a Practical Theory of Programming

2012-3-30 edition

Eric C.R. Hehner

a Practical Theory of Programming

2012-3-30 edition

Eric C.R. Hehner

Department of Computer Science
University of Toronto
Toronto ON M5S 2E4
Canada

The first edition of this book was published by
Springer-Verlag Publishers
New York
1993
ISBN 0-387-94106-1
QA76.6.H428

The current edition is available free at

www.cs.utoronto.ca/~hehner/aPToP

You may copy freely as long as you
include all the information on this page.

Contents

0 Preface 0
- 0.0 Introduction 0
- 0.1 Current Edition 1
- 0.2 Quick Tour 1
- 0.3 Acknowledgements 2

1 Basic Theories 3
- 1.0 Boolean Theory 3
  - 1.0.0 Axioms and Proof Rules 5
  - 1.0.1 Expression and Proof Format 7
  - 1.0.2 Monotonicity and Antimonotonicity 9
  - 1.0.3 Context 10
  - 1.0.4 Formalization 12
- 1.1 Number Theory 12
- 1.2 Character Theory 13

2 Basic Data Structures 14
- 2.0 Bunch Theory 14
- 2.1 Set Theory (optional) 17
- 2.2 String Theory 17
- 2.3 List Theory 20
  - 2.3.0 Multidimensional Structures 22

3 Function Theory 23
- 3.0 Functions 23
  - 3.0.0 Abbreviated Function Notations 25
  - 3.0.1 Scope and Substitution 25
- 3.1 Quantifiers 26
- 3.2 Function Fine Points (optional) 29
  - 3.2.0 Function Inclusion and Equality (optional) 30
  - 3.2.1 Higher-Order Functions (optional) 30
  - 3.2.2 Function Composition (optional) 31
- 3.3 List as Function 32
- 3.4 Limits and Reals (optional) 33

4 Program Theory 34
- 4.0 Specifications 34
  - 4.0.0 Specification Notations 36
  - 4.0.1 Specification Laws 37
  - 4.0.2 Refinement 39
  - 4.0.3 Conditions (optional) 40
  - 4.0.4 Programs 41
- 4.1 Program Development 43
  - 4.1.0 Refinement Laws 43
  - 4.1.1 List Summation 43
  - 4.1.2 Binary Exponentiation 45

4.2 Time 46
4.2.0 Real Time 46
4.2.1 Recursive Time 48
4.2.2 Termination 50
4.2.3 Soundness and Completeness (optional) 51
4.2.4 Linear Search 51
4.2.5 Binary Search 53
4.2.6 Fast Exponentiation 57
4.2.7 Fibonacci Numbers 59
4.3 Space 61
4.3.0 Maximum Space 63
4.3.1 Average Space 64

5 Programming Language 66
5.0 Scope 66
5.0.0 Variable Declaration 66
5.0.1 Variable Suspension 67
5.1 Data Structures 68
5.1.0 Array 68
5.1.1 Record 69
5.2 Control Structures 69
5.2.0 While Loop 69
5.2.1 Loop with Exit 71
5.2.2 Two-Dimensional Search 72
5.2.3 For Loop 74
5.2.4 Go To 76
5.3 Time and Space Dependence 76
5.4 Assertions (optional) 77
5.4.0 Checking 77
5.4.1 Backtracking 77
5.5 Subprograms 78
5.5.0 Result Expression 78
5.5.1 Function 79
5.5.2 Procedure 80
5.6 Alias (optional) 81
5.7 Probabilistic Programming (optional) 82
5.7.0 Random Number Generators 84
5.7.1 Information (optional) 87
5.8 Functional Programming (optional) 88
5.8.0 Function Refinement 89

6 Recursive Definition 91
6.0 Recursive Data Definition 91
6.0.0 Construction and Induction 91
6.0.1 Least Fixed-Points 94
6.0.2 Recursive Data Construction 95
6.1 Recursive Program Definition 97
6.1.0 Recursive Program Construction 98
6.1.1 Loop Definition 99

7 Theory Design and Implementation 100
7.0 Data Theories 100
7.0.0 Data-Stack Theory 100
7.0.1 Data-Stack Implementation 101
7.0.2 Simple Data-Stack Theory 102
7.0.3 Data-Queue Theory 103
7.0.4 Data-Tree Theory 104
7.0.5 Data-Tree Implementation 104
7.1 Program Theories 106
7.1.0 Program-Stack Theory 106
7.1.1 Program-Stack Implementation 106
7.1.2 Fancy Program-Stack Theory 107
7.1.3 Weak Program-Stack Theory 107
7.1.4 Program-Queue Theory 108
7.1.5 Program-Tree Theory 108
7.2 Data Transformation 109
7.2.0 Security Switch 111
7.2.1 Take a Number 112
7.2.2 Parsing 113
7.2.3 Limited Queue 115
7.2.4 Soundness and Completeness (optional) 117

8 Concurrency 118
8.0 Independent Composition 118
8.0.0 Laws of Independent Composition 120
8.0.1 List Concurrency 120
8.1 Sequential to Parallel Transformation 121
8.1.0 Buffer 122
8.1.1 Insertion Sort 123
8.1.2 Dining Philosophers 124

9 Interaction 126
9.0 Interactive Variables 126
9.0.0 Thermostat 128
9.0.1 Space 129
9.1 Communication 131
9.1.0 Implementability 132
9.1.1 Input and Output 133
9.1.2 Communication Timing 134
9.1.3 Recursive Communication (optional) 134
9.1.4 Merge 135
9.1.5 Monitor 136
9.1.6 Reaction Controller 137
9.1.7 Channel Declaration 138
9.1.8 Deadlock 139
9.1.9 Broadcast 140

10 Exercises 147
10.0 Preface 147
10.1 Basic Theories 147
10.2 Basic Data Structures 154
10.3 Function Theory 156
10.4 Program Theory 161
10.5 Programming Language 177
10.6 Recursive Definition 181
10.7 Theory Design and Implementation 187
10.8 Concurrency 193
10.9 Interaction 195

11 Reference 201
11.0 Justifications 201
11.0.0 Notation 201
11.0.1 Basic Theories 201
11.0.2 Basic Data Structures 202
11.0.3 Function Theory 204
11.0.4 Program Theory 204
11.0.5 Programming Language 206
11.0.6 Recursive Definition 207
11.0.7 Theory Design and Implementation 207
11.0.8 Concurrency 208
11.0.9 Interaction 208
11.1 Sources 209
11.2 Bibliography 211
11.3 Index 215
11.4 Laws 223
11.4.0 Booleans 223
11.4.1 Generic 225
11.4.2 Numbers 225
11.4.3 Bunches 226
11.4.4 Sets 227
11.4.5 Strings 227
11.4.6 Lists 228
11.4.7 Functions 228
11.4.8 Quantifiers 229
11.4.9 Limits 231
11.4.10 Specifications and Programs 231
11.4.11 Substitution 232
11.4.12 Conditions 232
11.4.13 Refinement 232
11.5 Names 233
11.6 Symbols 234
11.7 Precedence 235
11.8 Distribution 235

End of Contents

0 Preface

0.0 Introduction

What good is a theory of programming? Who wants one? Thousands of programmers program every day without any theory. Why should they bother to learn one? The answer is the same as for any other theory. For example, why should anyone learn a theory of motion? You can move around perfectly well without one. You can throw a ball without one. Yet we think it important enough to teach a theory of motion in high school.

One answer is that a mathematical theory gives a much greater degree of precision by providing a method of calculation. It is unlikely that we could send a rocket to Jupiter without a mathematical theory of motion. And even baseball pitchers are finding that their pitch can be improved by hiring an expert who knows some theory. Similarly a lot of mundane programming can be done without the aid of a theory, but the more difficult programming is very unlikely to be done correctly without a good theory. The software industry has an overwhelming experience of buggy programs to support that statement. And even mundane programming can be improved by the use of a theory.

Another answer is that a theory provides a kind of understanding. Our ability to control and predict motion changes from an art to a science when we learn a mathematical theory. Similarly programming changes from an art to a science when we learn to understand programs in the same way we understand mathematical theorems. With a scientific outlook, we change our view of the world. We attribute less to spirits or chance, and increase our understanding of what is possible and what is not. It is a valuable part of education for anyone.

Professional engineering maintains its high reputation in our society by insisting that, to be a professional engineer, one must know and apply the relevant theories. A civil engineer must know and apply the theories of geometry and material stress. An electrical engineer must know and apply electromagnetic theory. Software engineers, to be worthy of the name, must know and apply a theory of programming.

The subject of this book sometimes goes by the name “programming methodology”, “science of programming”, “logic of programming”, “theory of programming”, “formal methods of program development”, or “verification”. It concerns those aspects of programming that are amenable to mathematical proof. A good theory helps us to write precise specifications, and to design programs whose executions provably satisfy the specifications. We will be considering the state of a computation, the time of a computation, the memory space required by a computation, and the interactions with a computation. There are other important aspects of software design and production that are not touched by this book: the management of people, the user interface, documentation, and testing.

The first usable theory of programming, often called “Hoare's Logic”, is still probably the most widely known. In it, a specification is a pair of predicates: a precondition and postcondition (these and all technical terms will be defined in due course). A closely related theory uses Dijkstra's weakest precondition predicate transformer, which is a function from programs and postconditions to preconditions, further advanced in Back's Refinement Calculus. Jones's Vienna Development Method has been used to advantage in some industries; in it, a specification is a pair of predicates (as in Hoare's Logic), but the second predicate is a relation. There are theories that specialize in real-time programming, some in probabilistic programming, some in interactive programming.

The theory in this book is simpler than any of those just mentioned. In it, a specification is just a boolean expression. Refinement is just ordinary implication. This theory is also more comprehensive than those just mentioned, applying to both terminating and nonterminating computation, to both sequential and parallel computation, to both stand-alone and interactive computation. All at the same time, we can have variables whose initial and final values are all that is of interest, variables whose values are continuously of interest, variables whose values are known only probabilistically, and variables that account for time and space. They all fit together in one theory whose basis is the standard scientific practice of writing a specification as a boolean expression whose (nonlocal) variables represent whatever is considered to be of interest.

There is an approach to program proving that exhaustively tests all inputs, called model-checking. Its advantage over the theory in this book is that it is fully automated. With a clever representation of boolean expressions (see Exercise 6), model-checking currently boasts that it can explore up to about 10^{60} states. That is something like the number of atoms in our galaxy! It is an impressive number until we realize that 10^{60} is about 2^{200}, which means we are talking about 200 bits. That is the state space of six 32-bit variables. To use model-checking on any program with more than six variables requires abstraction, and each abstraction requires proof that it preserves the properties of interest. These abstractions and proofs are not automatic. To be practical, model-checking must be joined with other methods of proving, such as those in this book.

The emphasis throughout this book is on program development with proof at each step, rather than on proof after development.

End of Introduction

0.1 Current Edition

Since the first edition of this book, new material has been added on space bounds, and on probabilistic programming. The **for**-loop rule has been generalized. The treatment of concurrency has been simplified. And for cooperation between parallel processes, there is now a choice: communication (as in the first edition), and interactive variables, which are the formally tractable version of shared memory. Explanations have been improved throughout the book, and more worked examples have been added.

As well as additions, there have been deletions. Any material that was usually skipped in a course has been removed to keep the book short. It's really only 147 pages; after that is just exercises and reference material.

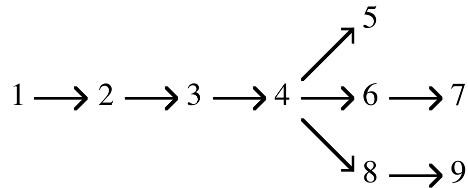
Lecture visuals and solutions to exercises are available to course instructors from the author.

End of Current Edition

0.2 Quick Tour

All technical terms used in this book are explained in this book. Each new term that you should learn is underlined. As much as possible, the terminology is descriptive rather than honorary (notable exception: “boolean”). There are no abbreviations, acronyms, or other obscurities of language to annoy you. No specific previous mathematical knowledge or programming experience is assumed. However, the preparatory material on booleans, numbers, lists, and functions in Chapters 1, 2, and 3 is brief, and previous exposure might be helpful.

The following chart shows the dependence of each chapter on previous chapters.



Chapter 4, Program Theory, is the heart of the book. After that, chapters may be selected or omitted according to interest and the chart. The only deviations from the chart are that Chapter 9 uses variable declaration presented in Subsection 5.0.0, and small optional Subsection 9.1.3 depends on Chapter 6. Within each chapter, sections and subsections marked as optional can be omitted without much harm to the following material.

Chapter 10 consists entirely of exercises grouped according to the chapter in which the necessary theory is presented. All the exercises in the section “Program Theory” can be done according to the methods presented in Chapter 4; however, as new methods are presented in later chapters, those same exercises can be redone taking advantage of the later material.

At the back of the book, Chapter 11 contains reference material. Section 11.0, “Justifications”, answers questions about earlier chapters, such as: why was this presented that way? why was this presented at all? why wasn't something else presented instead? It may be of interest to teachers and researchers who already know enough theory of programming to ask such questions. It is probably not of interest to students who are meeting formal methods for the first time. If you find yourself asking such questions, don't hesitate to consult the justifications.

Chapter 11 also contains an index of terminology and a complete list of all laws used in the book. To a serious student of programming, these laws should become friends, on a first name basis. The final pages list all the notations used in the book. You are not expected to know these notations before reading the book; they are all explained as we come to them. You are welcome to invent new notations if you explain their use. Sometimes the choice of notation makes all the difference in our ability to solve a problem.

End of Quick Tour

0.3 Acknowledgements

For inspiration and guidance I thank Working Group 2.3 (Programming Methodology) of the International Federation for Information Processing, particularly Edsger Dijkstra, David Gries, Tony Hoare, Jim Horning, Cliff Jones, Bill McKeeman, Jay Misra, Carroll Morgan, Greg Nelson, John Reynolds, and Wlad Turski; I especially thank Doug McIlroy for encouragement. I thank my graduate students and teaching assistants from whom I have learned so much, especially Lorene Gupta, Peter Kanareitsev, Yannis Kassios, Victor Kwan, Albert Lai, Chris Lengauer, Andrew Malton, Lev Naiman, Theo Norvell, Rich Paige, Hugh Redelmeier, Alan Rosenthal, Anya Tafliovich, Justin Ward, and Robert Will. For their critical and helpful reading of the first draft I am most grateful to Wim Hesselink, Jim Horning, and Jan van de Snepscheut. For good ideas I thank Ralph Back, Wim Feijen, Netty van Gasteren, Nicolas Halbwachs, Gilles Kahn, Leslie Lamport, Alain Martin, Joe Morris, Martin Rem, Pierre-Yves Schobbens, Mary Shaw, Bob Tennent, and Jan Tijmen Udding. For reading the draft and suggesting improvements I thank Jules Desharnais, Andy Gravell, Ali Mili, Bernhard Möller, Helmut Partsch, Jørgen Steensgaard-Madsen, and Norbert Völker. I thank my classes for finding errors.

End of Acknowledgements

End of Preface

1 Basic Theories

1.0 Boolean Theory

Boolean Theory, also known as logic, was designed as an aid to reasoning, and we will use it to reason about computation. The expressions of Boolean Theory are called boolean expressions. We call some boolean expressions theorems, and others antitheorems.

The expressions of Boolean Theory can be used to represent statements about the world; the theorems represent true statements, and the antitheorems represent false statements. That is the original application of the theory, the one it was designed for, and the one that supplies most of the terminology. Another application for which Boolean Theory is perfectly suited is digital circuit design. In that application, boolean expressions represent circuits; theorems represent circuits with high voltage output, and antitheorems represent circuits with low voltage output. In general, Boolean Theory can be used for any application that has two values.

The two simplest boolean expressions are $\top$ and $\bot$. The first one, $\top$, is a theorem, and the second one, $\bot$, is an antitheorem. When Boolean Theory is being used for its original purpose, we pronounce $\top$ as "true" and $\bot$ as "false" because the former represents an arbitrary true statement and the latter represents an arbitrary false statement. When Boolean Theory is being used for digital circuit design, we pronounce $\top$ and $\bot$ as "high voltage" and "low voltage", or as "power" and "ground". Similarly we may choose words from other application areas. Or, to be independent of application, we may call them "top" and "bottom". They may also be called the zero-operand boolean operators because they have no operands.

