\}\!\}(e)(s)) \mid value)$$

if: Determine which statement to evaluate.

$$\mathcal{C}'\{\!\{if\ exp\ then\ stmt_1\ else\ stmt_2\}\!\} \ = \\ (env\ e; store\ s)store : ((bool\ b)store \rightarrow store : \\ (b \rightarrow \mathcal{C}'\{\!\{stmt_1\}\!\}(e))|(\mathcal{C}'\{\!\{stmt_2\}\!\}(e))) \\ ((\mathcal{E}'\{\!\{exp\}\!\}(e)(s)) \mid bool)(s)$$

while: Same recursive definition.

$$\mathcal{C}'\{\!\{while\ exp\ do\ stmt\}\!\} = \\ rec(env\ e; store\ s)store : ((bool\ b)store \rightarrow store : \\ (b \rightarrow \mathcal{C}'\{\!\{stmt\}\!\}(e) \circ \mathcal{C}'\{\!\{while\ exp\ do\ stmt\}\!\}(e)) \\ \mid ((store\ s')store : s'))((\mathcal{E}'\{\!\{exp\}\!\}(e)(s)) \mid bool)(s)$$

3.3. Continuations

The one problem with the definitions in the last section is that they are dependent on the order of evaluation. To show this, consider the function which has the effect that for each identifier it returns 0, and it "clears" memory. It will have the signature

$$clear : env \rightarrow [store \rightarrow store]$$

and can be defined as:

$$clear \ = \ (env\ e; store\ s)store : \ (loc\ l) : \mathbf{0}$$

The statement sequence: $stmt_1; clear$ has meaning:

$$\mathcal{C}'\{\!\{stmt_1; clear\}\!\}(e)(s) = \\ \mathcal{C}'\{\!\{clear\}\!\}(e)(\mathcal{C}'\{\!\{stmt_1\}\!\}(e)(s)) = \\ ((env\ e; store\ s) : store(loc\ l)value : \mathbf{0})(e)(\mathcal{C}'\{\!\{stmt_1\}\!\}(e)(s))$$

If we let $\mathcal{C}'\{\!\{stmt_1\}\!\}(e)(s)$ be a nonterminating computation, two possible interpretations can be given to the meaning function, depending upon the interpretation rule. If we choose call by value, the result is also a nonterminating computation. Call by name, however, produces the resulting store $(loc\ l)value$: 0.

The usual method is to assume normal order of evaluation, thus using call by name for λexpressions. This is not the usually assumed semantics for programming languages. Two approaches can be used to give call by value semantics:

• If f is a member of the domain $D \rightarrow D'$, then we can define a function $call_{value} : [D \rightarrow D'] \rightarrow [D \rightarrow D']$ that effectively mimics f where it terminates. We define it as:

$$call_{value}(\bot) = \bot'$$
$$call_{value}(\top) = \top'$$
$$call_{value}(f)(d) = f(d),\ otherwise$$

Then every place where call by value is required, replacement of the function f by $call_{value}$ will produce the proper results.

• We can alter the definitions so that their interpretation is independent of the order of evaluation. In doing so, call by name achieves our purpose. The use of continuations allows for this.

Call a computation *serious* if it is possibly nonterminating. The problem we have is the application of functions to serious arguments. To remove this possibility, extend each serious function to include an additional argument which will be used to represent the remainder of the computation to be performed (i.e., the *continuation*). If the serious function terminates, it will apply the continuation to the result produced. Thus, a serious function $f(x_1, \ldots, x_n)$ in the previous definition will be mapped into $g(x_1, \ldots, x_n, c)$ such that:

$$g(x_1, \ldots, x_n, c) \ = \ c(f(x_1, \ldots, x_n))$$

4. EXERCISES

Problem 1. Show that the λcalculus definitions for and and or agree with our intutive definitions for these logical functions; that is $and(a, b)$ is true if both a and b are true and $or(a, b)$ is true if either a or b are true.

Problem 2. Using the λcalculus definitions, show the following:

$$
\begin{aligned}
&1 + 1 = 2 \\
&2 + 3 = 5 \\
&2 \times 3 = 6 \\
&2^2 = 4
\end{aligned}
$$

Problem 3. If we allow side effects in expression evaluation, we need to modify the definition of $\mathcal{E}$ to something like:

$$\mathcal{E} : exp \rightarrow [state \rightarrow [eval \times state]]$$

Modify the model presented here to take into account side effects.

Problem 4. Assume we add a **for** statement to our language with the syntax:

$$< stmt > \rightarrow \quad \mathbf{for} \ < id > := < expr > \ \mathbf{to} \ < expr > \ \mathbf{do} \ < stmt >$$

Show that the following is a reasonable semantic definition for this statement:

$$
\begin{aligned}
&\mathcal{C}\{\!\{for\ id := exp_1\ to\ exp_2\ do\ stmt\}\!\} \ = \\
&\quad (state\ s)state : \\
&\quad let\ iterate = rec(num\ n_1, num\ n_2; state\ s')state : \\
&\quad if\ n_1 > n_2\ then\ s'\ else\ iterate(n_1 + 1, n_2)(\mathcal{C}\{\!\{stmt\}\!\}(s'[id/n_1])) \\
&\quad in \\
&\qquad (num\ v_1, num\ v_2)state \rightarrow state : if\ v_1 = \bot\ then\ \bot \\
&\qquad else\ if\ v_2 = \bot\ then\ \bot\ else\ if\ v_1 = \top\ then\ \top \\
&\qquad else\ if\ v_2 = \top\ then\ \top\ else\ iterate(v_1, v_2)) \\
&\qquad ((\mathcal{E}\{\!\{exp_1\}\!\}(s)) \mid num, (\mathcal{E}\{\!\{exp_2\}\!\}(s) \mid num))\ (s)
\end{aligned}
$$

Problem 5. Modify the model of Section 3.2 to allow for dynamic environments with the addition of a pointer-assignment statement. That is, add the statement $id_1 >> id_2$ which means that any reference to id_1 after the assignment will be to the same location as the current value of id_2. Redo all other semantic equations to conform to this new addition.

5. SUGGESTED READINGS

The lambda expression is a simple predecessor to denotational semantics. The Wegner reference is a good summary of that.

• D. A. Schmidt, *Denotational Semantics*, Allyn and Bacon, New York, 1986.

• P. Wegner, *Programming Languages, Information Structures and Machine organization*, McGraw Hill, New York, 1968, pp. 180-196.

• R. D. Tennent, "The denotational semantics of programming languages," *Communications of the ACM*, Vol. 19, No. 8, 1976, pp. 437-453.

Chapter 7

Specification Models

Previous chapters have described various notations for verifying that a program meets its specifications. However, most notations are hard to adapt to realistic programs. Here we look at approaches that attempt to model real programming issues, based upon the formal methods we already discussed.

We will first describe VDM, a notation that uses many of the techniques we have already explained. We complete this chapter with a summary of some recent results on risk analysis and evaluation of software specifications.

1. VIENNA DEVELOPMENT METHOD

The IBM Vienna Laboratory developed a notation called the Vienna Development Method (or VDM[1]). As we shall see, VDM adds very little new technology to what has already been described, except, perhaps, for an arcane notation. What VDM does do, however, is to pick and choose from among the techniques we have studied, and to develop a formalism and presentation that lends itself to realistic problems. VDM has a much wider following in Europe today, and is being used for many difficult programming tasks.

[1]VDM is not to be confused with the Vienna Definition Language, or VDL, which was briefly described earlier.

While VDM was initially developed within IBM, the IBM group moved on to other topics and major development of the current model was by Dines Bjorner of Lyngby, Denmark, who was concerned about systems software specifications and by Cliff Jones in Manchester, UK, who was concerned about VDM proof obligations and algorithm and data structure refinement.

VDM contains the following features:

• The basic concept is to define a specification of a program, define a solution to that specification, and prove that the program meets the specification.

• It is based upon denotational semantics and Mills' functional correctness models in that it develops functions of the underlying program, and is concerned about transformations between program states.