There are four one-operand boolean operators, of which only one is interesting. Its symbol is $\neg$, pronounced "not". It is a prefix operator (placed before its operand). An expression of the form $\neg x$ is called a negation. If we negate a theorem we obtain an antitheorem; if we negate an antitheorem we obtain a theorem. This is depicted by the following truth table.

	$\top$	$\bot$
$\neg$	$\bot$	$\top$

Above the horizontal line, $\top$ means that the operand is a theorem, and $\bot$ means that the operand is an antitheorem. Below the horizontal line, $\top$ means that the result is a theorem, and $\bot$ means that the result is an antitheorem.

There are sixteen two-operand boolean operators. Mainly due to tradition, we will use only six of them, though they are not the only interesting ones. These operators are infix (placed between their operands). Here are the symbols and some pronunciations.

$\wedge$	"and"
$\vee$	"or"
$\Rightarrow$	"implies", "is equal to or stronger than"
$\Leftarrow$	"follows from", "is implied by", "is weaker than or equal to"
$=$	"equals", "if and only if"
$\neq$	"differs from", "is unequal to", "exclusive or", "boolean plus"

An expression of the form $x \wedge y$ is called a conjunction, and the operands x and y are called conjuncts. An expression of the form $x \vee y$ is called a disjunction, and the operands are called disjuncts. An expression of the form $x \Rightarrow y$ is called an implication, x is called the antecedent, and y is called the consequent. An expression of the form $x \Leftarrow y$ is also called an implication, but now

x is the consequent and y is the antecedent. An expression of the form $x=y$ is called an equation, and the operands are called the left side and the right side. An expression of the form $x \neq y$ is called an unequation, and again the operands are called the left side and the right side.

The following truth table shows how the classification of boolean expressions formed with two-operand operators can be obtained from the classification of the operands. Above the horizontal line, the pair ⊤⊤ means that both operands are theorems; the pair ⊤⊥ means that the left operand is a theorem and the right operand is an antitheorem; and so on. Below the horizontal line, ⊤ means that the result is a theorem, and ⊥ means that the result is an antitheorem.

	⊤⊤	⊤⊥	⊥⊤	⊥⊥
∧	⊤	⊥	⊥	⊥
∨	⊤	⊤	⊤	⊥
⇒	⊤	⊥	⊤	⊤
⇐	⊤	⊤	⊥	⊤
=	⊤	⊥	⊥	⊤
⧧	⊥	⊤	⊤	⊥

Infix operators make some expressions ambiguous. For example, $\bot \wedge \top \vee \top$ might be read as the conjunction $\bot \wedge \top$, which is an antitheorem, disjoined with $\top$, resulting in a theorem. Or it might be read as $\bot$ conjoined with the disjunction $\top \vee \top$, resulting in an antitheorem. To say which is meant, we can use parentheses: either $(\bot \wedge \top) \vee \top$ or $\bot \wedge (\top \vee \top)$. To prevent a clutter of parentheses, we employ a table of precedence levels, listed on the final page of the book. In the table, $\wedge$ can be found on level 9, and $\vee$ on level 10; that means, in the absence of parentheses, apply $\wedge$ before $\vee$. The example $\bot \wedge \top \vee \top$ is therefore a theorem.

Each of the operators $=$ $\Rightarrow$ $\Leftarrow$ appears twice in the precedence table. The large versions $=$ $\Longrightarrow$ $\Longleftarrow$ on level 16 are applied after all other operators. Except for precedence, the small versions and large versions of these operators are identical. Used with restraint, these duplicate operators can sometimes improve readability by reducing the parenthesis clutter still further. But a word of caution: a few well-chosen parentheses, even if they are unnecessary according to precedence, can help us see structure. Judgement is required.

There are 256 three-operand operators, of which we show only one. It is called conditional composition, and written **if** x **then** y **else** z. Here is its truth table.

	⊤⊤⊤	⊤⊤⊥	⊤⊥⊤	⊤⊥⊥	⊥⊤⊤	⊥⊤⊥	⊥⊥⊤	⊥⊥⊥
if then else	⊤	⊤	⊥	⊥	⊤	⊥	⊤	⊥

For every natural number n, there are 2^{2^n} operators of n operands, but we now have quite enough.

When we stated earlier that a conjunction is an expression of the form $x \wedge y$, we were using $x \wedge y$ to stand for all expressions obtained by replacing the variables x and y with arbitrary boolean expressions. For example, we might replace x with $(\bot \Rightarrow \neg(\bot \vee \top))$ and replace y with $(\bot \vee \top)$ to obtain the conjunction

$$(\bot \Rightarrow \neg(\bot \vee \top)) \wedge (\bot \vee \top)$$

Replacing a variable with an expression is called substitution or instantiation. With the understanding that variables are there to be replaced, we admit variables into our expressions, being careful of the following two points.

- We sometimes have to insert parentheses around expressions that are replacing variables in order to maintain the precedence of operators. In the example of the preceding paragraph, we replaced a conjunct x with an implication $\bot \Rightarrow \neg(\bot \vee \top)$; since conjunction comes before implication in the precedence table, we had to enclose the implication in parentheses. We also replaced a conjunct y with a disjunction $\bot \vee \top$, so we had to enclose the disjunction in parentheses.

- When the same variable occurs more than once in an expression, it must be replaced by the same expression at each occurrence. From $x \wedge x$ we can obtain $\top \wedge \top$, but not $\top \wedge \bot$. However, different variables may be replaced by the same or different expressions. From $x \wedge y$ we can obtain both $\top \wedge \top$ and $\top \wedge \bot$.

As we present other theories, we will introduce new boolean expressions that make use of the expressions of those theories, and classify the new boolean expressions. For example, when we present Number Theory we will introduce the number expressions $1+1$ and 2 , and the boolean expression $1+1=2$, and we will classify it as a theorem. We never intend to classify a boolean expression as both a theorem and an antitheorem. A statement about the world cannot be both true and (in the same sense) false; a circuit's output cannot be both high and low voltage. If, by accident, we do classify a boolean expression both ways, we have made a serious error. But it is perfectly legitimate to leave a boolean expression unclassified. For example, $1/0=5$ will be neither a theorem nor an antitheorem. An unclassified boolean expression may correspond to a statement whose truth or falsity we do not know or do not care about, or to a circuit whose output we cannot predict. A theory is called consistent if no boolean expression is both a theorem and an antitheorem, and inconsistent if some boolean expression is both a theorem and an antitheorem. A theory is called complete if every fully instantiated boolean expression is either a theorem or an antitheorem, and incomplete if some fully instantiated boolean expression is neither a theorem nor an antitheorem.

1.0.0 Axioms and Proof Rules

We present a theory by saying what its expressions are, and what its theorems and antitheorems are. The theorems and antitheorems are determined by the five rules of proof. We state the rules first, then discuss them after.

Axiom Rule — If a boolean expression is an axiom, then it is a theorem. If a boolean expression is an antiaxiom, then it is an antitheorem.

Evaluation Rule — If all the boolean subexpressions of a boolean expression are classified, then it is classified according to the truth tables.

Completion Rule — If a boolean expression contains unclassified boolean subexpressions, and all ways of classifying them place it in the same class, then it is in that class.

Consistency Rule — If a classified boolean expression contains boolean subexpressions, and only one way of classifying them is consistent, then they are classified that way.

Instance Rule — If a boolean expression is classified, then all its instances have that same classification.

An <u>axiom</u> is a boolean expression that is stated to be a theorem. An <u>antiaxiom</u> is similarly a boolean expression stated to be an antitheorem. The only axiom of Boolean Theory is $\top$ and the only antiaxiom is $\bot$. So, by the Axiom Rule, $\top$ is a theorem and $\bot$ is an antitheorem. As we present more theories, we will give their axioms and antiaxioms; they, together with the other rules of proof, will determine the new theorems and antitheorems of the new theory.

Before the invention of formal logic, the word "axiom" was used for a statement whose truth was supposed to be obvious. In modern mathematics, an axiom is part of the design and presentation of a theory. Different axioms may yield different theories, and different theories may have different applications. When we design a theory, we can choose any axioms we like, but a bad choice can result in a useless theory.

The entry in the top left corner of the truth table for the two-operand operators does not say $\top\wedge\top = \top$. It says that the conjunction of any two theorems is a theorem. To prove that $\top\wedge\top = \top$ is a theorem requires the boolean axiom (to prove that $\top$ is a theorem), the first entry on the $\wedge$ row of the truth table (to prove that $\top\wedge\top$ is a theorem), and the first entry on the $=$ row of the truth table (to prove that $\top\wedge\top = \top$ is a theorem).

The boolean expression

$$\top \vee x$$

contains an unclassified boolean subexpression, so we cannot use the Evaluation Rule to tell us which class it is in. If x were a theorem, the Evaluation Rule would say that the whole expression is a theorem. If x were an antitheorem, the Evaluation Rule would again say that the whole expression is a theorem. We can therefore conclude by the Completion Rule that the whole expression is indeed a theorem. The Completion Rule also says that

$$x \vee \neg x$$

is a theorem, and when we come to Number Theory, that

$$1/0 = 5 \ \vee \ \neg\ 1/0 = 5$$

is a theorem. We do not need to know that a subexpression is unclassified to use the Completion Rule. If we are ignorant of the classification of a subexpression, and we suppose it to be unclassified, any conclusion we come to by the use of the Completion Rule will still be correct.

In a classified boolean expression, if it would be inconsistent to place a boolean subexpression in one class, then the Consistency Rule says it is in the other class. For example, suppose we know that *expression0* is a theorem, and that *expression0* $\Rightarrow$ *expression1* is also a theorem. Can we determine what class *expression1* is in? If *expression1* were an antitheorem, then by the Evaluation Rule *expression0* $\Rightarrow$ *expression1* would be an antitheorem, and that would be inconsistent. So, by the Consistency Rule, *expression1* is a theorem. This use of the Consistency Rule is traditionally called "detachment" or "modus ponens". As another example, if $\neg$*expression* is a theorem, then the Consistency Rule says that *expression* is an antitheorem.

Thanks to the negation operator and the Consistency Rule, we never need to talk about antiaxioms and antitheorems. Instead of saying that *expression* is an antiaxiom or antitheorem, we can say that $\neg$*expression* is an axiom or theorem. But a word of caution: if a theory is incomplete, it is possible that neither *expression* nor $\neg$*expression* is a theorem. Thus "antitheorem" is not the same as "not a theorem". Our preference for theorems over antitheorems encourages some shortcuts of speech. We sometimes state a boolean expression, such as $1+1=2$, without saying anything about it; when we do so, we mean that it is a theorem. We sometimes say we will prove a boolean expression, meaning we will prove it is a theorem.

End of Axioms and Proof Rules

We now replace the boolean antiaxiom ($\bot$) with an axiom ($\neg\bot$). With our two boolean axioms and five proof rules we can now prove theorems. Some theorems are useful enough to be given a name and be memorized, or at least be kept in a handy list. Such a theorem is called a law. Some laws of Boolean Theory are listed at the back of the book. Laws concerning $\Leftarrow$ have not been included, but any law that uses $\Rightarrow$ can be easily rearranged into one using $\Leftarrow$. All of them can be proven using the Completion Rule, classifying the variables in all possible ways, and evaluating each way. When the number of variables is more than about 2, this kind of proof is quite inefficient. It is much better to prove new laws by making use of already proven old laws. In the next three subsections we see how.

1.0.1 Expression and Proof Format

The precedence table on the final page of this book tells how to parse an expression in the absence of parentheses. To help the eye group the symbols properly, it is a good idea to leave space for absent parentheses. Consider the following two ways of spacing the same expression.

$a{\wedge}b \;\vee\; c$

$a \;\wedge\; b{\vee}c$

According to our rules of precedence, the parentheses belong around $a\wedge b$, so the first spacing is helpful and the second misleading.

An expression that is too long to fit on one line must be broken into parts. There are several reasonable ways to do it; here is one suggestion. A long expression in parentheses can be broken at its main connective, which is placed under the opening parenthesis. For example,

(*first part*
∧ *second part*)

A long expression without parentheses can be broken at its main connective, which is placed under where the opening parenthesis belongs. For example,

first part
= *second part*

Attention to format makes a big difference in our ability to understand a complex expression.

A proof is a boolean expression that is clearly a theorem. What is clear to one person may not be clear to another, so a proof is written for an intended reader. One form of proof is a continuing equation with hints.

	expression0	hint 0
=	*expression1*	hint 1
=	*expression2*	hint 2
=	*expression3*	

This continuing equation is a short way of writing the longer boolean expression

expression0 = *expression1*
∧ *expression1* = *expression2*
∧ *expression2* = *expression3*

The hints on the right side of the page are used, when necessary, to help make it clear that this continuing equation is a theorem. The best kind of hint is the name of a law. The "hint 0" is supposed to make it clear that *expression0* = *expression1* is a theorem. The "hint 1" is supposed to make it clear that *expression1* = *expression2* is a theorem. And so on. By the transitivity of = , this proof proves the theorem *expression0* = *expression3* . A formal proof is a proof in which every step fits the form of the law given as hint. The advantage of making a proof formal is that each step can be checked by a computer, and its validity is not a matter of opinion.

Here is an example. Suppose we want to prove the first Law of Portation

$$a \land b \Rightarrow c \;=\; a \Rightarrow (b \Rightarrow c)$$

using only previous laws in the list at the back of this book. Here is a proof.

	$a \land b \Rightarrow c$	Material Implication
$=$	$\neg(a \land b) \lor c$	Duality
$=$	$\neg a \lor \neg b \lor c$	Material Implication
$=$	$a \Rightarrow \neg b \lor c$	Material Implication
$=$	$a \Rightarrow (b \Rightarrow c)$	

From the first line of the proof, we are told to use "Material Implication", which is the first of the Laws of Inclusion, to obtain the second line of the proof. The first two lines together

$$a \land b \Rightarrow c \;=\; \neg(a \land b) \lor c$$

fit the form of the Law of Material Implication, which is

$$a \Rightarrow b \;=\; \neg a \lor b$$

because $a \land b$ in the proof fits where a is in the law, and c in the proof fits where b is in the law. The next hint is "Duality", and we see that the next line is obtained by replacing $\neg(a \land b)$ with $\neg a \lor \neg b$ in accordance with the first of the Duality Laws. By not using parentheses on that line, we silently use the Associative Law of disjunction, in preparation for the next step. The next hint is again "Material Implication"; this time it is used in the opposite direction, to replace the first disjunction with an implication. And once more, "Material Implication" is used to replace the remaining disjunction with an implication. Therefore, by transitivity of $=$, we conclude that the first Law of Portation is a theorem.

Here is the proof again, in a different form.

	$(a \land b \Rightarrow c \;=\; a \Rightarrow (b \Rightarrow c))$	Material Implication, 3 times
$=$	$(\neg(a \land b) \lor c \;=\; \neg a \lor (\neg b \lor c))$	Duality
$=$	$(\neg a \lor \neg b \lor c \;=\; \neg a \lor \neg b \lor c)$	Reflexivity of $=$
$=$	$\top$	

The final line is a theorem, hence each of the other lines is a theorem, and in particular, the first line is a theorem. This form of proof has some advantages over the earlier form. First, it makes proof the same as simplification to $\top$. Second, although any proof in the first form can be written in the second form, the reverse is not true. For example, the proof

	$(a \Rightarrow b = a \land b) = a$	Associative Law for =
$=$	$(a \Rightarrow b = (a \land b = a))$	a Law of Inclusion
$=$	$\top$	

cannot be converted to the other form. And finally, the second form, simplification to $\top$, can be used for theorems that are not equations; the main operator of the boolean expression can be anything, including $\land$, $\lor$, or $\neg$.

The proofs in this book are intended to be read by people, rather than by a computer. Sometimes it is clear enough how to get from one line to the next without a hint, and in that case no hint will be given. Hints are optional, to be used whenever they are helpful. Sometimes a hint is too long to fit on the remainder of a line. We may have

	expression0	short hint
=	*expression1*	and now a very long hint, written just as this is written, on as many lines as necessary, followed by
=	*expression2*	

We cannot excuse an inadequate hint by the limited space on one line.

End of Expression and Proof Format

1.0.2 Monotonicity and Antimonotonicity

A proof can be a continuing equation, as we have seen; it can also be a continuing implication, or a continuing mixture of equations and implications. As an example, here is a proof of the first Law of Conflation, which says

$$(a \Rightarrow b) \land (c \Rightarrow d) \ \Longrightarrow\ a \land c \Rightarrow b \land d$$

The proof goes this way: starting with the right side,

$$\begin{array}{lll} & a \land c \Rightarrow b \land d & \text{distribute } \Rightarrow \text{ over second } \land \\ = & (a \land c \Rightarrow b) \land (a \land c \Rightarrow d) & \text{antidistribution twice} \\ = & ((a \Rightarrow b) \lor (c \Rightarrow b)) \land ((a \Rightarrow d) \lor (c \Rightarrow d)) & \text{distribute } \land \text{ over } \lor \text{ twice} \\ = & (a \Rightarrow b) \land (a \Rightarrow d) \lor (a \Rightarrow b) \land (c \Rightarrow d) \lor (c \Rightarrow b) \land (a \Rightarrow d) \lor (c \Rightarrow b) \land (c \Rightarrow d) & \text{generalization} \\ \Longleftarrow & (a \Rightarrow b) \land (c \Rightarrow d) & \end{array}$$

From the mutual transitivity of $=$ and $\Longleftarrow$, we have proven

$$a \land c \Rightarrow b \land d \ \Longleftarrow\ (a \Rightarrow b) \land (c \Rightarrow d)$$

which can easily be rearranged to give the desired theorem.

The implication operator is reflexive $a \Rightarrow a$, antisymmetric $(a \Rightarrow b) \land (b \Rightarrow a) = (a=b)$, and transitive $(a \Rightarrow b) \land (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$. It is therefore an ordering (just like $\le$ for numbers). We pronounce $a \Rightarrow b$ either as " a implies b ", or, to emphasize the ordering, as " a is <u>stronger</u> than or equal to b ". Likewise $a \Leftarrow b$ is pronounced either as " a is implied by b ", or as " a is <u>weaker</u> than or equal to b ". The words "stronger" and "weaker" may have come from a philosophical origin; we ignore any meaning they may have other than the boolean order, in which $\bot$ is stronger than $\top$.

The Monotonic Law $a \Rightarrow b \ \Longrightarrow\ c \land a \Rightarrow c \land b$ can be read (a little carelessly) as follows: if a is weakened to b, then $c \land a$ is weakened to $c \land b$. (To be more careful, we should say "weakened or equal".) If we weaken a, then we weaken $c \land a$. Or, the other way round, if we strengthen b, then we strengthen $c \land b$. Whatever happens to a conjunct (weaken or strengthen), the same happens to the conjunction. We say that conjunction is <u>monotonic</u> in its conjuncts.

The Antimonotonic Law $a \Rightarrow b \ \Longrightarrow\ (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$ says that whatever happens to an antecedent (weaken or strengthen), the opposite happens to the implication. We say that implication is <u>antimonotonic</u> in its antecedent.

Here are the monotonic and antimonotonic properties of boolean expressions.

- $\lnot a$ is antimonotonic in a
- $a \land b$ is monotonic in a and monotonic in b
- $a \lor b$ is monotonic in a and monotonic in b
- $a \Rightarrow b$ is antimonotonic in a and monotonic in b
- $a \Leftarrow b$ is monotonic in a and antimonotonic in b
- **if** a **then** b **else** c is monotonic in b and monotonic in c

These properties are useful in proofs. For example, in Exercise 2(k), to prove $\lnot(a \land \lnot(a \lor b))$, we can employ the Law of Generalization $a \Rightarrow a \lor b$ to strengthen $a \lor b$ to a. That weakens $\lnot(a \lor b)$ and that weakens $a \land \lnot(a \lor b)$ and that strengthens $\lnot(a \land \lnot(a \lor b))$.

$$\begin{array}{lll} & \lnot(a \land \lnot(a \lor b)) & \text{use the Law of Generalization} \\ \Longleftarrow & \lnot(a \land \lnot a) & \text{now use the Law of Noncontradiction} \\ = & \top & \end{array}$$

We thus prove that $\lnot(a \land \lnot(a \lor b)) \Longleftarrow \top$, and by an identity law, that is the same as proving $\lnot(a \land \lnot(a \lor b))$. In other words, $\lnot(a \land \lnot(a \lor b))$ is weaker than or equal to $\top$, and since there is

nothing weaker than $\top$, it is equal to $\top$. When we drive toward $\top$, the left edge of the proof can be any mixture of $=$ and $\Leftarrow$ signs.

Similarly we can drive toward $\bot$, and then the left edge of the proof can be any mixture of $=$ and $\Rightarrow$ signs. For example,

$$
\begin{array}{lll}
 & a \land \neg(a \lor b) & \text{use the Law of Generalization} \\
\Rightarrow & a \land \neg a & \text{now use the Law of Noncontradiction} \\
= & \bot &
\end{array}
$$

This is called "proof by contradiction". It proves $a \land \neg(a \lor b) \Rightarrow \bot$, which is the same as proving $\neg(a \land \neg(a \lor b))$. Any proof by contradiction can be converted to a proof by simplification to $\top$ at the cost of one $\neg$ sign per line.

End of Monotonicity and Antimonotonicity

1.0.3 Context

A proof, or part of a proof, can make use of local assumptions. A proof may have the format

$$
\begin{array}{lll}
 & \mathit{assumption} & \\
\Rightarrow & (& \mathit{expression0} \\
 & = & \mathit{expression1} \\
 & = & \mathit{expression2} \\
 & = & \mathit{expression3}\ )
\end{array}
$$

for example. The step $\mathit{expression0} = \mathit{expression1}$ can make use of the *assumption* just as though it were an axiom. So can the step $\mathit{expression1} = \mathit{expression2}$, and so on. Within the parentheses we have a proof; it can be any kind of proof including one that makes further local assumptions. We thus can have proofs within proofs, indenting appropriately. If the subproof is proving $\mathit{expression0} = \mathit{expression3}$, then the whole proof is proving

$$\mathit{assumption} \Rightarrow (\mathit{expression0} = \mathit{expression3})$$

If the subproof is proving *expression0*, then the whole proof is proving

$$\mathit{assumption} \Rightarrow \mathit{expression0}$$

If the subproof is proving $\bot$, then the whole proof is proving

$$\mathit{assumption} \Rightarrow \bot$$

which is equal to $\neg \mathit{assumption}$. Once again, this is "proof by contradiction".

We can also use **if then else** as a proof, or part of a proof, in a similar manner. The format is

if *possibility*
then (first subproof
assuming *possibility*
as a local axiom)
else (second subproof
assuming $\neg$*possibility*
as a local axiom)

If the first subproof proves *something* and the second proves *anotherthing*, the whole proof proves

if *possibility* **then** *something* **else** *anotherthing*

If both subproofs prove the same thing, then by the Case Idempotent Law, so does the whole proof, and that is its most frequent use.

Consider a step in a proof that looks like this:

$$
\begin{array}{ll}
 & \mathit{expression0} \land \mathit{expression1} \\
= & \mathit{expression0} \land \mathit{expression2}
\end{array}
$$

When we are changing *expression1* into *expression2* , we can assume *expression0* as a local axiom just for this step. If *expression0* really is a theorem, then we have done no harm by assuming it as a local axiom. If, however, *expression0* is an antitheorem, then both *expression0* ∧ *expression1* and *expression0* ∧ *expression2* are antitheorems no matter what *expression1* and *expression2* are, so again we have done nothing wrong. Symmetrically, when proving

$$\begin{array}{ll} & expression0 \land expression1 \\ = & expression2 \land expression1 \end{array}$$

we can assume *expression1* as a local axiom. However, when proving

$$\begin{array}{ll} & expression0 \land expression1 \\ = & expression2 \land expression3 \end{array}$$

we cannot assume *expression0* to prove *expression1=expression3* and in the same step assume *expression1* to prove *expression0=expression2* . For example, starting from $a \land a$, we can assume the first a and so change the second one to $\top$,

$$\begin{array}{lll} & a \land a & \text{assume first } a \text{ to simplify second } a \\ = & a \land \top & \end{array}$$

or we can assume the second a and so change the first one to $\top$,

$$\begin{array}{lll} & a \land a & \text{assume second } a \text{ to simplify first } a \\ = & \top \land a & \end{array}$$

but we cannot assume both of them at the same time.

$$\begin{array}{lll} & a \land a & \text{this step is wrong} \\ = & \top \land \top & \end{array}$$

In this paragraph, the equal signs could have been implications in either direction.

Here is a list of context rules for proof.
In *expression0* ∧ *expression1* , when changing *expression0* , we can assume *expression1* .
In *expression0* ∧ *expression1* , when changing *expression1* , we can assume *expression0* .
In *expression0* ∨ *expression1* , when changing *expression0* , we can assume ¬*expression1* .
In *expression0* ∨ *expression1* , when changing *expression1* , we can assume ¬*expression0* .
In *expression0* ⇒ *expression1* , when changing *expression0* , we can assume ¬*expression1* .
In *expression0* ⇒ *expression1* , when changing *expression1* , we can assume *expression0* .
In *expression0* ⇐ *expression1* , when changing *expression0* , we can assume *expression1* .
In *expression0* ⇐ *expression1* , when changing *expression1* , we can assume ¬*expression0* .
In **if** *expression0* **then** *expression1* **else** *expression2* , when changing *expression0* ,
we can assume *expression1*⧧*expression2* .
In **if** *expression0* **then** *expression1* **else** *expression2* , when changing *expression1* ,
we can assume *expression0* .
In **if** *expression0* **then** *expression1* **else** *expression2* , when changing *expression2* ,
we can assume ¬*expression0* .

In the previous subsection we proved Exercise 2(k): $\neg(a \land \neg(a \lor b))$. Here is another proof, this time using context.

$$\begin{array}{lll} & \neg(a \land \neg(a \lor b)) & \text{assume } a \text{ to simplify } \neg(a \lor b) \\ = & \neg(a \land \neg(\top \lor b)) & \text{Symmetry Law and Base Law for } \lor \\ = & \neg(a \land \neg\top) & \text{Truth Table for } \neg \\ = & \neg(a \land \bot) & \text{Base Law for } \land \\ = & \neg\bot & \text{Boolean Axiom, or Truth Table for } \neg \\ = & \top & \end{array}$$

End of Context

1.0.4 Formalization

We use computers to solve problems, or to provide services, or just for fun. The desired computer behavior is usually described at first informally, in a natural language (like English), perhaps with some diagrams, perhaps with some hand gestures, rather than formally, using mathematical formulas (notations). In the end, the desired computer behavior is described formally as a program. A programmer must be able to translate informal descriptions to formal ones.

A statement in a natural language can be vague, ambiguous, or subtle, and can rely on a great deal of cultural context. This makes formalization difficult, but also necessary. We cannot possibly say how to formalize, in general; it requires a thorough knowledge of the natural language, and is always subject to argument. In this subsection we just point out a few pitfalls in the translation from English to boolean expressions.

The best translation may not be a one-for-one substitution of symbols for words. The same word in different places may be translated to different symbols, and different words may be translated to the same symbol. The words "and", "also", "but", "yet", "however", and "moreover" might all be translated as $\land$. Just putting things next to each other sometimes means $\land$. For example, "They're red, ripe, and juicy, but not sweet." becomes $red \land ripe \land juicy \land \neg sweet$.

The word "or" in English is sometimes best translated as $\lor$, and sometimes as ⧧ . For example, "They're either small or rotten." probably includes the possibility that they're both small and rotten, and should be translated as $small \lor rotten$. But "Either we eat them or we preserve them." probably excludes doing both, and is best translated as *eat* ⧧ *preserve* .

The word "if" in English is sometimes best translated as $\Rightarrow$, and sometimes as $=$. For example, "If it rains, we'll stay home." probably leaves open the possibility that we might stay home even if it doesn't rain, and should be translated as $rain \Rightarrow home$. But "If it snows, we can go skiing." probably also means "and if it doesn't, we can't", and is best translated as $snow = ski$.

End of Formalization

End of Boolean Theory

1.1 Number Theory

Number Theory, also known as arithmetic, was designed to represent quantity. In the version we present, a number expression is formed in the following ways.

a sequence of one or more decimal digits

∞	"infinity"
$-x$	"minus x "
$x+y$	" x plus y "
$x-y$	" x minus y "
$x \times y$	" x times y "
x / y	" x divided by y "
x^y	" x to the power y "

if a **then** x **else** y

where x and y are any number expressions, and a is any boolean expression. The infinite number expression ∞ will be essential when we talk about the execution time of programs. We also introduce several new ways of forming boolean expressions:

$x < y$	" x is less than y "
$x \le y$	" x is less than or equal to y "
$x > y$	" x is greater than y "
$x \ge y$	" x is greater than or equal to y "
$x = y$	" x equals y ", " x is equal to y "
$x \neq y$	" x differs from y ", " x is unequal to y "

The axioms of Number Theory are listed at the back of the book. It's a long list, but most of them should be familiar to you already. Notice particularly the two axioms

$-\infty \le x \le \infty$	extremes
$-\infty < x \Rightarrow \infty + x = \infty$	absorption

Number Theory is incomplete. For example, the boolean expressions $1/0 = 5$ and $0 < (-1)^{1/2}$ can neither be proven nor disproven.

End of Number Theory

1.2 Character Theory

The simplest character expressions are written as a graphical shape enclosed by double-quotes. For example, "A" is the "capital A" character, "1" is the "one" character, and " " is the "space" character. The double-quote character must be written twice, and enclosed, like this: """" . Character Theory is trivial. It has operators *succ* (successor), *pred* (predecessor), and $= \neq < \le > \ge$ **if then else** . We leave the details of this theory to the reader's inclination.

End of Character Theory

All our theories use the operators $= \neq$ **if then else** , so their laws are listed at the back of the book under the heading "Generic", meaning that they are part of every theory. These laws are not needed as axioms of Boolean Theory; for example, $x=x$ can be proven using the Completion and Evaluation rules. But in Number Theory and other theories, they are axioms; without them we cannot even prove $5=5$.

The operators $< \le > \ge$ apply to some, but not all, types of expression. Whenever they do apply, their axioms, as listed under the heading "Generic" at the back of the book, go with them.

End of Basic Theories

We have talked about boolean expressions, number expressions, and character expressions. In the following chapters, we will talk about bunch expressions, set expressions, string expressions, list expressions, function expressions, predicate expressions, relation expressions, specification expressions, and program expressions; so many expressions. For brevity in the following chapters, we will often omit the word "expression", just saying boolean, number, character, bunch, set, string, list, function, predicate, relation, specification, and program, meaning in each case a type of expression. If this bothers you, please mentally insert the word "expression" wherever you would like it to be.

2 Basic Data Structures

A data structure is a collection, or aggregate, of data. The data may be booleans, numbers, characters, or data structures. The basic kinds of structuring we consider are packaging and indexing. These two kinds of structure give us four basic data structures.

unpackaged, unindexed:	bunch
packaged, unindexed:	set
unpackaged, indexed:	string
packaged, indexed:	list

2.0 Bunch Theory

A bunch represents a collection of objects. For contrast, a set represents a collection of objects in a package or container. A bunch is the contents of a set. These vague descriptions are made precise as follows.

Any number, character, or boolean (and later also set, string of elements, and list of elements) is an elementary bunch, or element. For example, the number 2 is an elementary bunch, or synonymously, an element. Every expression is a bunch expression, though not all are elementary.

From bunches A and B we can form the bunches

A, B	" A union B "
$A ‘ B$	" A intersection B "

and the number

$¢A$	"size of A ", "cardinality of A "

and the boolean

$A: B$	" A is in B ", " A is included in B "

The size of a bunch is the number of elements it includes. Elements are bunches of size 1 .

$¢2 = 1$
$¢(0, 2, 5, 9) = 4$

Here are three quick examples of bunch inclusion.

$2: 0, 2, 5, 9$
$2: 2$
$2, 9: 0, 2, 5, 9$

The first says that 2 is in the bunch consisting of 0, 2, 5, 9 . The second says that 2 is in the bunch consisting of only 2 . Note that we do not say "a bunch contains its elements", but rather "a bunch consists of its elements". The last example says that both 2 and 9 are in 0, 2, 5, 9 , or in other words, the bunch 2, 9 is included in the bunch 0, 2, 5, 9 .

Here are the axioms of Bunch Theory. In these axioms, x and y are elements (elementary bunches), and A , B , and C are arbitrary bunches.