• It permits formal proofs of program properties using a set of inference rules, similar to the Hoare rules we studied earlier.

1.1. Overview of VDM

We present this short overview of VDM by describing the four basic features in the method:

1. the underlying mathematical logic,
2. the definition of functions,
3. the definition of specifications, and
4. the method for verifying a function with its specification.

Underlying mathematical logic

VDM uses the predicate calculus for proving properties of specifications. That means the mathematical theory presented earlier in Chapter 1, Section 5.4 forms the basis of the proof methodology. The same set of rules of inference apply to this method as well as to the previously defined methods. VDM proofs look very similar to the Hoare axiomatic proofs we studied in Chapter 2.

One major difference between the model we are building here and traditional mathematics is that traditionally, functions are usually considered to be total – that is, defined for all members of the domain. However, programs rarely are total. They are usually defined to give an appropriate answer for specific input data (i.e., only data that meets a program's precondition). The implication of this, is that given predicate E, we can state in classical mathematics that $E \vee \neg E = true$. However, for a programming logic, we cannot assume this.

In order to handle this problem, the operator δ will be defined. $\delta(E)$ means that E is defined (to be $true$ or $false$). More importantly, in this context, this also implies that E must be a terminating computation.

For example, the inference rule:

$$\frac{E_1 \vdash E_2}{E_1 \Rightarrow E_2}$$

now becomes

$$\frac{E_1 \vdash E_2;\ \delta(E_1)}{E_1 \Rightarrow E_2}$$

The need for δ occurs in the axioms for the **if** and **while** statements. The VDM axioms for these two statements are given as follows:

$$\frac{\{pre \wedge B\} stmt_1 \{post\};\ \{pre \wedge \neg B\} stmt_2 \{post\}; pre \Rightarrow \delta(B)}{\{pre\} \textbf{if } B \textbf{ then } stmt_1 \textbf{ else } stmt_2 \{post\}}$$

$$\frac{\{I \wedge B\} stmt \{I\};\ I \Rightarrow \delta(B)}{\{I\} \textbf{while } B \textbf{ do } stmt \{I \wedge \neg B\}}$$

Definition of functions

A function is defined as:

$$FunctionName : signature$$
$$FunctionName(parameters) \triangleq expression$$

where $FunctionName$ is the name of the function, $signature$ is the signature of the function, $parameters$ are the parameters, and $expression$ is the definition of that function.

In order to specify parameters, we need to give the data type of each such specified object. Several predefined data types are defined:

Token:single objects.

Enumeration:the enumerated type.

$\mathcal{B}$**:**truth values.

$\mathcal{N}$**:**the natural numbers.

$\mathcal{Z}$**:**the integers, and.

$\mathcal{R}$**:**the reals.

VDM supports six classes of data objects: sets, sequences, maps, composite objects, Cartesian products, and unions. For example, sets of object X are given as $X - -\mathsf{set}$. Thus a set of integers would be given as $\mathcal{Z} - -\mathsf{set}$.

For example, a function to multiply two natural numbers could be given as:

$$multiply : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N}$$
$$multiply(x, y) \triangleq x \times y$$

Definition of specifications

A specification consists of four basic components:

1. name
2. external data
3. precondition
4. postcondition

The structure of each of these components is as follows:

1. **name:** $Name$ gives the name of the function to be defined and its signature. It uses a notation similar to a Pascal function declaration. For example, to create a function $multiply$ that takes two parameters x and y and produces an integer result z, the specification would be:

$$multiply : (x : \mathcal{Z}; y : \mathcal{Z}) : z : \mathcal{Z};$$

In this case, $z = multiply(x, y)$.

2. **external data:** The external data segment of a specification is optional. When present it specifies the name of the variables used by the specification, but defined elsewhere. The syntax is:

$$\mathsf{ext}\ names : \ mode\ type$$

where $names$ is a list of identifiers used by the specification and $mode$ determines how each variable is used. A mode of wr means that the variables are "read–write" and can be altered by the specification, while a mode of rd means that the variables are "read-only" and cannot be altered.

For example, if integers a and b are accessed by a specification and c is altered, the external declaration could be given as:

$$\mathsf{ext}\ a, b : \ \mathsf{rd}\ \mathcal{Z};$$
$$\mathsf{ext}\ c : \ \mathsf{wr}\ \mathcal{Z};$$

3. **precondition:** The precondition is specified as:

$$\mathsf{pre}\ expression$$

where $expression$ is any Boolean expression involving the variables of the specification.

For the $multiply$ example given above, if we let $\mathsf{pre} - -multiply$ be a predicate describing the precondition, we can refer to this precondition in formulae as $\mathsf{pre}{-}multiply(x, y)$, which in this case would just be the predicate $true$.

4. **postcondition:** The postcondition is specified as:

$$\mathsf{post}\ expression$$

where $expression$ is any boolean expression involving the variables of the specification. The postcondition generally specifies the functionality that is desired by the specification.

Often it is necessary to specify the output of a program as a function of the initial input values. In VDM the symbol $\leftharpoondown$ means an initial value. Thus if A is a parameter to a specification, then $\overleftharpoondown{A}$ means the value A had when the function was initially invoked.

For the $multiply$ example given above, the postcondition is denoted as:

$$\begin{aligned}&\mathsf{post} - -multiply(x, y, z) = \\ &\quad \mathsf{post} - -multiply(x, y, multiply(x, y))\end{aligned}$$

This can be clarified further as:

$$\mathsf{post} - -multiply(x, y, \overleftharpoondown{x} \times \overleftharpoondown{y})$$

This was an example of an operational abstraction for a specification. VDM also permits applicative statements to specify functions and imperative statements to specify operations.

1.2. Example: Multiplication One Last Time

In presenting a brief overview of VDM, let us revisit our multiplication program $MULT$ that we developed earlier in Chapter 2, Section 1.3. Using VDM, we can give the specification to this program as follows:

$$\begin{aligned}&MULT : (A : \mathcal{N};\ B : \mathcal{N})\ y : \mathcal{N} \\ &\quad \mathsf{ext}\ a, b : \mathsf{wr}\ \mathcal{N}; \\ &\quad \mathsf{pre}\ B \geq \mathbf{0} \\ &\quad \mathsf{post}\ y = \overleftharpoondown{A} \times \overleftharpoondown{B}\end{aligned}$$

This states that $MULT$ is a function with two arguments (A and B) of type $\mathcal{N}$(natural number) that computes a result of type $\mathcal{N}$. It accesses state variables (i.e., global storage) a and b and writes to them.

This function has as a precondition $b \geq 0$ and as a postcondition $y = \overleftharpoondown{A}\overleftharpoondown{B}$, where $\overleftharpoondown{x}$ is the original value of x on entry to the function.

Any program that meets this specification must satisfy the following predicate:

$$\forall (i,j) \in \mathcal{N} \times \mathcal{N} \mid \mathsf{pre}--MULT(i,j) \Rightarrow$$
$$MULT(i,j) \in \mathcal{N} \ \wedge \ \mathsf{post}--MULT(i,j,MULT(i,j)) \equiv$$
$$\forall (i,j) \in \mathcal{N} \times \mathcal{N} \mid j \geq \mathbf{0} \Rightarrow$$
$$MULT(i,j) \in \mathcal{N} \ \wedge \ MULT(i,j) = \overleftarrow{i}\,\overleftarrow{j}$$

We will first specify two functions $INIT$ and $LOOP$ whose composition gives us the above result. Their specification is as follows:

$INIT : (A : \mathcal{N};\ B : \mathcal{N})\ y : \mathcal{N}$
ext a, b : wr $\mathcal{N}$;
pre $B \geq \mathbf{0}$
post $a = \overleftarrow{A} \wedge b = \overleftarrow{B} \wedge y = \mathbf{0}$
$\wedge b \geq \mathbf{0}$

$LOOP : y : \mathcal{N}$
ext a, b : wr $\mathcal{N}$;
pre $b \geq \mathbf{0} \wedge y = \mathbf{0}$
post $y = \overleftarrow{a}\,\overleftarrow{b}$

We now show that our previous solution meets this specification.