$x: y \;=\; x=y$	elementary axiom
$x: A,B \;=\; x: A \lor x: B$	compound axiom
$A,A = A$	idempotence
$A,B = B,A$	symmetry

$A,(B,C) = (A,B),C$	associativity
$A‘A = A$	idempotence
$A‘B = B‘A$	symmetry
$A‘(B‘C) = (A‘B)‘C$	associativity
$A,B: C \;=\; A: C \;\land\; B: C$	antidistributivity
$A: B‘C \;=\; A: B \;\land\; A: C$	distributivity
$A: A,B$	generalization
$A‘B: A$	specialization
$A: A$	reflexivity
$A: B \;\land\; B: A \;=\; A=B$	antisymmetry
$A: B \;\land\; B: C \;\Rightarrow\; A: C$	transitivity
$¢x = 1$	size
$¢(A, B) + ¢(A‘B) = ¢A + ¢B$	size
$\neg\, x: A \;\Rightarrow\; ¢(A‘x) = 0$	size
$A: B \;\Rightarrow\; ¢A \le ¢B$	size

From these axioms, many laws can be proven. Among them:

$A,(A‘B) \;=\; A$	absorption
$A‘(A,B) \;=\; A$	absorption
$A: B \;\Rightarrow\; C,A: C,B$	monotonicity
$A: B \;\Rightarrow\; C‘A: C‘B$	monotonicity
$A: B \;=\; A,B = B \;=\; A = A‘B$	inclusion
$A,(B,C) \;=\; (A,B),(A,C)$	distributivity
$A,(B‘C) \;=\; (A,B)‘(A,C)$	distributivity
$A‘(B,C) \;=\; (A‘B), (A‘C)$	distributivity
$A‘(B‘C) \;=\; (A‘B)‘(A‘C)$	distributivity
$A: B \;\land\; C: D \;\Rightarrow\; A,C: B,D$	conflation
$A: B \;\land\; C: D \;\Rightarrow\; A‘C: B‘D$	conflation

Here are several bunches that we will find useful:

null			the empty bunch
bool	=	$\top, \bot$	the booleans
nat	=	$0, 1, 2, ...$	the natural numbers
int	=	$..., -2, -1, 0, 1, 2, ...$	the integer numbers
rat	=	$..., -1, 0, 2/3, ...$	the rational numbers
real	=	$..., 2^{1/2}, ...$	the real numbers
xnat	=	$0, 1, 2, ..., \infty$	the extended naturals
xint	=	$-\infty, ..., -2, -1, 0, 1, 2, ..., \infty$	the extended integers
xrat	=	$-\infty, ..., -1, 0, 2/3, ..., \infty$	the extended rationals
xreal	=	$-\infty, ..., \infty$	the extended reals
char	=	..., "a", "A", ...	the characters

In these equations, whenever three dots appear they mean “guess what goes here”. This use of three dots is informal, so these equations cannot serve as definitions, though they may help to give you the idea. We define these bunches formally in a moment.

The operators $, ‘ ¢ : = +$ **if then else** apply to bunch operands according to the axioms already presented. Some other operators can be applied to bunches with the understanding that they apply to the elements of the bunch. In other words, they distribute over bunch union. For example,

$$-null = null$$
$$-(A, B) = -A, -B$$
$$A+null = null+A = null$$
$$(A, B)+(C, D) = A+C, A+D, B+C, B+D$$

This makes it easy to express the positive naturals $(nat+1)$, the even naturals $(nat\times 2)$, the squares (nat^2), the powers of two (2^{nat}), and many other things. (The operators that distribute over bunch union are listed on the final page.)

We define the empty bunch, $null$, with the axioms

$$null: A$$
$$¢A = 0 \;\;=\;\; A = null$$

This gives us three more laws:

$A, null = A$	identity
$A \text{ ‘ } null = null$	base
$¢\, null = 0$	size

The bunch $bool$ is defined by the axiom

$$bool = \top, \bot$$

The bunch nat is defined by the two axioms

$0, nat+1: nat$	construction
$0, B+1: B \;\Rightarrow\; nat: B$	induction

Construction says that $0, 1, 2,$ and so on, are in nat. Induction says that nothing else is in nat by saying that of all the bunches B satisfying the construction axiom, nat is the smallest. In some books, particularly older ones, the natural numbers start at 1; we will use the term with its current and more useful meaning, starting at 0. The bunches int, rat, $xnat$, $xint$, and $xrat$ can be defined as follows.

$$int = nat, -nat$$
$$rat = int/(nat+1)$$
$$xnat = nat, \infty$$
$$xint = -\infty, int, \infty$$
$$xrat = -\infty, rat, \infty$$

The definition of $real$ is postponed until the next chapter (functions). Bunch $real$ won't be used before it is defined, except to say

$$xreal = -\infty, real, \infty$$

We do not care enough about the bunch $char$ to define it.

We also use the notation

$x,..y$ “ x to y ” (not “ x through y ”)

where x is an integer and y is an extended integer and $x \le y$. Its axiom is

$$i: x,..y \;\;=\;\; x \le i < y$$

where i is an extended integer. The notation $,..$ is asymmetric as a reminder that the left end of the interval is included and the right end is excluded. For example,

$$0,..\infty = nat$$
$$5,..5 = null$$
$$¢(x,..y) = y-x$$

The $,..$ notation is formal. We have an axiom defining it, so we don't have to guess what is included.

End of Bunch Theory

2.1 Set Theory optional

Let A be any bunch (anything). Then

$\{A\}$ "set containing A"

is a set. Thus $\{null\}$ is the empty set, and the set containing the first three natural numbers is expressed as $\{0, 1, 2\}$ or as $\{0,..3\}$. All sets are elements; not all bunches are elements; that is the difference between sets and bunches. We can form the bunch $1, \{3, 7\}$ consisting of two elements, and from it the set $\{1, \{3, 7\}\}$ containing two elements, and in that way we build a structure of nested sets.

The inverse of set formation is also useful. If S is any set, then

$\sim S$ "contents of S"

is its contents. For example,

$\sim\{0, 1\} = 0, 1$

The power operator ϟ applies to a bunch and yields all sets that contain only elements of the bunch. Here is an example.

ϟ$(0, 1) = \{null\}, \{0\}, \{1\}, \{0, 1\}$

We "promote" the bunch operators to obtain the set operators $\$ \in \subseteq \cup \cap =$. Here are the axioms.

$\{A\} \neq A$	structure
$\{\sim S\} = S$	set formation
$\sim\{A\} = A$	"contents"
$\$\{A\} = ¢A$	"size", "cardinality"
$A \in \{B\} \;=\; A: B$	"elements"
$\{A\} \subseteq \{B\} \;=\; A: B$	"subset"
$\{A\}:$ ϟ$B \;=\; A: B$	"power"
$\{A\} \cup \{B\} = \{A, B\}$	"union"
$\{A\} \cap \{B\} = \{A ‘ B\}$	"intersection"
$\{A\} = \{B\} \;=\; A = B$	"equation"

End of Set Theory

Bunches are unpackaged collections and sets are packaged collections. Similarly, strings are unpackaged sequences and lists are packaged sequences. There are sets of sets, and lists of lists, but there are neither bunches of bunches nor strings of strings.

2.2 String Theory

The simplest string is

nil the empty string

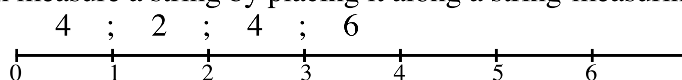
Any number, character, boolean, set, (and later also list and function) is a one-item string, or <u>item</u>. For example, the number 2 is a one-item string, or item. A nonempty bunch of items is also an item. Strings are <u>catenate</u>d (joined) together by semicolons to make longer strings. For example,

$4; 2; 4; 6$

is a four-item string. The length of a string is obtained by the $\leftrightarrow$ operator.

$\leftrightarrow(4; 2; 4; 6) = 4$

We can measure a string by placing it along a string-measuring ruler, as in the following picture.



Each of the numbers under the ruler is called an index. When we are considering the items in a string from beginning to end, and we say we are at index n, it is clear which items have been considered and which remain because we draw the items between the indexes. (If we were to draw an item at an index, saying we are at index n would leave doubt as to whether the item at that index has been considered.)

The picture saves one confusion, but causes another: we must refer to the items by index, and two indexes are equally near each item. We adopt the convention that most often avoids the need for a "+1" or "−1" in our expressions: the index of an item is the number of items that precede it. In other words, indexing is from 0 . Your life begins at year 0 , a highway begins at mile 0 , and so on. An index is not an arbitrary label, but a measure of how much has gone before. We refer to the items in a string as "item 0", "item 1", "item 2", and so on; we never say "the third item" due to the possible confusion between item 2 and item 3. When we are at index n, then n items have been considered, and item n will be considered next.

We obtain an item of a string by subscripting. For example,

$(3; 5; 7; 9)_2 = 7$

In general, S_n is item n of string S. We can even pick out a whole string of items, as in the following example.

$(3; 5; 7; 9)_{2;\,1;\,2} = 7; 5; 7$

If n is an extended natural and S is a string, then $n{*}S$ means n copies of S catenated together.

$3 * (0; 1) = 0; 1; 0; 1; 0; 1$

Without any left operand, $*S$ means all strings formed by catenating any number of copies of S.

$*(0; 1) = nil\ , 0;1\ , 0;1;0;1\ , ...$

If S is a string and n is an index of S and i is an item (not necessarily of S), then $S{\triangleleft}n{\triangleright}i$ is a string like S except that the item at index n is i. For example,

$(3; 5; 9){\triangleleft}2{\triangleright}8 = 3; 5; 8$

Strings can be compared for equality and order. To be equal, strings must be of equal length, and have equal items at each index. The order of two strings is determined by the items at the first index where they differ. For example,

$3; 6; 4; 7 < 3; 7; 2$

If there is no index where they differ, the shorter string comes before the longer one.

$3; 6; 4 < 3; 6; 4; 7$

This ordering is known as lexicographic order; it is the ordering used in dictionaries.

Here is the syntax of strings. If i is an item, S and T are strings, and n is an extended natural number, then

nil	the empty string
i	an item
$S;T$	" S catenate T "
S_T	" S sub T "
$n{*}S$	" n copies of S "
$S{\triangleleft}n{\triangleright}i$	" S but at n there's i "

are strings,

$*S$	"copies of S "

is a bunch of strings, and

$\leftrightarrow S$	"length of S "

is an extended natural number. The order operators $< \le > \ge$ apply to strings.

Here are the axioms of String Theory. In these axioms, S , T , and U are strings, i and j are items, and n is an extended natural number.

$nil; S \;=\; S; nil \;=\; S$ identity
$S; (T; U) \;=\; (S; T); U$ associativity
$\leftrightarrow nil \;=\; 0$ base
$\leftrightarrow i \;=\; 1$ base
$\leftrightarrow(S; T) \;=\; \leftrightarrow S + \leftrightarrow T$
$S_{nil} \;=\; nil$
$\leftrightarrow S<\infty \;\Rightarrow\; (S; i; T)_{\leftrightarrow S} \;=\; i$
$S_{T;\, U} \;=\; S_T ; S_U$
$S_{(T_U)} \;=\; (S_T)_U$
$0{*}S \;=\; nil$
$(n{+}1){*}S \;=\; n{*}S; S$
$\leftrightarrow S<\infty \;\Rightarrow\; S;i;T \triangleleft \leftrightarrow S \triangleright j \;=\; S;j;T$
$\leftrightarrow S<\infty \;\Rightarrow\; nil \;\le\; S \;<\; S; i; T$
$\leftrightarrow S<\infty \;\Rightarrow\; (i<j \;\;=\;\; S; i; T \;<\; S; j; U)$
$\leftrightarrow S<\infty \;\Rightarrow\; (i=j \;\;=\;\; S; i; T \;=\; S; j; T)$

We also use the notation

$x;..y$ “ x to y ” (same pronunciation as $x,..y$)

where x is an integer and y is an extended integer and $x{\le}y$. As in the similar bunch notation, x is included and y excluded, so that

$\leftrightarrow(x;..y) \;=\; y{-}x$

Here are the axioms.

$x;..x \;=\; nil$
$x;..x{+}1 \;=\; x$
$(x;..y) \;;\; (y;..z) \;=\; x;..z$

The <u>text</u> notation is an alternative way of writing a string of characters. A text begins with a double-quote, continues with any natural number of characters (but a double-quote character within the text must be written twice), and concludes with a double-quote. Here is a text of length 15 .

"Don't say ""no""." = "D"; "o"; "n"; "'"; "t"; " "; "s"; "a"; "y"; " "; """"; "n"; "o"; """"; "."

The empty text "" is another way of writing nil . Indexing a text with a string of indexes, we obtain a subtext. For example,

"abcdefghij"$_{3;..6}$ = "def"

Here is a self-describing expression (self-reproducing automaton).

"""$_{0;0;2*(0;..15)}$"$_{0;0;2*(0;..15)}$

Perform the indexing and see what you get.

String catenation distributes over bunch union:

$A; null; B \;=\; null$
$(A, B); (C, D) \;=\; A;C, A;D, B;C, B;D$

So a string of bunches is equal to a bunch of strings. Thus, for example,

$0; 1; 2{:}\;\; nat; 1; (0,..10)$

because $0{:}\, nat$ and $1{:}\, 1$ and $2{:}\, 0,..10$. A string is an element (elementary bunch) just when all its items are elements; so $0;1;2$ is an element, but $nat; 1; (0,..10)$ is not. Progressing to larger bunches,

$0; 1; 2{:}\;\; nat; 1; (0,..10){:}\;\; 3{*}nat{:}\;\; {*}nat$

The $*$ operator distributes over bunch union in its left operand only.

$null*A = null$
$(A, B) * C = A*C, B*C$
Using this left-distributivity, we define the one-operand * by the axiom
$*A = nat*A$

The strings we have just defined have natural indexes and extended natural lengths. By adding a new operator, the inverse of catenation, we obtain strings that have negative indexes and lengths. We leave this development as Exercise 46.

End of String Theory

2.3 List Theory

A list is a packaged string. For example,
[0; 1; 2]
is a list of three items. List brackets [ ] distribute over bunch union.
$[null] = null$
$[A, B] = [A], [B]$
Because 0: *nat* and 1: 1 and 2: 0,..10 we can say
[0; 1; 2]: [*nat*; 1; (0,..10)]
On the left of the colon we have a list of integers; on the right we have a list of bunches, or equivalently, a bunch of lists. A list is an element (elementary bunch) just when all its items are elements; [0; 1; 2] is an element, but [*nat*; 1; (0,..10)] is not. Progressing to larger bunches,
[0; 1; 2]: [*nat*; 1; (0,..10)]: [3**nat*]: [**nat*]

Here is the syntax of lists. Let S be a string, L and M be lists, n be a natural number, and i be an item. Then

$[S]$	"list containing S"
$L\ M$	"$L\ M$" or "L composed with M"
$L{+}M$	"L catenate M"
$n{\to}i \mid L$	"n maps to i otherwise L"

are lists,

$\sim L$	"contents of L"

is a string,

$\#L$	"length of L"

is an extended natural number, and

$L\ n$	"$L\ n$" or "L at index n"

is an item. Of course, parentheses may be used around any expression, so we may write $L(n)$ if we want. If the index is not simple, we must enclose it in parentheses. When there is no danger of confusion, we may write Ln without a space between, but when we use multicharacter names, we must put a space between.

The contents of a list is the string of items it contains.
~[3; 5; 7; 4] = 3; 5; 7; 4
The length of a list is the number of items it contains.
#[3; 5; 7; 4] = 4
List indexes, like string indexes, start at 0 . An item can be selected from a list by juxtaposing (sitting next to each other) a list and an index.
[3; 5; 7; 4] 2 = 7
A list of indexes gives a list of selected items. For example,
[3; 5; 7; 4] [2; 1; 2] = [7; 5; 7]

This is called list composition. List catenation is written with a small raised plus sign $^+$.

$[3; 5; 7; 4]^+[2; 1; 2] \ = \ [3; 5; 7; 4; 2; 1; 2]$

The notation $n \to i \mid L$ gives us a list just like L except that item n is i .

$2 \to 22 \mid [10;..15] \ = \ [10; 11; 22; 13; 14]$

$2 \to 22 \mid 3 \to 33 \mid [10;..15] \ = \ [10; 11; 22; 33; 14]$

Let $L \ = \ [10;..15]$. Then

$2 \to L3 \mid 3 \to L2 \mid L \ = \ [10; 11; 13; 12; 14]$

The order operators $< \le > \ge$ apply to lists; the order is lexicographic, just like string order.

Here are the axioms. Let L be a list, let S and T be strings, let n be a natural number, and let i and j be items.

$[S] \ \ne \ S$	structure
$[\sim L] \ = \ L$	list formation
$\sim[S] \ = \ S$	contents
$\#[S] \ = \ \leftrightarrow S$	length
$[S]^+[T] \ = \ [S; T]$	catenation
$[S]\, n \ = \ S_n$	indexing
$[S]\, [T] \ = \ [S_T]$	composition
$n \to i \mid [S] \ = \ [S \triangleleft n \triangleright i]$	modification
$[S] = [T] \ \ = \ \ S = T$	equation
$[S] < [T] \ \ = \ \ S < T$	order

We can now prove a variety of theorems, such as for lists L , M , N , and natural n , that

$(L\, M)\, n \ = \ L\, (M\, n)$	
$(L\, M)\, N \ = \ L\, (M\, N)$	associativity
$L\, (M^+N) \ = \ L\, M\, ^+\, L\, N$	distributivity

When a list is indexed by a list, we get a list of results. For example,

$[1; 4; 2; 8; 5; 7; 1; 4]\ [1; 3; 7] \ = \ [4; 8; 4]$

We say that list M is a sublist of list L if M can be obtained from L by a list of increasing indexes. So $[4; 8; 4]$ is a sublist of $[1; 4; 2; 8; 5; 7; 1; 4]$. If the list of indexes is not only increasing but consecutive $[i;..j]$, then the sublist is called a segment.

If a list is indexed by a list, the result is a list. More generally, strings and lists can be indexed by any structure, and the result will have that same structure. Let A and B be bunches, let S , T , and U be strings, and let L be a list.

$S_{null} \ = \ null$	$L\, null \ = \ null$
$S_{A,B} \ = \ S_A, S_B$	$L\, (A, B) \ = \ L\, A, L\, B$
$S_{\{A\}} \ = \ \{S_A\}$	$L\, \{A\} \ = \ \{L\, A\}$
$S_{nil} \ = \ nil$	$L\, nil \ = \ nil$
$S_{T;U} \ = \ S_T\, ; S_U$	$L\, (S; T) \ = \ L\, S; L\, T$
$S_{[T]} \ = \ [S_T]$	$L\, [S] \ = \ [L\, S]$

Here is a fancy string example. Let $S = 10; 11; 12$. Then

$S_{0,\ \{1,\ [2;\ 1];\ 0\}}$

$= \quad S_0, \{S_1, [S_2; S_1]; S_0\}$

$= \quad 10, \{11, [12; 11]; 10\}$

Here is a fancy list example. Let $L = [10; 11; 12]$. Then

$L\, (0, \{1, [2; 1]; 0\})$

$= \quad L\, 0, \{L\, 1, [L\, 2; L\, 1]; L\, 0\}$

$= \quad 10, \{11, [12; 11]; 10\}$

2.3.0 Multidimensional Structures

Lists can be items in a list. For example, let

$A = [\ [6; 3; 7; 0] ;$
$\quad [4; 9; 2; 5] ;$
$\quad [1; 5; 8; 3]\ ]$

Then A is a 2-dimensional array, or more particularly, a 3×4 array. Formally, A: [3*[4*nat]] . Indexing A with one index gives a list

$A\ 1 = [4; 9; 2; 5]$

which can then be indexed again to give a number.

$A\ 1\ 2 = 2$

Warning: The notations $A(1,2)$ and $A[1,2]$ are used in several programming languages to index a 2-dimensional array. But in this book,

$A\ (1, 2) = A\ 1, A\ 2 = [4; 9; 2; 5], [1; 5; 8; 3]$
$A\ [1, 2] = [A\ 1, A\ 2] = [\ [4; 9; 2; 5], [1; 5; 8; 3]\ ] = [[4; 9; 2; 5]], [[1; 5; 8; 3]]$

We have just seen a rectangular array, a very regular structure, which requires two indexes to give a number. Lists of lists can also be quite irregular in shape, not just by containing lists of different lengths, but in dimensionality. For example, let

$B = [\ [2; 3]; 4; [5; [6; 7]\ ]\ ]$

Now $B\ 0\ 0 = 2$ and $B\ 1 = 4$, and $B\ 1\ 1$ is undefined. The number of indexes needed to obtain a number varies. We can regain some regularity in the following way. Let L be a list, let n be an index, and let S and T be strings of indexes. Then

$L@nil = L$
$L@n = L\ n$
$L@(S; T) = L@S@T$

Now we can always "index" with a single string, called a pointer, obtaining the same result as indexing by the sequence of items in the string. In the example list,

$B@(2; 1; 0) = B\ 2\ 1\ 0 = 6$

We generalize the notation $S \to i \mid L$ to allow S to be a string of indexes. The axioms are

$nil \to i \mid L = i$
$(S;T) \to i \mid L = S \to (T \to i \mid L@S) \mid L$

Thus $S \to i \mid L$ is a list like L except that S points to item i . For example,

$(0;1) \to 6 \mid [\ [0; 1; 2] ;$
$\quad [3; 4; 5]\ ] = [\ [0; 6; 2] ;$
$\quad [3; 4; 5]\ ]$

———End of Multidimensional Structures

———End of List Theory

———End of Basic Data Structures

3 Function Theory

We are always allowed to invent new syntax if we explain the rules for its use. A ready source of new syntax is names (identifiers), and the rules for their use are most easily given by some axioms. Usually when we introduce names and axioms we want them for some local purpose. The reader is supposed to understand their scope, the region where they apply, and not use them beyond it. Though the names and axioms are formal (expressions in our formalism), up to now we have introduced them informally by English sentences. But the scope of informally introduced names and axioms is not always clear. In this chapter we present a formal notation for introducing a local name and axiom.

A variable is a name that is introduced for the purpose of instantiation (replacing it). For example, the law $x\times1=x$ uses variable x to tell us that any number multiplied by 1 equals that same number. A constant is a name that is not intended to be instantiated. For example, we might introduce the name pi and the axiom $3.14 < pi < 3.15$, but we do not mean that every number is between 3.14 and 3.15 . Similarly we might introduce the name i and the axiom $i^2=-1$ and we do not want to instantiate i .

The function notation is the formal way of introducing a local variable together with a local axiom to say what expressions can be used to instantiate the variable.

3.0 Functions

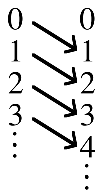
Let v be a name, let D be a bunch of items (possibly using previously introduced names but not using v), and let b be any expression (possibly using previously introduced names and possibly using v). Then

$\langle v: D\to b\rangle$ "map v in D to b ", "local v in D maps to b "

is a function of variable v with domain D and body b . The inclusion $v: D$ is a local axiom within the body b . The brackets $\langle\ \rangle$ indicate the scope of the variable and axiom. For example,

$\langle n: nat\to n+1\rangle$

is the successor function on the natural numbers. Here is a picture of it.



If f is a function, then

Δf "domain of f "

is its domain. The Domain Axiom is

$\Delta\langle v: D\to b\rangle \ = \ D$

We say both that D is the domain of function $\langle v: D\to b\rangle$ and that within the body b , D is the domain of variable v . The range of a function consists of the elements obtained by substituting each element of the domain for the variable in the body. The range of our successor function is $nat+1$.

A function introduces a variable, or synonymously, a parameter. The purpose of the variable is to help express the mapping from domain elements to range elements. The choice of name is irrelevant as long as it is fresh, not already in use for another purpose. The Renaming Axiom says that if v and w are names, and neither v nor w appears in D, and w does not appear in b, then

$$\langle v: D \rightarrow b\rangle \;=\; \langle w: D \rightarrow (\text{substitute } w \text{ for } v \text{ in } b)\rangle$$

The substitution must replace every occurrence of v with w.

If f is a function and x is an element of its domain, then

$f\,x$ "f applied to x" or "f of x"

is the corresponding element of the range. This is function application, and x is the argument. Of course, parentheses may be used around any expression, so we may write $f(x)$ if we want. If either the function or the argument is not simple, we will have to enclose it in parentheses. When there is no danger of confusion, we may write fx without a space between, but when we use multicharacter names, we must put a space between the function and the argument. As an example of application, if $suc = \langle n: nat \rightarrow n+1\rangle$, then

$$suc\ 3 \;=\; \langle n: nat \rightarrow n+1\rangle\ 3 \;=\; 3+1 \;=\; 4$$

Here is the Application Axiom. If element $x: D$, then

$$\langle v: D \rightarrow b\rangle\ x \;=\; (\text{substitute } x \text{ for } v \text{ in } b)$$

Operators and functions are similar; just as we apply operator $-$ to operand x to get $-x$, we apply function f to argument x to get $f\,x$.

A function of more than one variable is a function whose body is a function. Here are two examples.

$$max \;=\; \langle x: xrat \rightarrow \langle y: xrat \rightarrow \textbf{if } x \ge y \textbf{ then } x \textbf{ else } y\rangle\rangle$$
$$min \;=\; \langle x: xrat \rightarrow \langle y: xrat \rightarrow \textbf{if } x \le y \textbf{ then } x \textbf{ else } y\rangle\rangle$$

If we apply max to an argument we obtain a function of one variable,

$$max\ 3 \;=\; \langle y: xrat \rightarrow \textbf{if } 3 \ge y \textbf{ then } 3 \textbf{ else } y\rangle$$

which can be applied to an argument to obtain a number.

$$max\ 3\ 5 \;=\; 5$$

A predicate is a function whose body is a boolean expression. Two examples are

$$even \;=\; \langle i: int \rightarrow i/2: int\rangle$$
$$odd \;=\; \langle i: int \rightarrow \neg\ i/2: int\rangle$$

A relation is a function whose body is a predicate. Here is an example:

$$divides \;=\; \langle n: nat+1 \rightarrow \langle i: int \rightarrow i/n: int\rangle\rangle$$
$$divides\ 2 \;=\; even$$
$$divides\ 2\ 3 \;=\; \bot$$

One more operation on functions is selective union. If f and g are functions, then

$f \mid g$ "f otherwise g", "the selective union of f and g"

is a function that behaves like f when applied to an argument in the domain of f, and otherwise behaves like g. The axioms are

$$\Delta(f \mid g) \;=\; \Delta f, \Delta g$$
$$(f \mid g)\ x \;=\; \textbf{if } x: \Delta f \textbf{ then } f\,x \textbf{ else } g\,x$$

All the rules of proof apply to the body of a function with the additional local axiom that the new variable is an element of the domain.

3.0.0 Abbreviated Function Notations

We allow some variations in the notation for functions partly for the sake of convenience and partly for the sake of tradition. The first variation is to group the introduction of variables. For example,

$\langle x, y\text{: } xrat \rightarrow$ **if** $x \ge y$ **then** x **else** $y\rangle$

is an abbreviation for the *max* function seen earlier.

We may omit the domain of a function (and preceding colon) if the surrounding explanation supplies it. For example, the successor function may be written $\langle n \rightarrow n+1\rangle$ in a context where it is understood that the domain is *nat* .

We may omit the variable (and following colon) when the body of a function does not use it. In this case, we also omit the scope brackets $\langle\ \rangle$. For example, $2 \rightarrow 3$ is a function that maps 2 to 3 , which we could have written $\langle n\text{: } 2 \rightarrow 3\rangle$ with an unused variable.

Some people refer to any expression as a function of its variables. For example, they might write

$x+3$

and say it is a function of x . They omit the formal variable and domain introduction, supplying them informally. There are problems with this abbreviation. One problem is that there may be variables that don't appear in the expression. For example,

$\langle x\text{: } int \rightarrow \langle y\text{: } int \rightarrow x+3\rangle\rangle$

which introduces two variables, would have the same abbreviation as

$\langle x\text{: } int \rightarrow x+3\rangle$

Another problem is that there is no precise indication of the scope of the variable(s). And another is that we do not know the order of the variable introductions, so we cannot apply such an abbreviated function to arguments. We consider this abbreviation to be too much, and we will not use it. We point it out only because it is common terminology, and to show that the variables we introduced informally in earlier chapters are the same as the variables we introduce formally in functions.

End of Abbreviated Function Notations

3.0.1 Scope and Substitution

A variable is <u>local</u> to an expression if its introduction is inside the expression (and therefore formal). A variable is <u>nonlocal</u> to an expression if its introduction is outside the expression (whether formal or informal). The words "local" and "nonlocal" are used relative to a particular expression or subexpression.

If we always use fresh names for our local variables, then a substitution replaces all occurrences of a variable. But if we reuse a name, we need to be more careful. Here is an example in which the gaps represent uninteresting parts.

$\langle x \rightarrow \quad x \quad \langle x \rightarrow \quad x \quad \rangle \quad x \quad \rangle\ 3$

Variable x is introduced twice: it is reintroduced in the inner scope even though it was already introduced in the outer scope. Inside the inner scope, the x is the one introduced in the inner scope. The outer scope is a function, which is being applied to argument 3 . Assuming 3 is in its domain, the Application Axiom says that this expression is equal to one obtained by substituting 3 for x . The intention is to substitute 3 for the x introduced by this function, the outer scope, not the one introduced in the inner scope. The result is

$= \quad (\quad 3 \quad \langle x \rightarrow \quad x \quad \rangle \quad 3 \quad)$

Here is a worse example. Suppose x is a nonlocal variable, and we reintroduce it in an inner scope.

$$\langle y\to \quad x \quad y \quad \langle x\to \quad x \quad y \quad \rangle \quad x \quad y \quad \rangle\, x$$

The Application Axiom tells us to substitute x for all occurrences of y. All three uses of y are the variable introduced by the outer scope, so all three must be replaced by the nonlocal x used as argument. But that will place a nonlocal x inside a scope that reintroduces x, making it look local. Before we substitute, we must use the Renaming Axiom for the inner scope. Choosing fresh name z, we get

$$= \quad \langle y\to \quad x \quad y \quad \langle z\to \quad z \quad y \quad \rangle \quad x \quad y \quad \rangle\, x$$

by renaming, and then substitution gives

$$= \quad (\quad x \quad x \quad \langle z\to \quad z \quad x \quad \rangle \quad x \quad x \quad)$$

The Application Axiom (for element $x\colon D$)

$$\langle v\colon D\to b\rangle\, x \;=\; (\text{substitute } x \text{ for } v \text{ in } b)$$

provides us with a formal notation for substitution. It is one of only two axioms (this one concerns variable introduction; the other, in Chapter 5, concerns variable removal) that we express informally, because formalizing it is equivalent to writing a program to perform substitution. The Renaming Axiom can be written formally as follows:

$$\langle v\colon D\to b\rangle \;=\; \langle w\colon D\to\langle v\colon D\to b\rangle\, w\rangle$$

And it needn't be an axiom, because it is an instance of the Axiom of Extension

$$f \;=\; \langle w\colon \Delta f\to f\, w\rangle$$

When the domain is obvious, or when it is obvious that we intend a domain that includes x, we write $\langle v\to b\rangle x$ for "replace v in b by x". For example, applying each side of the Renaming Axiom to argument x

$$\langle v\to b\rangle x \;=\; \langle w\to\langle v\to b\rangle w\rangle x$$

says that replacing v by x is the same as replacing v by w and then replacing w by x.

End of Scope and Substitution

End of Functions

3.1 Quantifiers

A quantifier is a one-operand prefix operator that applies to functions. Any two-operand symmetric associative operator can be used to define a quantifier. Here are four examples: the operators $\wedge\ \vee\ +\ \times$ are used to define, respectively, the quantifiers $\forall\ \exists\ \Sigma\ \Pi$. If p is a predicate, then universal quantification $\forall p$ is the boolean result of applying p to all its domain elements and conjoining all the results. Similarly, existential quantification $\exists p$ is the boolean result of applying p to all its domain elements and disjoining all the results. If f is a function with a numeric result, then Σf is the numeric result of applying f to all its domain elements and adding up all the results; and Πf is the numeric result of applying f to all its domain elements and multiplying together all the results. Here are four examples.

$\forall\langle r : rat\to r<0 \vee r=0 \vee r>0\rangle$	"for all r in rat ..."
$\exists\langle n\colon nat\to n=0\rangle$	"there exists n in nat such that ..."
$\Sigma\langle n\colon nat+1\to 1/2^n\rangle$	"the sum, for n in $nat+1$, of ..."
$\Pi\langle n\colon nat+1\to(4\times n^2)/(4\times n^2-1)\rangle$	"the product, for n in $nat+1$, of ..."