INIT function

We will define our candidate solution for $INIT$ using the same three initialization statements as before:

$$INIT \ \triangleq \ a := A; b := B; y := \mathbf{0}$$

Our rules of inference are very similar to the Hoare axioms discussed previously. However, we want to make sure that each function terminates as part of the rule. For assignment, this is not a problem; potential problems only occur during loop statements.

The two rules of inference for assignment that we need are:

$$\mathbf{asgn_1} : \frac{}{\{true\}x := e\{x = \overleftarrow{e}\}} \qquad \mathbf{asgn_2} : \frac{x \textit{ does not occur in } E}{\{E\}x := e\{E\}}$$

We can prove $\mathsf{pre}{-}INIT \Rightarrow \mathsf{post}{-}INIT$ via a proof similar to a Hoare-type axiomatic proof:

$\{B \geq 0\}a := A\{B \geq 0\}$ $asgn_2$

$\{true\}a := A\{a = \overleftarrow{A}\}$ $asgn_1$

$\{B \geq 0\}a := A\{B \geq 0 \wedge a = \overleftarrow{A}\}$ $Consequence$

$\{B \geq 0 \wedge a = \overleftarrow{a}\}b := B\{B \geq 0 \wedge a = \overleftarrow{a} \wedge b = \overleftarrow{B}\}$ $asgn_1$ & $asgn_2$

$\{B \geq 0\}a := A; b := B\{B \geq 0 \wedge a = \overleftarrow{a} \wedge b = \overleftarrow{B}\}$ $composition$

$\{B \geq 0\}a := A; b := B; y := 0\{B \geq 0 \wedge a = \overleftarrow{a} \wedge b = \overleftarrow{B} \wedge y = 0\}$ $asgn_1$ & $asgn_2$ & $composition$

$\{B \geq 0\}a := A; b := B; y := 0\{b \geq 0 \wedge a = \overleftarrow{A} \wedge b = \overleftarrow{B} \wedge y = 0\}$ $consequence$

LOOP function

We will define the same **while** statement to solve this $LOOP$ specification:

$$LOOP \stackrel{\triangle}{=} \textbf{while } b > 0 \textbf{ do begin } b := b - 1; y := y + a \textbf{ end}$$

We need the following inference rule. If I is the invariant, T is the predicate on the **while** statement, and S is the loop body:

$$\frac{\{I \wedge T\}\ S\ \{I\}, I \Rightarrow \delta(T)}{\{I\}\textbf{while}\cdots\{I \wedge \neg T\}}$$

where $\delta(T)$ states that T is *well founded*, that is, the predicate eventually becomes false and the loop terminates.[2]

Using steps similar to the axiomatic proof in Chapter 2, Section 1.3, we are able to complete the proof in a similar manner.

1.3. Summary of VDM

This example by no means describes all of the features of VDM. However, the simple exercise should indicate the essential characteristics of a VDM development:

• It is better than most methods at separating the specification from implementation issues.

[2]Formally, we must show that there is no function f such that $f(i)\ LOOP\ f(i+1)$ is a relation, meaning that we eventually have to find some value of f outside of the domain of the test, and the test must then fail.

• It can be completely formal, but can also be used informally.

• It can be used at several levels – proving a design meets a specification, or proving that a source program meets the specs.

• While practical, the process is still not easy.

2. TEMPORAL LOGIC

As we have discussed the issue of program correctness in this book, we have been concerned about the effects of certain programming language structures on the results of executing that program. Thus, we have been mostly addressing the question "If I execute a program under certain input constraints, will I get the desired result?" No mention is ever made of *when* one can expect such results. Most correctness models simply deal with the issue of whether the desired results eventually will or will not occur. We have ignored the passage of *time* in this discussion.

We can, however, include time within our logical system, and we call such models *temporal logics*. In this section, we briefly describe such logics and outline their major properties.

2.1. Properties of Temporal Logics

When we considered axiomatic correctness (via Hoare axioms of Chapter 2 or predicate transforms of Chapter 4) we were mostly concerned with the question: "If predicate P is true and program S is executed, will predicate Q be true whenever S terminates?" When we consider the question of *when* program S terminates, we can ask additional questions. The following list is indicative of the kinds of issues temporal logic addresses:

• **Safety property:** Can something bad ever happen?

• **Liveness property:** Does something good eventually happen?

• **Deadlock property:** Does the program ever enter a state where no further progress is possible?

• **Mutual exclusion:** Can we ever have two different processes in the same critical state at the same time?

We can also represent our traditional programming logic as a temporal question: If the precondition is initially true, then is it possible for the program to terminate with the postcondition false?

For programs that execute sequentially, such as those usually written in Pascal, FORTRAN, or even C, the techniques given earlier in this book should suffice. However, when we consider simultaneous execution of sets of these programs, such as the concurrent execution of a set of processes within an operating system, the results are not so clear, and it is necessary to resort to techniques, like temporal logic, to determine the correct behavior of the system.

Primitive temporal operators

Predicate calculus provides for the evaluation of expressions of Boolean variables. In order to discuss time, we need to add two operators to our predicate calculus (of Chapter 2, Section 5):

- $\Box$: Now and forever
- $\rightarrow$: Now or sometime in the future

Thus $\Box p$ means that p is true now and remains true forever. The expression $\rightarrow p$ means that p is either true now or will eventually become true.

The differences between these operations can be given by the following examples:

- $x > 0$ — The expression is now true, and may be true or false in the future.
- $\Box(x > 0)$ — The expression is now true and will remain true forever.
- $\rightarrow (x > 0)$ — The expression will sometime be true (either now or later).

We will call our expressions of temporal logic *assertions*. Any assertion which does not contain any of the temporal operators will be called a *predicate* and is obviously equivalent to the predicate calculus predicates we studied before.

Non-primitive operators

For convenience, we will introduce the following two nonprimitive operators: $\Diamond$ and $\leadsto$.

This $\Diamond$ is defined as the dual to $\Box$. That is:

$$\Diamond p \triangleq \neg\Box\neg p$$

Since $\Box\neg p$ states that it is always true that p is never true (i.e., that p is always false), $\Diamond p$ states that it is *not* never true that p is always false. So does this mean that $\Diamond p$ is sometimes true and $\Diamond p \equiv\rightarrow p$? The answer is an unqualified "maybe."

To understand this statement, consider that developing temporal assertions is further complicated by the realization that we have at least two definitions of time. In one case, we view time as a linear sequence of events, while in the other we view time as all possible events that may occur in the future, much like the distinction between nondeterministic and deterministic state machines when discussing formal computational models. We call the former *linear-time* and the latter *branching-time* temporal logic.

With linear-time logic, there is a single time line of future events determining what will happen. There is only one possible future, so "not never" does indeed mean sometime, and we do have the equivalence: $\Diamond p \equiv\rightarrow p$.

However, with branching time logic, at each instance there may be alternative applicable choices, thus forming a tree of future actions. p may become true on some path that depends upon a future decision. Since p may become true since that path may be traversed, we can say that $\Diamond p$ is true. However, we cannot say that $\rightarrow p$ is true since we do not yet know if that path that makes p true *must* be traversed.

Obviously, a single future to deal with is simpler than branching-time logic, and we will restrict the discussion in this chapter to linear time logic where $\Diamond p \equiv\rightarrow p$.

A second nonprimitive operator is leadsto, the "leads to" operator, which is defined to be:

$$P \leadsto Q \triangleq \Box(P \Rightarrow \Diamond Q)$$

That is, $P \leadsto Q$ means that it is always true that if P is ever true, then eventually (i.e., sometimes) Q will be true. We will see this operator again when we discuss verification conditions for concurrent programs.

States

Much like other programming language models described earlier, we can describe temporal assertions by their effects upon system *states*. However, since we need to consider concurrent execution where several program statements may execute next, we need to include a "next state" component to our model. In the sequential model, we could have simply modeled the next state function as a mapping from location counter to statement number. However, here there may be more than one next state ready for execution.