For the sake of tradition and convenience, we allow two abbreviated quantifier notations. First, we allow the scope brackets $\langle\ \rangle$ following a quantifier to be omitted; now we have to change the arrow to a raised dot to avoid ambiguity. For example we write

$$\forall r\colon rat\cdot r<0 \vee r=0 \vee r>0$$

$$\Sigma n\colon nat+1\cdot 1/2^n$$

This abbreviated quantifier notation makes the scope of variables less clear, and it complicates the precedence rules, but the mathematical tradition is strong, and so we will use it. Second, we can group the variables in a repeated quantification. In place of

$$\forall x\colon rat\cdot \forall y\colon rat\cdot x = y{+}1 \;\Rightarrow\; x > y$$

we can write

$$\forall x, y\colon rat\cdot x = y{+}1 \;\Rightarrow\; x > y$$

and in place of

$$\Sigma n\colon 0,..10\cdot \Sigma m\colon 0,..10\cdot n{\times}m$$

we can write

$$\Sigma n, m\colon 0,..10\cdot n{\times}m$$

The axioms for these quantifiers fall into two patterns, depending on whether the operator on which it is based is idempotent. The axioms are as follows (v is a name, A and B are bunches, b is a boolean expression, n is a number expression, and x is an element).

$$\forall v\colon null\cdot b \;=\; \top$$
$$\forall v\colon x\cdot b \;=\; \langle v\colon x\rightarrow b\rangle\, x$$
$$\forall v\colon A,B\cdot b \;=\; (\forall v\colon A\cdot b) \wedge (\forall v\colon B\cdot b)$$

$$\exists v\colon null\cdot b \;=\; \bot$$
$$\exists v\colon x\cdot b \;=\; \langle v\colon x\rightarrow b\rangle\, x$$
$$\exists v\colon A,B\cdot b \;=\; (\exists v\colon A\cdot b) \vee (\exists v\colon B\cdot b)$$

$$\Sigma v\colon null\cdot n \;=\; 0$$
$$\Sigma v\colon x\cdot n \;=\; \langle v\colon x\rightarrow n\rangle\, x$$
$$(\Sigma v\colon A,B\cdot n) + (\Sigma v\colon A`B\cdot n) \;=\; (\Sigma v\colon A\cdot n) + (\Sigma v\colon B\cdot n)$$

$$\Pi v\colon null\cdot n \;=\; 1$$
$$\Pi v\colon x\cdot n \;=\; \langle v\colon x\rightarrow n\rangle x$$
$$(\Pi v\colon A,B\cdot n) \times (\Pi v\colon A`B\cdot n) \;=\; (\Pi v\colon A\cdot n) \times (\Pi v\colon B\cdot n)$$

Care is required when translating from the English words “all” and “some” to the formal notations $\forall$ and $\exists$. For example, the statement “All is not lost.” should not be translated as $\forall x\cdot \neg\, lost\; x$, but as $\exists x\cdot \neg\, lost\; x$ or as $\neg\forall x\cdot lost\; x$ or as $\neg\forall lost$. Notice that when a quantifier is applied to a function with an empty domain, it gives the identity element of the operator it is based on. It is probably not a surprise to find that the sum of no numbers is 0 , but it may surprise you to learn that the product of no numbers is 1 . You probably agree that there is not an element in the empty domain with property b (no matter what b is), and so existential quantification over an empty domain gives the result you expect. You may find it harder to accept that all elements in the empty domain have property b , but look at it this way: to deny it is to say that there is an element in the empty domain without property b . Since there isn't any element in the empty domain, there isn't one without property b , so all (zero) elements have the property.

We can also form quantifiers from functions that we define ourselves. For example, functions min and max are two-operand symmetric associative idempotent functions, so we can define corresponding quantifiers MIN and MAX as follows.

$$MIN\; v\colon null\cdot n \;=\; \infty$$
$$MIN\; v\colon x\cdot n \;=\; \langle v\colon x\rightarrow n\rangle\, x$$
$$MIN\; v\colon A,B\cdot n \;=\; min\; (MIN\; v\colon A\cdot n)\; (MIN\; v\colon B\cdot n)$$

$MAX\ v: null\cdot n \ = \ -\infty$
$MAX\ v: x\cdot n \ = \ \langle v: x\to n\rangle\ x$
$MAX\ v: A,B\cdot n \ = \ max\ (MAX\ v: A\cdot n)\ (MAX\ v: B\cdot n)$

Our final quantifier applies to predicates. The solution quantifier § ("solutions of", "those") gives the bunch of solutions of a predicate. Here are the axioms.

$\S v: null\cdot b \ = \ null$
$\S v: x\cdot b \ = \ \textbf{if}\ \langle v: x\to b\rangle\ x\ \textbf{then}\ x\ \textbf{else}\ null$
$\S v: A,B\cdot b \ = \ (\S v: A\cdot b), (\S v: B\cdot b)$

We have all practiced solving equations, and we are comfortable with

$\S i: int\cdot i^2 = 4 \ = \ -2, 2$ "those i in int such that ... "

Equations are just a special case of boolean expression; we can just as well talk about the solutions of any predicate. For example,

$\S n: nat\cdot n<3 \ = \ 0,..3$

Once again, for tradition and convenience, when the solution quantifier is used within a set, we can abbreviate by omitting the quantifier. For example, instead of writing $\{\S n: nat\cdot n<3\}$, we might write $\{n: nat\cdot n<3\}$, which is a standard notation for sets.

There are further axioms to say how each quantifier behaves when the domain is a result of the § quantifier; they are listed at the back of the book, together with other laws concerning quantification. These laws are used again and again during programming; they must be studied until they are all familiar. Some of them can be written in a nicer, though less traditional, way. For example, the Specialization and Generalization laws at the back of the book say that if $x: D$,

$\forall v: D\cdot b \ \Rightarrow\ \langle v: D\to b\rangle\ x$
$\langle v: D\to b\rangle\ x \ \Rightarrow\ \exists v: D\cdot b$

Together they can be written as follows: if $x: \Delta f$

$\forall f \ \Rightarrow\ f\,x \ \Rightarrow\ \exists f$

If f results in $\top$ for all its domain elements, then f results in $\top$ for domain element x . And if f results in $\top$ for domain element x , then there is a domain element for which f results in $\top$.

The One-Point Laws say that if $x: D$, and v does not appear in x , then

$\forall v: D\cdot v=x \Rightarrow b \ = \ \langle v: D\to b\rangle\ x$
$\exists v: D\cdot v=x \wedge b \ = \ \langle v: D\to b\rangle\ x$

For instance, in the universal quantification $\forall n: nat\cdot n=3 \Rightarrow n<10$, we see an implication whose antecedent equates the variable to an element. The One-Point Law says this can be simplified by getting rid of the quantifier and antecedent, keeping just the consequent, but replacing the variable by the element. So we get $3<10$, which can be further simplified to $\top$. In an existential quantification, we need a conjunct equating the variable to an element, and then we can make the same simplification. For example, $\exists n: nat\cdot n=3 \wedge n<10$ becomes $3<10$, which can be further simplified to $\top$. If P is a predicate that does not mention nonlocal variable x , and element y is in the domain of P , then the following are all equivalent:

$\forall x: \Delta P\cdot x=y \Rightarrow Px$
$= \ \exists x: \Delta P\cdot x=y \wedge Px$
$= \ \langle x: \Delta P\to Px\rangle\ y$
$= \ Py$

Some of the laws may be a little surprising; for example, we can prove

$MIN\ n: nat\cdot 1/(n+1) \ = \ 0$

even though 0 is never a result of the function $\langle n: nat\to 1/(n+1)\rangle$.

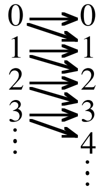
End of Quantifiers

3.2 Function Fine Points optional

Consider a function in which the body is a bunch: each element of the domain is mapped to zero or more elements of the range. For example,

$\langle n: nat \to n, n+1 \rangle$

maps each natural number to two natural numbers.



Application works as usual:

$\langle n: nat \to n, n+1 \rangle\ 3 \ = \ 3, 4$

A function that sometimes produces no result is called "partial". A function that always produces at least one result is called "total". A function that always produces at most one result is called "deterministic". A function that sometimes produces more than one result is called "nondeterministic". The function $\langle n: nat \to 0,..n \rangle$ is both partial and nondeterministic.

A union of functions applied to an argument gives the union of the results:

$(f, g)\ x \ = \ fx, gx$

A function applied to a union of arguments gives the union of the results:

$f\ null \ = \ null$

$f(A, B) \ = \ f A, f B$

$f(\S g) \ = \ \S y: f(\Delta g) \cdot \exists x: \Delta g \cdot fx{=}y \wedge gx$

In other words, function application distributes over bunch union. The range of function f is $f(\Delta f)$.

In general, we cannot apply a function to a non-elementary bunch using the Application Law. For example, if we define $double = \langle n: nat \to n{+}n \rangle$ we can say

$\quad double\ (2, 3)$ this step is right
$= \ double\ 2, double\ 3$
$= \ 4, 6$

but we cannot say

$\quad double\ (2, 3)$ this step is wrong
$= \ (2, 3) + (2, 3)$
$= \ 4, 5, 6$

Suppose we really do want to apply a function to a collection of items, for example to report if there are too many items in the collection. Then the collection must be packaged as a set to make it an elementary argument.

If the body of a function uses its variable exactly once, and in a distributing context, then the function can be applied to a non-elementary argument because the result will be the same as would be obtained by distribution. For example,

$\quad \langle n: nat \to n{\times}2 \rangle\ (2, 3)$ this step is ok
$= \ (2, 3){\times}2$
$= \ 4, 6$

3.2.0 Function Inclusion and Equality

optional

A function f is included in a function g according to the Function Inclusion Law:

$$f\text{: } g \;\;=\;\; \Delta g\text{: } \Delta f \;\wedge\; \forall x\text{: } \Delta g\cdot fx\text{: } gx$$

Using it both ways round, we find function equality is as follows:

$$f = g \;\;=\;\; \Delta f = \Delta g \;\wedge\; \forall x\text{: } \Delta f\cdot fx = gx$$

We now prove $suc\text{: } nat\to nat$. Function suc was defined earlier as $suc = \langle n\text{: } nat\to n+1\rangle$. Function $nat\to nat$ is an abbreviation of $\langle n\text{: } nat\to nat\rangle$, which has an unused variable. It is a nondeterministic function whose result, for each element of its domain nat , is the bunch nat . It is also the bunch of all functions whose domain includes nat and whose result is included in nat .

$$suc\text{: } nat\to nat \qquad \text{use Function Inclusion Law}$$
$$=\; nat\text{: } \Delta suc \;\wedge\; \forall n\text{: } nat\cdot suc\ n\text{: } nat \qquad \text{definition of } suc$$
$$=\; nat\text{: } nat \;\wedge\; \forall n\text{: } nat\cdot n+1\text{: } nat \qquad \text{reflexivity, and } nat \text{ construction axiom}$$
$$=\; \top$$

We can prove similar inclusions about other functions defined in the first section of this chapter.

$$max\text{: } xrat\to xrat\to xrat$$
$$min\text{: } xrat\to xrat\to xrat$$
$$even\text{: } int\to bool$$
$$odd\text{: } int\to bool$$
$$divides\text{: } (nat+1)\to int\to bool$$

And, more generally,

$$f\text{: } A\to B \;\;=\;\; A\text{: } \Delta f \;\wedge\; fA\text{: } B$$

End of Function Inclusion and Equality

We earlier defined suc by the axiom

$$suc = \langle n\text{: } nat\to n+1\rangle$$

This equation can be written instead as

$$\Delta suc = nat \;\wedge\; \forall n\text{: } nat\cdot suc\ n = n+1$$

We could have defined suc by the weaker axiom

$$nat\text{: } \Delta suc \;\wedge\; \forall n\text{: } nat\cdot suc\ n = n+1$$

which is almost as useful in practice, and allows suc to be extended to a larger domain later, if desired. A similar comment holds for max , min , $even$, odd , and $divides$.

3.2.1 Higher-Order Functions

optional

A higher-order function is a function whose parameter is function-valued, and whose argument must therefore be a function. If $g\text{: } A\to B$, then

$$\langle f\text{: } (A\to B)\to\ \ldots f \ldots\rangle\ g$$

applies a higher-order function to a function argument. A parameter stands for an element of the domain, and the Application Law requires the argument to be an element of the domain, but functions are not elements. Therefore we consider a higher-order function applied to an argument, as written above, to be an abbreviation for

$$\langle f\text{: } \natural(A\to B)\to\ \ldots \sim\! f \ldots\rangle\ \{g\}$$

The power operator $\natural$ and the set brackets $\{\ \}$ just make the parameter and argument into elements, as required, and the content operator $\sim$ then removes the set structure.

Here is a predicate whose parameter is a function.

$$\langle f\text{: } ((0,..10)\to int)\to \forall n\text{: } 0,..10\cdot even\ (f\ n)\rangle$$

Let us call this predicate $check$. An argument for $check$ must be a function whose domain

includes 0,..10 because *check* will be applying its argument to all elements in 0,..10 . When an argument for *check* is applied to the first 10 natural numbers, the results must be included in *int* because they will be tested for evenness. An argument for *check* may have a larger domain (extra domain elements will be ignored), and it may have a smaller range. If $A: B$ and $f: B \rightarrow C$ and $C: D$ then $f: A \rightarrow D$. Therefore $suc: (0,..10) \rightarrow int$. We can apply *check* to *suc* and the result is $\bot$.

End of Higher-Order Functions

3.2.2 Function Composition

optional

Let f and g be functions such that f is not in the domain of g ($\neg f: \Delta g$). Then $g\,f$ is the composition of g and f, defined by the Function Composition Axioms:

$$\Delta(g\,f) = \S x: \Delta f \cdot f x: \Delta g$$
$$(g\,f)\,x = g\,(f\,x)$$

For example, since *suc* is not in the domain of *even* ,

$$\Delta(even\ suc) = \S x: \Delta suc \cdot suc\ x: \Delta even = \S x: nat \cdot x{+}1: int = nat$$
$$(even\ suc)\ 3 = even\ (suc\ 3) = even\ 4 = \top$$

Suppose $x, y: int$ and $f, g: int \rightarrow int$ and $h: int \rightarrow int \rightarrow int$. Then

	$h\,f\,x\,g\,y$	juxtaposition is left-associative
=	$(((h\,f)\,x)\,g)\,y$	use function composition on $h\,f$
=	$((h\,(f\,x))\,g)\,y$	use function composition on $(h\,(f\,x))\,g$
=	$(h\,(f\,x))\,(g\,y)$	drop superfluous parentheses
=	$h\,(f\,x)\,(g\,y)$	

The Composition Axiom says that we can write complicated combinations of functions and arguments without parentheses. They sort themselves out properly according to their functionality. (This is called "Polish prefix" notation.)

Composition and application are closely related. Suppose $f: A \rightarrow B$ and $g: B \rightarrow C$ and $\neg f: \Delta g$ so that g can be composed with f . Although g cannot be applied to f , we can change g into a function $g': (A \rightarrow B) \rightarrow (A \rightarrow C)$ that can be applied to f to obtain the same result as composition: $g'\,f = g\,f$. Here is an example. Define

$$double = \langle n: nat \rightarrow n{+}n \rangle$$

We can compose *double* with *suc* .

	$(double\ suc)\ 3$	use composition
=	$double\ (suc\ 3)$	apply *double* to *suc* 3
=	$suc\ 3 + suc\ 3$	

From *double* we can form a new function

$$double' = \langle f \rightarrow \langle n \rightarrow f\,n + f\,n \rangle\rangle$$

which can be applied to *suc*

$$(double'\ suc)\ 3 = \langle n \rightarrow suc\ n + suc\ n \rangle\ 3 = suc\ 3 + suc\ 3$$

to obtain the same result as before. This close correspondence has led people to take a notational shortcut: they go ahead and apply *double* to *suc* even though it does not apply, then distribute the next argument to all occurrences of *suc* . Beginning with

$(double\ suc)\ 3$	they "apply" *double* to *suc*
$(suc + suc)\ 3$	then distribute 3 to all occurrences of *suc*
$suc\ 3 + suc\ 3$	and get the right answer.

As in this example, the shortcut usually works, but beware: it can sometimes lead to inconsistencies. (The word "apposition" has been suggested as a contraction of "application" and "composition", and it perfectly describes the notation, too!)