We define a state s as the pair: (1) a function from names to values, and (2) a set *ready* giving the set of statements available for execution. We let S be the set of all such states.

In discussing the issues of safety, liveness, and other such properties, we are usually concerned about the sequence of states that a program passes through. We write such a sequence as $s = s_0, s_1, s_2, \ldots$ where each s_i is in S.

If s is a sequence of states, the tail of that sequence (i.e., the sequence with the first element deleted) is given by s^+, that is: $(s^+)_i = s_{i+1}$.

We define the *semantics* of a program as all valid execution sequences that a program may have. For traditional programming languages, there will be a single execution sequence describing the behavior of a program; however, as we add parallel processing and distributed computing primitives, a program may have more than one valid execution sequence. (We saw some of this with the guarded commands of Dijkstra earlier.)

Immediate assertions

We combine our notion of predicates with concurrent states via *immediate assertions* and the operator $\models$. We write $s \models P$ to state that P has value true in state s. That is, if $s = < f, ready >$, then values of f applied to P result in P being true.

We can combine the previously defined temporal operators $\Box$ and $\Diamond$ with our notion of states, as follows:

$s \models (P \wedge Q)$	if and only if $s \models P$ and $s \models Q$.
$s \models (P \vee Q)$	if and only if $s \models P$ or $s \models Q$.
$s \models \neg P$	if and only if it is not the case that $s \models P$.

For sequences, we can state:

$s \models \Box P$	if and only if $\forall i,\ s_i \models P$
$s \models \Diamond P$	if and only if $\exists i,\ s_i \models P$

Formal properties of temporal logic

With the above definitions, we can now formally define several properties that we wish to investigate with concurrent programs. In what follows, $P \Rightarrow Q$ means that if there is some sequence s for which $s \models P$, then $s \models Q$. For sequence s with $s \models P$ we will assume that state s_0 satisfies $s_0 \models P$:

1. $P \Rightarrow \Box Q$. This is the basic safety property we would like to prove. That is, if P is true now, then Q will always remain true.

2. $\Box(I \Rightarrow \Box I)$. If I ever becomes true, then I will remain true. This states that I is an invariant.

3. $\Box(P \Rightarrow \Diamond Q)$. This states that if P becomes true then eventually Q will be true. This is the essential liveness property.

4. The following are all theorems that can be proven:

$$\Box(P \Rightarrow Q) \Rightarrow (P \leadsto Q)$$
$$\Box(P \wedge Q) \equiv \Box P \wedge \Box Q$$
$$\Diamond(P \vee Q) \equiv \Diamond P \vee \Diamond Q)$$

Duality of the second theorem, however, does not follow, since the following is not a theorem:

$$\Box(P \vee Q) \equiv \Box P \vee \Box Q$$

However, the following is a theorem:

$$\Box P \vee \Box Q \Rightarrow \Box(P \vee Q)$$

2.2. Programming Assertion

Consider a programming language consisting of **if, while**, and assignment statements. We need to extend such a language with two constructs to handle concurrency: (a) concurrent execution and (b) atomicity.

• Concurrent execution is modeled with the **cobegin** statement:

$$\mathbf{cobegin}\ s_1 [\!] s_2 [\!] \ldots\ \mathbf{coend}$$

meaning that each s_i can execute concurrently. Execution does not continue until all components of the **cobegin** terminate.

• Atomicity is modeled by the operator $< \ldots >$. This means that once execution of this component of the program begins, it proceeds to the end. For sequential programs, this is not a problem, but with concurrent execution, we need such a rule in order to determine the outcome of many computations.

For example, consider the following program:

$$\begin{array}{l} x := 1; \\ \mathbf{cobegin} \\ \quad x := 2 [\!] \\ \quad y := < x + x > \\ \mathbf{coend} \end{array}$$

If we did not have atomicity, the computation of y could result in any one of the following: $1+1$, $2+2$, or even $1+2$ depending upon the order that the first or second **cobegin** statements were executed in. Because of concurrency, we could also have two different values of x in the same expression, depending upon evaluation order. Atomicity forces a result of $1+1$ or $2+2$ only. We can be sure that the value of x does not change within the computation of that single expression.

We define three immediate assertions on such a programming language as follows:

1. **at:** $at\ P$ refers to states where execution is about to execute statement P. $s \models at\ P$ if and only if:

- If P is $l : x := expr$, then $l \in ready(s)$.

- If P is l : $\mathbf{while}$ B $\mathbf{do}$ Q, then $l \in ready(s)$.
- If P is $\mathbf{cobegin}$ $s_1 \| s_2 \| \ldots \mathbf{coend}$, then ($s \models at\ s_1$) and ($s \models at\ s_2$).
- If P is $Q; R$ then $s \models at\ Q$.

2. **in:** We state that we are $in\ P$ if we are executing some component making up P. We can define it as: $s \models in\ P$ if and only if either $s \models at\ P$ or there is some component of Q of P such that $s \models at\ Q$.

3. **after:** We state that $after\ P$ is true if we are finished with the execution of P or any component of P.

Let Q be the statement that contains P. $s \models after\ P$ if and only if:

- If P is the entire program, then $ready(s) = \emptyset$.
- If Q is l : $\mathbf{while}$ B $\mathbf{do}$ P, then $s \models at\ Q$
- If Q is $\mathbf{cobegin}$ $s_1 \| P \| \ldots \mathbf{coend}$, then $(s \models after\ Q)$ or (($s \models in\ Q$) and not ($s \models in\ P$)).
- If Q is $P; R$, then $s \models at\ R$.
- If Q is $R; P$, then $s \models after\ Q$.

Inference rules

With the in, at, and $after$ assertion rules, we are now in a position to define a set of inference rules for concurrent programs, much like the earlier axiomatic inference rules.

- **Atomic assignment:**

$$\frac{< S >}{at\ S \leadsto after\ S}$$

- **Single exit:**

$$\frac{true}{in\ S \Rightarrow (\Box in\ S \vee \Diamond after\ S)}$$

- **While axioms:** Fairness is given as:

$$\frac{w:\ \textbf{while}\ <b>\ \textbf{do}\ S}{at\ w \leadsto (at\ S \vee after\ w)}$$

While execution is given by:

$$\frac{w:\ \textbf{while}\ <b>\ \textbf{do}\ S}{at\ w \leadsto ((at\ S \wedge b) \vee (after\ w \wedge \neg b))}$$

While termination is given by:

$$\frac{w:\ \textbf{while}\ <b>\ \textbf{do}\ S}{(at\ w \wedge \Box(at\ w \Rightarrow b)) \leadsto at\ S,\ (at\ w \wedge \Box(at\ w \Rightarrow \neg b)) \leadsto after\ w}$$

- **Concatenation rules:**

$$\frac{S;T,\ at\ S \leadsto after\ S,\ at\ T \leadsto after\ T}{at\ S \leadsto after\ T}$$

- **Cobegin rule:**

$$\frac{c:\textbf{cobegin}\ S \| T\ \textbf{coend},\ at\ S \leadsto after\ S,\ at\ T \leadsto after\ T}{at\ c \leadsto after\ c}$$

- **Atomic statement:**

$$\frac{\{P\} <S> \{Q\},\ \Box(at\ <S> \Rightarrow P)}{at\ <S> \leadsto (after\ <S> \wedge Q)}$$

- **General statement:**

$$\frac{\{P\}S\{Q\},\ \Box(in\ S \Rightarrow P),\ in\ S \leadsto after\ S}{in\ S \leadsto (after\ S \wedge Q)}$$

This only scratches the surface on temporal logics. See the paper by Owicki and Lamport[48] for more details.

3. RISK ANALYSIS

While there are many other attributes which can affect a specification, their enumeration is beyond the scope of this book. Suffice it to say that resource estimation and safety analysis are representative of the others.

Assume we have a set of such attributes which define a specification to a program. How does one choose an appropriate design that meets that specification? We know that a complete specification needs to address many attributes, including functionality, resource usage, security, safety, and other concerns. Most designs will meet certain requirements well and probably not meet others.