Like application, composition distributes over bunch union.

$f(g, h) = fg, fh$
$(f, g)\, h = fh, gh$

Operators and functions are similar; each applies to its operands to produce a result. Just as we compose functions, we can compose operators, and we can compose an operator with a function. For example, we can compose $-$ with any function f that produces a number to obtain a new function.

$(-f)\, x = -(f\, x)$

In particular,

$(-suc)\, 3 = -(suc\, 3) = -4$

Similarly if p is a predicate, then

$(\neg p)\, x = \neg(p\, x)$

We can compose $\neg$ with $even$ to obtain odd again.

$\neg even = odd$

We can write the Duality Laws this way:

$\neg\forall f = \exists\neg f$
$\neg\exists f = \forall\neg f$

It would be even nicer if we could write them this way:

$\neg\forall = \exists\neg$
$\neg\exists = \forall\neg$

End of Function Composition

End of Function Fine Points

3.3 List as Function

For some purposes, a list can be regarded as a kind of function; the domain of list L is $0,..\#L$. And conversely, a function whose domain is $0,..n$ for some natural n , and whose body is an item, can sometimes be regarded as a kind of list. Indexing a list is the same as function application, and the same notation $L\, n$ is used. List composition is the same as function composition, and the same notation $L\, M$ is used. It is handy, and not harmful, to mix operators and lists and functions in a composition. For example,

$-[3; 5; 2] = [-3; -5; -2]$
$suc\, [3; 5; 2] = [4; 6; 3]$

We can also mix lists and functions in a selective union. With function $1 \rightarrow 21$ as left operand, and list $[10; 11; 12]$ as right operand, we get

$1 \rightarrow 21 \mid [10; 11; 12] = [10; 21; 12]$

just as we defined it for lists.

We can apply quantifiers to lists. Since list L corresponds to the function $\langle n: 0,..\#L \rightarrow Ln \rangle$, then ΣL can be used to mean $\Sigma n: 0,..\#L \cdot Ln$, and conveniently expresses the sum of the items of the list.

In some respects, lists and functions differ. Catenation and length apply to lists, not to functions. Order is defined for lists, not for functions. List inclusion and function inclusion do not coincide.

End of List as Function

3.4 Limits and Reals optional

Let $f: nat \to rat$ so that $f0; f1; f2; ...$ is a sequence of rationals. The limit of the function (limit of the sequence) is expressed as $LIM\ f$. For example,

$$LIM\ n: nat \cdot (1 + 1/n)^n$$

is the base of the natural logarithms, often denoted e, approximately equal to 2.718281828459. We define the LIM quantifier by the following Limit Axiom:

$$(MAX\ m\cdot MIN\ n\cdot f(m+n)) \le (LIM\ f) \le (MIN\ m\cdot MAX\ n\cdot f(m+n))$$

with all domains being nat. This axiom gives a lower bound and an upper bound for $LIM\ f$. When those bounds are equal, the Limit Axiom tells us $LIM\ f$ exactly. For example,

$$LIM\ n\cdot 1/(n+1) = 0$$

For some functions, the Limit Axiom tells us a little less. For example,

$$-1 \le (LIM\ n\cdot (-1)^n) \le 1$$

In general,

$$(MIN\ f) \le (LIM\ f) \le (MAX\ f)$$

For monotonic (nondecreasing) f, $LIM\ f = MAX\ f$. For antimonotonic (nonincreasing) f, $LIM\ f = MIN\ f$.

Now we can define the extended real numbers.

$$x: xreal = \exists f: nat \to rat\cdot x = LIM\ f$$

We take the limits of all functions with domain nat and range rat. Now the reals:

$$r: real = r: xreal \wedge -\infty < r < \infty$$

Exploration of this definition is a rich subject called real analysis, and we leave it to other books.

Let $p: nat \to bool$ so that p is a predicate and $p0; p1; p2; ...$ is a sequence of booleans. The limit of predicate p is defined by the axiom

$$\exists m\cdot \forall n\cdot p(m+n) \Longrightarrow LIM\ p \Longrightarrow \forall m\cdot \exists n\cdot p(m+n)$$

with all domains being nat. The limit axiom for predicates is very similar to the limit axiom for numeric functions. One way to understand it is to break it into two separate implications, and change the second variable as follows.

$$\exists m\cdot \forall i\cdot i \ge m \Rightarrow pi \Longrightarrow LIM\ p$$

$$\exists m\cdot \forall i\cdot i \ge m \Rightarrow \neg pi \Longrightarrow \neg LIM\ p$$

For any particular assignment of values to (nonlocal) variables, the first implication says that $LIM\ p$ is $\top$ if there is a point in the sequence $p0\ p1\ p2\ ...$ past which p is always $\top$, and the second implication says that $LIM\ p$ is $\bot$ if there is a point in the sequence past which p is always $\bot$. For example,

$$\neg LIM\ n\cdot 1/(n+1) = 0$$

Even though the limit of $1/(n+1)$ is 0, the limit of $1/(n+1) = 0$ is $\bot$.

If, for some particular assignment of values to variables, the sequence never settles on one boolean value, then the axiom does not determine the value of $LIM\ p$ for that assignment of values.

End of Limits and Reals

The purpose of a function is to introduce a local variable. But we must remember that any expression talks about its nonlocal variables. For example,

$$\exists n: nat\cdot x = 2 \times n$$

says that x is an even natural. The local variable n, which could just as well have been m or any other name, is used to help say that x is an even natural. The expression is talking about x, not about n.

End of Function Theory

4 Program Theory

We begin with a very simple model of computation. A computer has a memory, and we can observe its contents, or state. Our input to a computation is to provide an initial state, or prestate. After a time, the output from the computation is the final state, or poststate. Although the memory contents may physically be a string of bits, we can consider it to be a string of any items; we only need to group the bits and view them through a code. A state σ (sigma) may, for example, be given by

$\sigma = -2; 15; "A"; 3.14$

The indexes of the items in a state are usually called "addresses". The bunch of possible states is called the state space. For example, the state space might be

int; (0,..20); *char*; *rat*

If the memory is in state σ , then the items in memory are σ_0 , σ_1 , σ_2 , and so on. Instead of using addresses, we find it much more convenient to refer to items in memory by distinct names such as i , n , c , and x . Names that are used to refer to items in the state are called state variables. We must always say what the state variables are and what their domains are, but we do not bother to say which address a state variable corresponds to. Formally, there is a function *address* to say where each state variable is. For example,

$x = \sigma_{address\ "x"}$

A state is then an assignment of values to state variables.

Our example state space in the previous paragraph is infinite, and this is unrealistic; any physical memory is finite. We allow this deviation from reality as a simplification; the theory of integers is simpler than the theory of integers modulo 2^{32} , and the theory of rational numbers is much simpler than the theory of 32-bit floating-point numbers. In the design of any theory we must decide which aspects of the world to consider and which to leave to other theories. We are free to develop and use more complicated theories when necessary, but we will have difficulties enough without considering the finite limitations of a physical memory.

To begin this chapter, we consider only the prestate and poststate of memory to be of importance. Later in this chapter we will consider execution time, and changing space requirements, and in Chapter 9 we will consider intermediate states and communication during the course of a computation. But to begin we consider only an initial input and a final output. The question of termination of a computation is a question of execution time; termination just means that the execution time is finite. In the case of a terminating computation, the final output is available after a finite time; in the case of a nonterminating computation, the final output is never available, or to say the same thing differently, it is available at time infinity. All further discussion of termination is postponed until we discuss execution time.

4.0 Specifications

A specification is a boolean expression whose variables represent quantities of interest. We are specifying computer behavior, and (for now) the quantities of interest are the prestate σ and the poststate σ' . We provide a prestate as input. A computation then computes and delivers a poststate as output. To satisfy a specification, a computation must deliver a satisfactory poststate. In other words, the given prestate and the computed poststate must make the specification true. We have an implementation when the specification describes (is true of) every computation. For a specification to be implementable, there must be at least one satisfactory output state for each input state.

Here are four definitions based on the number of satisfactory outputs for each input.

Specification S is unsatisfiable for prestate σ : $¢(§\sigma'\cdot S) < 1$
Specification S is satisfiable for prestate σ : $¢(§\sigma'\cdot S) \ge 1$
Specification S is deterministic for prestate σ : $¢(§\sigma'\cdot S) \le 1$
Specification S is nondeterministic for prestate σ : $¢(§\sigma'\cdot S) > 1$

We can rewrite the definition of satisfiable as follows:

Specification S is satisfiable for prestate σ : $\exists\sigma'\cdot S$

And finally,

Specification S is implementable: $\forall\sigma\cdot \exists\sigma'\cdot S$

For convenience, we prefer to write specifications in the initial values x , y , ... and final values x' , y' , ... of some state variables (we make no typographic distinction between a state variable and its initial value). Here is an example. Suppose there are two state variables x and y each with domain *int* . Then

$x' = x+1 \land y' = y$

specifies the behavior of a computer that increases the value of x by 1 and leaves y unchanged. Let us check that it is implementable. We replace $\forall\sigma$ by either $\forall x, y$ or $\forall y, x$ and we replace $\exists\sigma'$ by either $\exists x', y'$ or $\exists y', x'$; according to the Commutative Laws, the order does not matter. We find

$\forall x, y\cdot \exists x', y'\cdot x' = x+1 \land y' = y$ — One-Point Law twice
$= \forall x, y\cdot \top$ — Identity Law twice
$= \top$

The specification is implementable. It is also deterministic for each prestate.

In the same state variables, here is a second specification.

$x' > x$

This specification is satisfied by a computation that increases x by any amount; it may leave y unchanged or may change it to any integer. This specification is nondeterministic for each initial state. Computer behavior satisfying the earlier specification $x' = x+1 \land y' = y$ also satisfies this one, but there are many ways to satisfy this one that do not satisfy the earlier one. In general, weaker specifications are easier to implement; stronger specifications are harder to implement.

At one extreme, we have the specification $\top$; it is the easiest specification to implement because all computer behavior satisfies it. At the other extreme is the specification $\bot$, which is not satisfied by any computer behavior. But $\bot$ is not the only unimplementable specification. Here is another.

$x\ge 0 \land y'=0$

If the initial value of x is nonnegative, the specification can be satisfied by setting variable y to 0 . But if the initial value of x is negative, there is no way to satisfy the specification. Perhaps the specifier has no intention of providing a negative input; in that case, the specifier should have written

$x\ge 0 \Rightarrow y'=0$

For nonnegative initial x , this specification still requires variable y to be assigned 0 . If we never provide a negative value for x then we don't care what would happen if we did. That's what this specification says: for negative x any result is satisfactory. It allows an implementer to provide an error indication when x is initially negative. If we want a particular error indication, we can strengthen the specification to say so. We can describe the acceptable inputs as $x\ge 0$, but not the computer behavior. We can describe the acceptable inputs and the computer behavior together as $x\ge 0 \land (x\ge 0 \Rightarrow y'=0)$, which can be simplified to $x\ge 0 \land y'=0$. But $x\ge 0 \land y'=0$ cannot be implemented as computer behavior because a computer cannot control its inputs.

There is an unfortunate clash between mathematical terminology and computing terminology that we have to live with. In mathematical terminology, a variable is something that can be instantiated, and a constant is something that cannot be instantiated. In computing terminology, a variable is something that can change state, and a constant is something that cannot change state. A computing variable is also known as a "state variable", and a computing constant is also known as a "state constant". A state variable x corresponds to two mathematical variables x and x'. A state constant is a single mathematical variable; it is there for instantiation, and it does not change state.

4.0.0 Specification Notations

For our specification language we will not be definitive or restrictive; we allow any well understood notations. Often this will include notations from the application area. When it helps to make a specification clearer and more understandable, a new notation may be invented and defined by new axioms.

In addition to the notations already presented, we add two more.

$$\begin{aligned} ok &= \sigma'=\sigma \\ &= x'=x \ \land\ y'=y \ \land\ \ldots \end{aligned}$$

$$\begin{aligned} x{:=}\ e &= \sigma' = \sigma \triangleleft \mathit{address}\ \text{"x"} \triangleright e \\ &= x'=e \ \land\ y'=y \ \land\ \ldots \end{aligned}$$

The notation ok specifies that the final values of all variables equal the corresponding initial values. A computer can satisfy this specification by doing nothing. The assignment $x{:=}\ e$ is pronounced " x is assigned e ", or " x gets e ", or " x becomes e ". In the assignment notation, x is any unprimed state variable and e is any unprimed expression in the domain of x. For example, in integer variables x and y,

$$x{:=}\ x+y \ = \ x'=x+y \ \land\ y'=y$$

So $x{:=}\ x+y$ specifies that the final value of x should be the sum of the initial values of x and y, and the value of y should be unchanged.

Specifications are boolean expressions, and they can be combined using any operators of Boolean Theory. If S and R are specifications, then $S \land R$ is a specification that is satisfied by any computation that satisfies both S and R. Similarly, $S \lor R$ is a specification that is satisfied by any computation that satisfies either S or R. Similarly, $\neg S$ is a specification that is satisfied by any computation that does not satisfy S. A particularly useful operator is **if** b **then** S **else** R where b is a boolean expression of the initial state; it can be implemented by a computer that evaluates b, and then, depending on the value of b, behaves according to either S or R. The $\lor$ and **if then else** operators have the nice property that if their operands are implementable, so is the result; the operators $\land$ and $\neg$ do not have that property.

Specifications can also be combined by <u>dependent composition</u>, which describes sequential execution. If S and R are specifications, then $S.R$ is a specification that can be implemented by a computer that first behaves according to S, then behaves according to R, with the final state from S serving as initial state for R. (The symbol for dependent composition is pronounced "dot". This is not the same as the raised dot used in the abbreviated form of quantification.) Dependent composition is defined as follows.

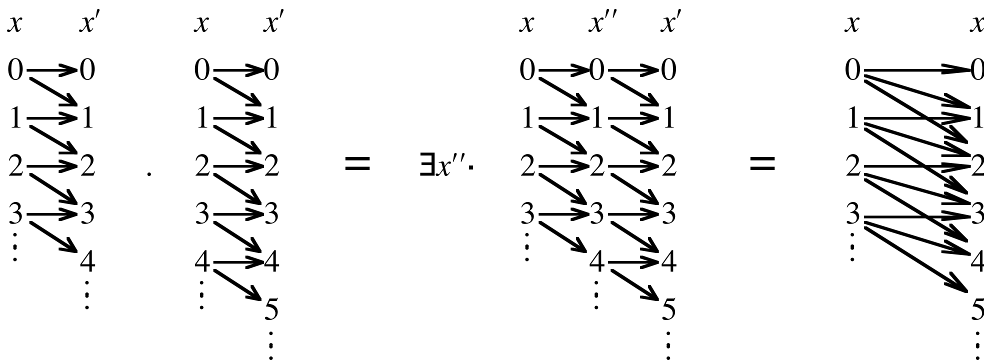
$$\begin{aligned} S.R &= \exists\sigma''\cdot\ \langle\sigma'\rightarrow S\rangle\sigma'' \ \land\ \langle\sigma\rightarrow R\rangle\sigma'' \\ &= \exists x'', y'', \ldots\cdot\ \langle x', y', \ldots\rightarrow S\rangle\, x''\, y''\, \ldots \ \land\ \langle x, y, \ldots\rightarrow R\rangle\, x''\, y''\, \ldots \\ &= \exists x'', y'', \ldots\cdot\ \ (\text{substitute } x'', y'', \ldots \text{ for } x', y', \ldots \text{ in } S) \\ &\qquad \land\ (\text{substitute } x'', y'', \ldots \text{ for } x, y, \ldots \text{ in } R) \end{aligned}$$

Here's an example. In one integer variable x, the specification $x'=x \lor x'=x+1$ says that the final value of x is either the same as the initial value or one greater. Let's compose it with itself.

$$
\begin{array}{lll}
 & x'=x \lor x'=x+1 \ .\ x'=x \lor x'=x+1 & \\
= & \exists x''\cdot\ (x''=x \lor x''=x+1) \land (x'=x'' \lor x'=x''+1) & \text{distribute } \land \text{ over } \lor \\
= & \exists x''\cdot\ x''=x \land x'=x'' \ \lor\ x''=x+1 \land x'=x'' \ \lor\ x''=x \land x'=x''+1 \ \lor\ x''=x+1 \land x'=x''+1 & \\
 & & \text{distribute } \exists \text{ over } \lor \\
= & \quad (\exists x''\cdot x''=x \land x'=x'') \lor (\exists x''\cdot x''=x+1 \land x'=x'') & \\
 & \lor\ (\exists x''\cdot x''=x \land x'=x''+1) \lor (\exists x''\cdot x''=x+1 \land x'=x''+1) & \text{One-Point, 4 times} \\
= & x'=x \lor x'=x+1 \lor x'=x+2 &
\end{array}
$$

If we either leave x alone or add 1 to it, and then again we either leave x alone or add 1 to it, the net result is that we either leave it alone, add 1 to it, or add 2 to it.