How do we know what project to build? How do we assess the risk for each decision? Will prototyping reduce this risk? All of these are important considerations, and have only recently been applied to the specification problem.

We now describe one such evaluation strategy. It consists of two components. In one, the designer knows all relevant details (*decision under certainty*). In the second case, there is some variability (and risk) associated with all potential choices. We call this *decision under uncertainty*.

3.1. Decisions under Certainty

We consider correct functionality to be just one of several attributes for a solution, with multiple designs implementing the same functionality. Let us first assume that our needed functionality is specified by a function (from state to state) and also the candidate programs are specified by functions from state to state. Let X be the functionality of program x. We extend this model to include other attributes as well. Since these other attributes are often concerned with nonfunctional characteristics such as resource usage, schedules, and performance, we will use the term *viable* for any solution satisfying a specification rather that the more specific term *correctness*.

Now assume that our specifications (for both our needed software and the candidate programs) are vectors of attributes, including the functionality as one of the elements of the vectors. For example, X and Y are vectors of attributes that specify alternative solutions to a specifi-

cation B. Let S be a vector of objective functions with domain being the set of specification attributes and range $[0..1]$. We call S_i a *scaling function* and it is the degree to which a given attribute meets its goal. We state that $X\ solves_S\ Y$ if $\forall i, S_i(X_i) \geq S_i(Y_i)$. We extend our previous definition of correctness to the following: design x is viable (i.e., is correct) with respect to specification B and scaling function vector S if and only if $P\ solves_S\ B$. We can show that the previous definition of correctness is simply a one-dimensional example of this more general definition of viability.

Each attribute may not have the same importance. Assume a vector of weights W called *constraints* such that each $w_i \in [0..1]$ and $\sum w_i = 1$.

Our evaluation measure, the *performance level*, merges multiple scaled attributes and their constraints. Given specification vector X, scaling function S and constraints W, the performance level is given by: $PL(X, S, W) = \sum_i (w_i \times S_i(X_i))$.

We use the performance level as our objective function: Given a specification vector B, scaling vector S, constraints W, and potential solutions x and y, $X\ improves\ Y$ with respect to $\langle B, S, W\rangle$ if and only if:

1. $X\ solves_S\ B$ and $Y\ solves_S\ B$
2. $PL(X, S, W) > PL(Y, S, W)$.

We use here a very simple weighted sum to compute the performance level. Our definition of *improves* depends only upon an appropriate definition of performance level for comparing two solutions, not on the details of how the two vectors are compared.

It should be noted that the model presented in this section depends upon the solution triple $\langle B, S, W\rangle$, which is a quantitative evaluation of how well each attribute of the proposed solution meets or exceeds the minimal specification B. We rarely know this in practice and this book only assumes an ordinal ranking of the attributes – that is, one attribute value is better than another.

3.2. Decisions under Uncertainty

We have so far assumed that the relative importance of each attribute is known a priori. However, we rarely know this with certainty. We

therefore consider the following model, based upon aspects from economic decision theory.

The performance level assumes that the relative importance of each attribute is a known constant, so the weight factors and scaling can be defined. However, this is not generally true. For example, in a program that includes a sort of a list of records, the importance of the sort algorithm itself depends upon how often it gets called and how long unsorted lists get. That is, if the list of items always remains short, then any sort algorithm will suffice, since sorting will take a negligible part of the execution overhead. In this case, any attribute value (i.e., specification) describing the sort function will have minimal effect upon the resulting program and have a very low weight. Our problem is then to modify the previous model to account for unknowns in the importance for these attribute values.

Using terminology from decision theory, the potential solutions to a specification are called *alternatives*, and the various possibilities that will determine the importance for the attribute values are *states of nature*. Each state of nature is associated with a fixed set of weights giving the relative importance of each system attribute.

We can now represent the performance level as a matrix PL where $PL_{i,j}$ is the performance level for solution i under state of nature j. As before, the performance levels give a measure of how good a system is. We can approximate this by defining the entries $PL_{i,j}$ of performance level matrix PL as the payoff (e.g., monetary value) for solution i under state j. For example, assume we have two potential solutions X^1 and X^2, and assume we have three potential states of nature st_1, st_2 and st_3 which are represented as the 6 possible *payoffs* in the matrix:

$$PL = \begin{bmatrix} 100 & 500 & 0 \\ 300 & 200 & 200 \end{bmatrix} \tag{7.1}$$

In this example, if we knew for sure that st_2 would be the resulting state of nature, then we would implement alternative X^1 (with payoff 500), and if we knew that either states st_1 or st_3 were the resultant states, then alternative X^2 would be most desirable. However, we may not know this beforehand.

When the probability for each state of nature can be estimated, we can use expected values to achieve an estimated performance level. Given probability distribution vector P, where p_i is the probability that state

of nature st_i is true, the expected payoff for alternative X^i is given by:

$$v_i = \sum_j pl_{i,j} \; p_j$$

Use the decision rule: Choose X^i which maximizes v_i or:

$$\max_i (\sum_j pl_{i,j} \; p_j)$$

For example, if we know that the probability distribution for each state of nature in our example is: P = (0.3, 0.5, 0.2), we can calculate the expected payoffs as follows:

$$\begin{aligned} v_1 &= 100 \times 0.3 + 500 \times 0.5 + 0 \times 0.2 \\ &= 280 \\ v_2 &= 300 \times 0.3 + 200 \times 0.5 + 200 \times 0.2 \\ &= 230 \end{aligned}$$

We would then choose X^1 over X^2 since $280 > 230$.

3.3. Risk Aversion

Risk aversion plays an important role in decision making. This implies subjective behavior on the part of the software manager. We assume that the following *reasonable* behavior rule (i.e., equilibrium probability) is true:

- *Decomposition:* Given three payoffs $a \leq b \leq c$, there exists a probability ρ such that the decision maker is indifferent to the choice of a guarantee of b, and the choice of getting c with probability ρ and getting a with probability $1 - \rho$. We shall refer to this probability as $decomp(a, b, c)$.

For example, assume there are two techniques to solve a problem. One is fully tested, giving a guaranteed payoff of $5,000 and a second new and more efficient technique promises a potentially larger payoff of $10,000 (but not completely tested) with a chance to give a payoff of only $2,000. If a software manager considers using the new technique only if the chances of getting the payoff of $10,000 are larger than 80%, the probability ρ is larger than 0.8. In this case, the expected payoff will be 10,000 $\times$ 0.8 + 2,000 $\times$ 0.2 = 8,400, so the given manager is somewhat risk-averse and conservative.

Let pl_0 be the minimal value in our payoff PL and let pl^* be the maximal value. In our example PL matrix (Section 3.1), we would choose $pl^* = 500$ and $pl_0 = 0$. We decompose each $pl_{i,j}$ as $e_{i,j} = decomp(pl_0, pl_{i,j}, pl^*,)$. This decomposition creates an equivalent pair of payoffs $\{pl_0, pl^*\}$, with probability $e_{i,j}$ of getting the more desirable pl^*. We call the matrix formed by these elements $e_{i,j}$'s the equilibrium matrix E.

Any element $e_{i,j}$ will satisfy the following inequality:

$$pl_0 \times (1 - e_{i,j}) + pl^* \times e_{i,j} \geq pl_{i,j}$$

The difference between the two sides of this equation reflects the manager's degree of risk averseness. If the two sides are equal, risk analysis reduces to the expected value.

3.4. Value of Prototyping

Given the various unknowns in the states of nature, the software manager may choose to get more information with a prototype so that a better final decision can be made. However, before undertaking the procedure to extract more information, one should be sure that the gain due to the information will outweigh the cost of obtaining it. Here, we try to establish an absolute boundary: What is the value of perfect information?

The best we can expect is that the results of the experiment will indicate for sure which state of nature will hold. Under this case, we can choose the alternative that gives the highest performance level under the given state of nature:

$$= \sum_j p_j \times \max_i pl_{i,j}$$

In our example, we would choose X^1 under st_2 and choose X^2 otherwise, resulting in performance level :

$$\begin{aligned} &= 0.3 \times 300 + 0.5 \times 500 + 0.2 \times 200 \\ &= 380 \end{aligned}$$

What is the value of this perfect information? Since the expected value of our performance level was computed previously as 280, the value of this information is an improvement in performance level of $380 - 280 = 100$. This is the most that we can expect our prototype to achieve and still be cost effective.

Assume we build a prototype to test which state of nature will be true. While we would like an exact answer, since a prototype is only an approximation to the real system, the results from prototyping are probabilistic. Let $result_1$, $result_2$, ... $result_k$ be the possible results of the prototype. This information will be presented in a conditional probability matrix C where $c_{i,j}$ represents the conditional probability of result $result_i$ given state of nature S_j.

Given the probabilities for each state (vector P) and the conditional probability matrix C, the marginal probability distribution (vector Q) for obtaining $result_i$ is given by:

$$q_i = \sum_j c_{i,j} \times p_j$$

We can compute the a posteriori distribution matrix P'. P' has as many rows as results from the prototype which are updated values of vector P. Row i gives the probabilities of the states of nature given that the result of the prototype is $result_i$:

$$p'_{i,j} = \frac{c_{i,j} \times p_j}{q_i}$$

Following our example, assume that a prototype of alternative X^2 is planned. The planned prototype can give the following results:

1. $result_1$: We are satisfied with the system as presented by prototype.
2. $result_2$: We are not satisfied.

Assume that the conditional probabilities are estimated beforehand. For example, we estimate that if the state of nature is st_2, we have probabilities 0.3 and 0.7 to obtain results $result_1$ and $result_2$ respectively from the prototype. The conditional probabilities appear in the matrix C having a column for each state and a row for each result of the prototype:

$$C = \begin{bmatrix} 0.9 & 0.3 & 0.4 \\ 0.1 & 0.7 & 0.6 \end{bmatrix}$$

From this we can calculate the probability q_i for each result i of the prototype, giving $Q = (0.5, 0.5)$, and the a posteriori distribution matrix:

$$P' = \begin{bmatrix} 0.54 & 0.30 & 0.16 \\ 0.06 & 0.70 & 0.24 \end{bmatrix}$$

If, for example, we get $result_1$ from the prototyping study, the new expected values for alternatives X^1 and X^2 are:

$$\begin{aligned} v_1 &= 100 \times 0.54 + 500 \times 0.3 + 0 \times 0.16 \\ &= 204 \\ v_2 &= 300 \times 0.54 + 200 \times 0.3 + 200 \times 0.16 \\ &= 254 \end{aligned}$$

In this case, alternative X^2 should be chosen, since it gives the higher performance level.

Similarly, if we should get $result_2$ from the prototyping study, the new performance levels for alternatives X^1 and X^2 are:

$$\begin{aligned} v_1 &= 100 \times 0.06 + 500 \times 0.7 + 0 \times 0.16 \\ &= 356 \\ v_2 &= 300 \times 0.06 + 200 \times 0.7 + 200 \times 0.24 \\ &= 206 \end{aligned}$$

In this case, alternative X^1 is the preferred choice.

Given that our expected performance level with no information was 280 (Section 3.2), we should only prototype if we gain from prototyping:

$$\begin{aligned} v_P &= 0.5 \times 356 + 0.5 \times 254 - 280 \\ &= 25 \end{aligned}$$

Since there is a positive gain $v_P = 25$, prototyping should be carried out as long as the cost to construct the prototyping study is less than this. Otherwise, an immediate decision should be made.

4. SUGGESTED READINGS

The Woodcock and Loomes paper describes a notation similar to Z.

- J. Woodcock and M. Loomes, *Software Engineering Mathematics*, Addison-Wesley, Reading, MA, 1988.
- C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

• M. I. Jackson, "Developing Ada programs using the Vienna Development Method (VDM)," *Software Practice and Experience*, Vol. 15, No. 3, 1985, pp. 305-318.

• A. Hall, "Seven myths of formal methods," *IEEE Software*, Vol. 7, No. 5, 1990, pp. 11-19.

• J. Wing, "A specifier's introduction to formal methods," *IEEE Computer*, vol. 23, No. 9, 1990, pp. 8-24.

Temporal logic is covered in greater detail in:

• L. Lamport, "'Sometime' is sometimes 'not never,:' On the temporal logic of programs," *Proceedings of the 7th ACM Principles of Programming Language Conference*, Las Vegas, NV, 1980, pp. 174-185.

• S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Transactions on Programming Languages and Systems* Vol. 4, No. 3, 1982, pp. 455-495.

The section on risk analysis is described in:

• S. Cárdenas and M. V. Zelkowitz, "Evaluation criteria for functional specifications," *Proceedings of the ACM/IEEE 12th International Conference on Software Engineering*, Nice, France, March 1990, pp. 26-33.

• S. Cárdenas, and M. V. Zelkowitz, "A management tool for evaluation of software designs," *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, , 1991.

Bibliography

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

[2]S. K. Basu and R. T. Yeh, "Strong Verification of Programs," *IEEE Transactions on Software Engineering*, Vol. 1, No. 3, 1975, pp. 339-346.

[3]J. Bentley, *Programming Pearls*, Addison-Wesley, Reading, MA, 1986.

[4]R. S. Boyer and J. S. Moore, "Proving Theorems about LISP Functions," *Journal of the ACM*, Vol. 22, 1975, pp. 129-144.

[5]R. Burstall, "Proving Properties of Program by Structural Induction," *Computer Journal*, Vol. 12, 1969, pp. 41-48.

[6]S. Cárdenas and M. V. Zelkowitz, "Evaluation criteria for functional specifications," *Proceedings of the ACM/IEEE 12th International Conference on Software Engeering*, Nice, France, 1990, pp. 26-33.

[7]S. Cárdenas and M. V. Zelkowitz, "A management tool for evaluation of software designs," *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, 1991.

[8]S. A. Cook, "Soundness and Completeness of an Axiom System for Program Verification," *SIAM Journal of Computing*, Vol. 7, 1978, pp. 70-90.

[9]D. Craigen (Ed.), *Formal methods for trustworthy computer systems (FM89)*, Springer Verlag, New York, 1990.

[10]E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, Vol. 18, No. 8, 1975, pp. 453-458.

[11]E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[12]D. D. Dunlop and V.R. Basili, "A Comparative Analysis of Functional Correctness," *ACM Computing Surveys*, Vol. 14, No. 2, 1982, pp. 229-244.

[13]D. D. Dunlop and V.R. Basili, "A Heuristic for Deriving Loop Functions," *IEEE Transactions on Software Engineering*, Vol. 10, 1984, pp. 275-285.

[14]C. N. Fischer and R. J. LeBlanc, *Crafting a compiler*, Benjamin Cummings, Menlo Park, CA, 1988.

[15]L. Flon and J. Misra, "A Unified Approach to the Specification and Verification of Abstract Data Types," *Proceedings of the IEEE Conference on Specifications of Reliable Software*, Cambridge, MA,, 1979, pp. 162-169.

[16]R. W. Floyd, "Assigning Meanings to Programs," *Symposium in Applied Mathematics*, 19, 1967, pp. 19-32.

[17]R. Forgaard and J. V. Guttag, "REVE: A term rewriting system generator with failure resistant Knuth-Bendix," *Workshop on the Rewrite Rule Laboratory*, National Science Foundation, 1984, pp.5-31.

[18]J. D. Gannon, R. B. Hamlet, and H. D. Mills, "Theory of Modules," *Transactions on Software Engineering*, Vol. 13, No. 7, 1987, pp. 820-829.

[19]S. Gerhart, "An Overview on AFFIRM: a Specification and Verification System," in *Information Processing*, Vol. 80 (S.H. Lavington (Ed)), North Holland, 1980, pp. 343-387.

[20]J. A. Goguen, J.W. Thatcher, and E.G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in *Current Trends in Programming Methodology*, Vol. 4, (R. T. Yeh (Ed)), Prentice-Hall, Englewood Cliff, NJ, 1978, pp. 80-149.

[21]J. A. Goguen, "How to Prove Algebraic Inductive Hypotheses without Induction," *LNSC*, Vol. 87, Springer-Verlag, New York, 1980, pp 356-373.

[22]D. Gries and G. Levin, "Assignment and Procedure Call Proof Rules," *Transactions on Programming Languages and Systems*, Vol. 2, No. 4, 1980, pp. 564-579.

[23]D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.

[24]J. V. Guttag and J.J. Horning, “The Algebraic Specification of Abstract Data Types,” *Acta Informatica*, Vol. 10, 1978, pp. 27-52.

[25]J. V. Guttag, E. Horowitz, and D. Musser, “Abstract Data Types and Software Validation,” *Communications of the ACM*, Vol. 21, 1978, pp. 1048-1064.

[26]J. V. Guttag, E. Horowitz, and D. R. Musser, “The Design of Data Type Specifications,” in *Current Trends in Programming Methodology 4: Data Structuring*, (R. T. Yeh (Ed)) Prentice-Hall, Englerwood Cliffs, NJ, 1978, pp. 60-79.

[27]J. V. Guttag, “Notes on Type Abstraction (Version 2),” *IEEE Transactions on Software Engineering*, Vol. 6, No. 1, 1980, pp. 13-23.

[28]A. Hall. “Seven myths of formal methods,” *IEEE Software* Vol. 7, No. 5, 1990, pp. 11-19.

[29]C. A. R. Hoare, “An Axiomatic Basic for Computer Programming,” *Communications of the ACM*, Vol. 12, No. 10, 1969, pp. 576-580, 583.

[30]C. A. R. Hoare, “Procedures and Parameters, An Axiomatic Approach,” *Symposium on the Semantics of Algorithmic Languages*, Springer-Verlag, New York, 1971, pp. 102-116.

[31]C. A. R. Hoare, and N. Wirth, “An Axiomatic Definition of the Programming Language PASCAL,” *Acta Informatica*, Vol. 2, 1973, pp. 335-355.

[32]G. Huet, “Confluent Reductions: Abstract Properties and Applications to Term-Rewriting Systems,” *Journal of the ACM*, Vol. 27, 1980, pp. 797-821.

[33]G. Huet and J.-M. Hullot, “Proofs by Induction in Equational Theories with Constructors,” *Journal of Computer and System Science*, Vol. 25, 1982, pp. 239-266.

[34]J-M. Hullot, “Canonical Forms and Unification,” *LNSC*, Vol. 87, Springer-Verlag, New York, 1980, pp. 318-334.

[35]C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[36]M. I. Jackson, “Developing Ada programs using the Vienna Development Method (VDM),” *Software Practice and Experience*, vo. 15, No. 3, 1985, pp. 305-318.

[37]S. Kamin, “The Expressive Theory of Stacks,” *Acta Informatica*, Vol. 24, 1987, pp. 695-709.

[38]D. E. Knuth and P.B. Bendix, “Simple Word Problems in Universal Algebras,” in *Computational Problems in Abstract Algebras,* J. Leech (ed)), Pergamon Press, New York, 1970, pp. 263-297.

[39]L. Lamport, “‘Sometime’ is sometimes ‘not never,:’ On the temporal logic of programs,” *Proceedings of the 7th ACM Prin. of Programming Languages Conference*, Las Vegas, NV, 1980, pp. 174-185.

[40]L. Lamport and F. B. Schneider, “The ‘Hoare Logic’ of CSP, and All That,” *Transactions on Programming Languages and Systems*, Vol. 6, No. 2, 1984, pp. 281-296.

[41]R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.

[42]B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*, MIT Press, McGraw-Hill, New York, NY, 1986.

[43]D. C. Luckham and N. Suzuki. “Verification of Array, Record, and Pointer Operations in Pascal,” *Trans on Programming Languages and Systems*, Vol. 1, No. 2, 1979, pp. 226-244.

[44]H. D. Mills, “The New Math of Computer Programming,” *Communications of the ACM*, Vol. 18, No. 1, 1975, pp. 43-48.

[45]H. D. Mills, V. R. Basili, J. D. Gannon and R. G. Hamlet. *Principles of Computer Programming: A Mathematical Approach*, William C. Brown, Dubuque, IA, 1987.

[46]D. R. Musser, “Abstract data type specifications in the AFFIRM system,” *IEEE Specifications of Reliable Software Conference*, Cambridge MA, 1979, pp. 47-57.

[47]D. R. Musser, “On Proving Inductive Properties of Abstract Data Types,” Proceedings of the 7th ACM Symposium on Principles of Programming Languages, 1980, pp. 154-162.

[48]S. Owicki and L. Lamport, “Proving liveness properties of concurrent programs,” *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, 1982, pp. 455-495.

[49]G. L. Peterson and M.E. Stickel, "Complete Sets of Reductions for Some Equational Theories," *Journal of the ACM*, Vol. 28, 1981, pp. 233-264.

[50]N. C. K. Phillips, "Safe Data Type Specification," *IEEE Software*, Vol. 10, 1984, pp. 285-289.

[51]T. W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[52]J. H. Remmers, "A Technique for Developing Loop Invariants," *Information Processing Letters*, Vol. 18, 1984, pp. 137-139.

[53]J. C. Reynolds, *The Craft of Computer Programming*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[54]R. Sethi, *Programming Languages: Concepts and Constructs*, Addison-Wesley, reading, MA, 1989.

[55]J. Spitzen and B. Wegbreit, "The Verification and Synthesis of Data Structures," *Acta Informatica*, Vol. 4, 1975, pp. 127-144.

[56]D. A. Schmidt, *Denotational Semantics*, Allyn and Bacon. New York, 1986.

[57]R. D. Tennent, "The denotational semantics of programming languages," *Communications of the ACM*, Vol. 19, No. 8, 1976, pp. 437-453.

[58]P. Wegner, *Programming Languages, Information Structures and Machine organization*, McGraw Hill, New York, 1968, pp. 180-196.

[59]J. Wing, "A specifier's introduction to formal methods," *IEEE Computer*, Vol. 23, No. 9, 1990.

[60]J. Woodcock and M. Loomes, *Software Engineering Mathematics*, Addison-Wesley, Reading, MA, 1988.

[61]W. Wulf, M. Shaw, L. Flon, and P. Hilfinger, *Fundamental structures of Computer Science*, Addison-Wesley, Reading, MA, 1980.

[62]W. Wulf, R. L. London and M. Shaw. "An introduction to the construction and verification of Alphard programs," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, 1976, pp. 253-265.

[63]M. V. Zelkowitz, "A functional correctness model of program verification," *IEEE Computer*, Vol. 23, No. 11, 1990, pp. 30-39.

[64]M. V.Zelkowitz, "The role of verification in the software specification process," in *Advances in Computer*, Vol. 36 (M. Yovits (ed)), Academic Press, Orlando, FL, 1993.

Author Index

Aho A., 28

Backus J., 5
Basili V., 117
Basu S. K., 134
Bendix P., 169
Bentley J., 28
Bjorner D., 214

Càrdenas S., 236
Church A., 191
Craigen D., 29

Dijkstra E. W., 6, 9, 10, 119, 134
Dunlop D., 117

Fischer C., 28
Flon L., 189
Floyd R., 7, 9, 84

Gannon J. D., 117
Gries D., 10, 28, 84, 134
Guttag J., 10, 189

Hall A., 16, 29, 235
Hamlet R., 117
Hoare C. A. R., 9, 33, 37, 84
Hopper G., 5
Horning J., 152, 189
Horowitz E., 189
Huet G., 171, 190
Hullot J., 190

Jackson M., 235
Jones C., 214, 235

Knuth D., 169

Lamport L., 84, 228, 235
LeBlanc R., 28
Levin G., 84
Linger R., 117
Loomis M., 29, 235
Luckham D., 84

McCarthy J., 90
Mills H. D., 13, 14, 117
Misra J., 189
Musser D., 174, 189

Naur P., 5

Oppen D., 171
Owicki S., 228, 235

Pratt T., 29

Remmers J., 84
Reynolds J., 84

Schmidt D., 211
Schneider F., 84
Scott D., 14
Sethi R., 28, 29
Strachey C., 14
Straub P., 145
Suzuki N., 84

Tennent R., 211

Ullman J., 28

Wegner P., 211
Wing J., 29, 235
Wirth N., 84
Witt B. I., 117
Woodcock J., 29, 235

Yeh R., 134, 189

Zelkowitz M. V., 1, 117, 236

Index

Ada
- Data storage, 11
- Package, 108

Adaptation, 62, 71
- Existential quantifiers, 64
- Quantification, 63

ADT (Abstract data type), *see* Algebraic specifications

Algebraic specifications, 10, 135, 139
- Axioms, 140
- Consistency, 149
- Constructors, 139, 145
- Data type, 136
- Data type induction, 150
- Developing axioms, 142
- Equality, 150
- Hints for writing axioms, 145
- Rewrite rules, 141
- Signatures, 145
- Verification, 158
- Verifying implementations, 154

Applicative languages
- Data storage, 12

Array assignment, 48

Atomicity, 226

Attribute grammars, 7

Axiomatic verification, 7, 9
- Adaptation, 62, 71
- Array assignment, 48
- Axioms, 31
- Composition axiom, 31
- Conditional, 33
- Consequence axiom, 31
- If, 33
- Invariant, 34
- Postcondition, 8
- Precondition, 8
- Procedure call substitution, 59
- Procedure calls, 57
- Procedure invocation, 58
- Recursion, 72
- Termination, 38, 41
- While, 33

Bottom, 196

Box notation, 13, 85

Bug, *see* Software error

C
- Class, 108
- Data storage, 11

Completeness, 20

Composition axiom, 31

Concurrent assignment, 90

Concurrent execution, 225

Conditional axiom, 33

Conditional functions, 22

Conditional probability, 233

Consequence axiom, 31

Continuations, 208

Continuity, 198

Continuous functions, 197

Correctness, 2
- Weakest precondition, 122

Critical pairs, 173

Data abstraction, 107
Data abstractions, 135
    Encapsulation, 135
Data storage models, 11
Data type induction, 150
    Inductionless induction, 169
Datatypes, 195
    Axioms, 195
Deadlock, 221
Decidable, 19
Decisions
    Certainty, 229
    Uncertainty, 230
Denotational semantics, 14, 191
    Aliasing, 207
    Assignment statement, 206, 208
    Begin statement, 206, 207
    Bottom, 196
    Composition, 206, 207
    Continuity, 198
    Continuous functions, 197
    Datatypes, 195
    Denotable values, 203
    Environment, 207
    Expressions, 204
    Factorial function, 199
    Fixed point, 200
    If statement, 206, 208
    Injection, 206
    Inspection, 205
    Pointers, 207
    Program semantics, 201
    Program statement, 203
    Projection, 206
    Recursive functions, 199
    Serious computation, 209
    State, 14
    Top, 196
    While statement, 206, 208
Domains, 196
    Simple, 197
Encapsulation, 135
Equilibrium probability, 232
Example
    Array reversal, 166
    Data type induction, 153, 178
    Fast multiplication, 42
    Finite integers, 147
    Functional design, 103
    Functional verification, 98
    Inference system, 26
    Integer division, 34, 128
    Knuth–Bendix, 177
    Multiplication, 38, 104, 129, 218
    Reversing an array, 53
    Shifting an array, 50
    Simple recusion, 73
    Simple sums, 66
    Stack verification, 160

Fixed point, 200
Formal methods
    Limitations, 16
FORTRAN
    Data storage, 11
Functional correctness, 13, 15, 85
    Assignment statement, 90, 93
    Begin statement, 93
    Box notation, 13, 85
    Concurrent assignment, 90
    Conditional statement, 91
    Correctness theorem, 85
    Design rules, 90
    If statement, 94
    Semantics, 86
    State, 86
    Statement execution, 87
    Statements, 87
    Symbolic execution, 88
    Termination, 97

- Trace table, 89
- Verification conditions, 92
- While loop design, 96
- While statement, 94

Guarded commands, 10, 119
- If statement, 120
- Repetitive statement, 120

Hoare axioms, *see* Axiomatic verification

If axiom, 33
Imperative languages
- Data storage, 11

Inductionless induction, 169
Inference rules, 19
- Modus ponens, 21
- Substitution, 21

Initial value $(\leftharpoonup)$, 22
Injection, 206
Inspection, 205
Invariant, 34
- Chossing, 47

Iteration theorem, 125

Knuth–Bendix algorithm, 169, 175
- Critical pairs, 173

Lambda calculus, 191
- Boolean values, 193
- Church–Rosser, 193
- Integers, 194

Lattice, 196
LISP
- Data storage, 12

Liveness, 221

Mills correctness, *see* Functional correctness
Modus ponens, 21
Mutual exclusion, 221

Notation
- Array access α, 49
- Bottom $\perp$, 196
- Box function $\sqcap$, 13
- Concurrent assignment $:=$, 90
- Conditional assignment $(\rightarrow)$, $(\rightarrow)$91
- Defines $\triangleq$, 21
- Denotational expressions $\{\!\!\{\}\!\!\}$, 204
- Eventually $\rightarrow$, 222
- Guard $[\!]$, 10
- Lambda expression λ, 191
- Leads to $\leadsto$, 223
- Not never $\diamond$, 223
- Now and forever $\square$, 222
- Original value $\leftharpoonup$, 22
- Pre-Postconditions $\{\}S\{\}$, 9
- Top $\top$, 196
- Weakest precondition wp(), 10

Operational semantics, 13, 137
Operational specifications
- Verification, 154

Ordering equations, *see* Knuth–Bendix algorithm

Pascal
- Data storage, 11

Payoff matrix, 232
Performance level, 229
Postcondition, 8
Precondition, 8
Predicate calculus, 17, 22
- Completeness, 20
- Decidable, 19
- Inference rules, 19
- Interpretation, 18, 24
- Quantifiers, 25
- Satisfiable, 19
- Substitution, 25

Truth assignment, 18
Truth table, 19
Valid, 19
Well-formed formula, 23
Predicate transformers, 10, 119
Composition axiom, 122
Do statement axiom, 125
Guarded commands, 10, 119
If statement axiom, 122
Iteration theorem, 125
Preedicate transformers
Assignment axiom, 122
Procedure call inference rules, 57
Procedure call invocation, 58
Procedure call substitution, 59
Projection, 206
Propositional calculus, *see* Predicate calculus
Prototyping, 232

Quantifiers, 25

Recursion, 72
Representation functions, 108
Rewrite rules, 141
Risk analysis, 228
Alternatives, 231
Conditional probability, 233
Payoff matrix, 232
Risk aversion, 232

Safety, 221
Satisfiable, 19
Scott–Strachey semantics, *see* Denotational semantics
Semantics, 15
Serious computation, 209
Software error, 5
Software failures, 1
Specifications, 2
Substitution, 21, 25
Symbolic execution, 88
Syntax, 5

Temporal logic, 221
Assignment, 227
Atomic statement, 228
Atomicity, 226
Branching time, 223
Cobegin, 228
Concatenation, 228
Concurrent execution, 225
Deadlock, 221
Exit, 227
Formal properties, 225
General statement, 228
Immediate assertions, 224
Inference rules, 227
Linear time, 223
Liveness, 221
Mutual exclusion, 221
Non–primitive operators, 223
Primitive operators, 222
Safety, 221
States, 224
While, 227
Termination, 38, 41
Testing, 6
Top, 196
Trace table, 89
Truth table, 19

Unification, 170

Valid, 19
VDL, 13
VDM, *see* Vienna Development Method
Verification, 7, 15
Vienna Definition Language, 13
Vienna Development Method, 213
Data, 216
Functions, 215
Logic, 214
Postcondition, 217

Precondition, 217
Specifications, 216

Weakest precondition, *see* Predicate transformers
Weakest preconditions, 122
Axioms, 122
Correctness, 122
Weighing words, *see* Knuth–Bendix algorithm
Well-formed formula, 23
While axiom, 33