Here is a picture of the same example. In the picture, an arrow from a to b means that the specification allows variable x to change value from a to b. We see that if x can change from a to b in the left operand of a dependent composition, and from b to c in the right operand, then it can change from a to c in the result.



We need to be clear on what is meant by (substitute $x'', y'', ...$ for $x', y', ...$ in S) and (substitute $x'', y'', ...$ for $x, y, ...$ in R) in the definition of $S.R$. To begin with, you should not conclude that substitution is impossible since the names S and R do not mention any state variables; presumably S and R stand for, or are equated to, expressions that do mention some state variables. And second, when S or R is an assignment, the assignment notation should be replaced by its equal using mathematical variables x, x', y, y', ... before substitution. Finally, when S or R is a dependent composition, the inner substitutions must be made first. Here is an example, again in integer variables x and y.

$$
\begin{array}{lll}
 & x:= 3.\ y:= x+y & \text{eliminate assignments first} \\
= & x'=3 \land y'=y.\ x'=x \land y'=x+y & \text{then eliminate dependent composition} \\
= & \exists x'', y'': int\cdot x''=3 \land y''=y \ \land\ x'=x'' \land y'=x''+y'' & \text{use One-Point Law twice} \\
= & x'=3 \ \land\ y' = 3+y &
\end{array}
$$

End of Specification Notations

4.0.1 Specification Laws

We have seen some of the following laws before. For specifications P, Q, R, and S, and boolean b,

$$
\begin{array}{ll}
ok.P \ =\ P.ok \ =\ P & \text{Identity Law} \\
P.(Q.R) \ =\ (P.Q).R & \text{Associative Law}
\end{array}
$$

if b **then** P **else** P $=$ P	Idempotent Law
if b **then** P **else** Q $=$ **if** $\neg b$ **then** Q **else** P	Case Reversal Law
P $=$ **if** b **then** $b \Rightarrow P$ **else** $\neg b \Rightarrow P$	Case Creation Law
if b **then** S **else** R $=$ $b{\wedge}S \vee \neg b{\wedge}R$	Case Analysis Law
if b **then** S **else** R $=$ $(b{\Rightarrow}S) \wedge (\neg b{\Rightarrow}R)$	Case Analysis Law
$P{\vee}Q.\ R{\vee}S$ $=$ $(P.\ R) \vee (P.\ S) \vee (Q.\ R) \vee (Q.\ \mathrm{S})$	Distributive Law
(**if** b **then** P **else** Q)$\wedge R$ $=$ **if** b **then** $P{\wedge}R$ **else** $Q{\wedge}R$	Distributive Law
$x{:=}$ **if** b **then** e **else** f $=$ **if** b **then** $x{:=}\ e$ **else** $x{:=}\ f$	Functional-Imperative Law

In the second Distributive Law, we can replace $\wedge$ with any other boolean operator. We can even replace it with dependent composition with a restriction: If b is a boolean expression of the prestate (in unprimed variables),

(**if** b **then** P **else** Q)$.\ R$ $=$ **if** b **then** $(P.\ R)$ **else** $(Q.\ R)$	Distributive Law

And finally, if e is any expression of the prestate (in unprimed variables),

$x{:=}\ e.\ P$ $=$ $\langle x{\rightarrow}P\rangle e$	Substitution Law

The Substitution Law says that an assignment followed by any specification is the same as the specification but with the assigned variable replaced by the assigned expression. Exercise 97 illustrates all the difficult cases, so let us do the exercise. The state variables are x and y .

(a) $x{:=}\ y{+}1.\ \ y'{>}x'$

Since x does not occur in $y'{>}x'$, replacing it is no change.

$=$ $y'{>}x'$

(b) $x{:=}\ x{+}1.\ \ y'{>}x \wedge x'{>}x$

Both occurrences of x in $y'{>}x \wedge x'{>}x$ must be replaced by $x{+}1$.

$=$ $y' > x{+}1 \ \wedge\ x' > x{+}1$

(c) $x{:=}\ y{+}1.\ \ y' = 2{\times}x$

Because multiplication has precedence over addition, we must put parentheses around $y{+}1$ when we substitute it for x in $y' = 2{\times}x$.

$=$ $y' = 2{\times}(y{+}1)$

(d) $x{:=}\ 1.\ \ x{\ge}1 \ \Rightarrow\ \exists x{\cdot}\ y' = 2{\times}x$

In $x{\ge}1 \ \Rightarrow\ \exists x{\cdot}\ y' = 2{\times}x$, the first occurrence of x is nonlocal, and the last occurrence is local. It is the nonlocal x that is being replaced. The local x could have been almost any other name, and probably should have been to avoid any possible confusion.

$=$ $1{\ge}1 \ \Rightarrow\ \exists x{\cdot}\ y' = 2{\times}x$

$=$ $\mathit{even}\ y'$

(e) $x{:=}\ y.\ \ x{\ge}1 \ \Rightarrow\ \exists y{\cdot}\ y' = x{\times}y$

Now we are forced to rename the local y before making the substitution, otherwise we would be placing the nonlocal y in the scope of the local y .

$=$ $x{:=}\ y.\ \ x{\ge}1 \ \Rightarrow\ \exists k{\cdot}\ y' = x{\times}k$

$=$ $y{\ge}1 \ \Rightarrow\ \exists k{\cdot}\ y' = y{\times}k$

(f) $x{:=}\ 1.\ \ ok$

The name ok is defined by the axiom $ok \ =\ x'{=}x \wedge y'{=}y$, so it depends on x .

$=$ $x{:=}\ 1.\ \ x'{=}x \wedge y'{=}y$

$=$ $x'{=}1 \wedge y'{=}y$

(g) $x:= 1.\ y:= 2$

Although x does not appear in $y:= 2$, the answer is not $y:= 2$. We must remember that $y:= 2$ is defined by an axiom, and it depends on x .

$=\quad x:= 1.\ x'=x \land y'=2$

$=\quad x'=1 \land y'=2$

(It is questionable whether $x'=1 \land y'=2$ is a "simplification" of $x:= 1.\ y:= 2$.)

(h) $x:= 1.\ P$ where $P = y:= 2$

This one just combines the points of parts (f) and (g).

$=\quad x'=1 \land y'=2$

(i) $x:= 1.\ y:= 2.\ x:= x+y$

In part (g) we saw that $x:= 1.\ y:= 2 \ = \ x'=1 \land y'=2$. If we use that, we are then faced with a dependent composition $x'=1 \land y'=2.\ x:= x+y$ for which the Substitution Law does not apply. In a sequence of assignments, it is much better to use the Substitution Law from right to left.

$=\quad x:= 1.\ x' = x+2 \ \land\ y'=2$

$=\quad x'=3 \land y'=2$

(j) $x:= 1.$ **if** $y>x$ **then** $x:= x+1$ **else** $x:= y$

This part is unremarkable. It just shows that the Substitution Law applies to **if**s.

$=$ **if** $y>1$ **then** $x:= 2$ **else** $x:=y$

(k) $x:= 1.\ x'>x.\ x' = x+1$

We can use the Substitution Law on the first two pieces of this dependent composition to obtain

$=\quad x'>1.\ x' = x+1$

Now we have to use the axiom for dependent composition to get a further simplification.

$=\quad \exists x'', y''\cdot x''>1 \ \land\ x' = x''+1$

$=\quad x'>2$

The error we avoided in the first step is to replace x with 1 in the last part of the composition $x' = x+1$.

End of Specification Laws

4.0.2 Refinement

Two specifications P and Q are equal if and only if each is satisfied whenever the other is. Formally,

$\forall\sigma, \sigma'\cdot P=Q$

If a customer gives us a specification and asks us to implement it, we can instead implement an equal specification, and the customer will still be satisfied.

Suppose we are given specification P and we implement a stronger specification S . Since S implies P , all computer behavior satisfying S also satisfies P , so the customer will still be satisfied. We are allowed to change a specification, but only to an equal or stronger specification.

Specification P is refined by specification S if and only if P is satisfied whenever S is satisfied.

$\forall\sigma, \sigma'\cdot P\Leftarrow S$

Refinement of a specification P simply means finding another specification S that is everywhere equal or stronger. We call P the "problem" and S the "solution". In practice, to prove that P is refined by S , we work within the universal quantifications and prove $P \Leftarrow S$. In this context, we can pronounce $P \Leftarrow S$ as " P is refined by S ".

Here are some examples of refinement.

$x'>x \Leftarrow x'=x+1 \wedge y'=y$
$x'=x+1 \wedge y'=y \Leftarrow x:= x+1$
$x'\le x \Leftarrow$ **if** $x=0$ **then** $x'=x$ **else** $x'<x$
$x'>y'>x \Leftarrow y:= x+1.\ x:= y+1$

In each, the problem (left side) is refined by (follows from, is implied by) the solution (right side) for all initial and final values of all variables.

End of Refinement

4.0.3 Conditions

optional

A condition is a specification that refers to at most one state. A condition that refers to (at most) the initial state (prestate) is called an initial condition or precondition, and a condition that refers to (at most) the final state (poststate) is called a final condition or postcondition. In the following two definitions let P and S be specifications.

The exact precondition for P to be refined by S is $\forall\sigma'\cdot P\Leftarrow S$.
The exact postcondition for P to be refined by S is $\forall\sigma\cdot P\Leftarrow S$.

For example, although $x'>5$ is not refined by $x:= x+1$, we can calculate (in one integer variable)

(the exact precondition for $x'>5$ to be refined by $x:= x+1$)
$=\ \forall x'\cdot x'>5 \Leftarrow (x:= x+1)$
$=\ \forall x'\cdot x'>5 \Leftarrow x'=x+1$ One-Point Law
$=\ x+1 > 5$
$=\ x > 4$

This means that a computation satisfying $x:= x+1$ will also satisfy $x'>5$ if and only if it starts with $x>4$. If we are interested only in prestates such that $x>4$, then we should weaken our problem with that antecedent, obtaining the refinement

$x>4 \Rightarrow x'>5 \Leftarrow x:= x+1$

There is a similar story for postconditions. For example, although $x>4$ is unimplementable,

(the exact postcondition for $x>4$ to be refined by $x:= x+1$)
$=\ \forall x\cdot x>4 \Leftarrow (x:= x+1)$
$=\ \forall x\cdot x>4 \Leftarrow x'=x+1$ One-Point Law
$=\ x'-1 > 4$
$=\ x' > 5$

This means that a computation satisfying $x:= x+1$ will also satisfy $x>4$ if and only if it ends with $x'>5$. If we are interested only in poststates such that $x'>5$, then we should weaken our problem with that antecedent, obtaining the refinement

$x'>5 \Rightarrow x>4 \Leftarrow x:= x+1$

For easier understanding, it may help to use the Contrapositive Law to rewrite the specification $x'>5 \Rightarrow x>4$ as the equivalent specification $x\le 4 \Rightarrow x'\le 5$.

We can now find the exact pre- and postcondition for P to be refined by S . Any precondition that implies the exact precondition is called a sufficient precondition. Any precondition implied by the exact precondition is called a necessary precondition. Any postcondition that implies the exact postcondition is called a sufficient postcondition. Any postcondition implied by the exact postcondition is called a necessary postcondition. The exact precondition is therefore the necessary and sufficient precondition, and similarly for postconditions.

Exercise 112(c) asks for the exact precondition and postcondition for $x{:=}\ x^2$ to move integer variable x farther from zero. To answer, we must first state formally what it means to move x farther from zero: $abs\ x' > abs\ x$ (where abs is the absolute value function; its definition can be found in Chapter 11). We now calculate

(the exact precondition for $abs\ x' > abs\ x$ to be refined by $x{:=}\ x^2$)

$= \forall x'\cdot\ abs\ x' > abs\ x \ \Leftarrow\ x' = x^2$ — One-Point Law

$= abs\ (x^2) > abs\ x$ — by the arithmetic properties of $abs\ x$ and x^2

$= x \ne -1\ \land\ x \ne 0\ \land\ x \ne 1$

(the exact postcondition for $abs\ x' > abs\ x$ to be refined by $x{:=}\ x^2$)

$= \forall x\cdot\ abs\ x' > abs\ x \ \Leftarrow\ x' = x^2$ — after several steps including domain splitting and variable change and using the arithmetic properties of $abs\ x$ and x^2

$= x' \ne 0 \land x' \ne 1$

If x starts anywhere but -1 , 0 , or 1 , we can be sure it will move farther from zero; if x ends anywhere but 0 or 1 , we can be sure it did move farther from zero.

Let P and Q be any specifications, and let C be a precondition, and let C' be the corresponding postcondition (in other words, C' is the same as C but with primes on all the state variables). Then the following are laws.

$C \land (P.\ Q) \ \Leftarrow\ C \land P.\ Q$

$C \Rightarrow (P.Q) \ \Leftarrow\ C \Rightarrow P.\ Q$

$(P.Q) \land C' \ \Leftarrow\ P.\ Q \land C'$

$(P.Q) \Leftarrow C' \ \Leftarrow\ P.\ Q \Leftarrow C'$

$P.\ C \land Q \ \Leftarrow\ P \land C'.\ Q$

$P.\ Q \ \Leftarrow\ P \land C'.\ C \Rightarrow Q$

Precondition Law:

C is a sufficient precondition for P to be refined by S
if and only if $C \Rightarrow P$ is refined by S .

Postcondition Law:

C' is a sufficient postcondition for P to be refined by S
if and only if $C' \Rightarrow P$ is refined by S .

End of Conditions

4.0.4 Programs

A program is a description or specification of computer behavior. A computer executes a program by behaving according to the program, by satisfying the program. People often confuse programs with computer behavior. They talk about what a program "does"; of course it just sits there on the page or screen; it is the computer executing the program that does something. They ask whether a program "terminates"; of course it does; it is the specified behavior that may not terminate. A program is not behavior, but a specification of behavior. Furthermore, a computer may not behave as specified by a program for a variety of reasons: a computer component may break, a compiler may have a bug, or a resource may become exhausted (stack overflow, number overflow), to mention a few. Then the difference between a program and the computer behavior is obvious.

A program is a specification of computer behavior; for now, that means it is a boolean expression relating prestate and poststate. Not every specification is a program. A program is an implemented specification, that is, a specification for which an implementation has been provided, so that a computer can execute it. In this chapter we need only a very few programming notations that are similar to those found in many popular programming languages. We take the following:

(a) ok is a program.
(b) If x is any state variable and e is an implemented expression of the initial values, then $x{:=}\ e$ is a program.
(c) If b is an implemented boolean expression of the initial values, and P and Q are programs, then **if** b **then** P **else** Q is a program.
(d) If P and Q are programs then $P.Q$ is a program.
(e) An implementable specification that is refined by a program is a program.

For the "implemented expressions" referred to in (b) and (c), we take the expressions of Chapters 1 and 2: booleans, numbers, characters, bunches, sets, strings, and lists, with all their operators. We omit functions and quantifiers because they are harder to implement, but they are still welcome in specifications.

Part (e) states that any implementable specification P is a program if a program S is provided such that $P \Leftarrow S$ is a theorem. To execute P, just execute S. The refinement acts as a procedure (void function, method) declaration; P acts as the procedure name, and S as the procedure body; use of the name P acts as a call. Recursion is allowed; calls to P may occur within S.

Here is an example refinement in one integer variable x.

$$x{\ge}0 \Rightarrow x'{=}0 \quad \Leftarrow \quad \textbf{if } x{=}0 \textbf{ then } ok \textbf{ else } (x{:=}\ x{-}1.\ \ x{\ge}0 \Rightarrow x'{=}0)$$

The problem is $x{\ge}0 \Rightarrow x'{=}0$. The solution is **if** $x{=}0$ **then** ok **else** $(x{:=}\ x{-}1.\ \ x{\ge}0 \Rightarrow x'{=}0)$. In the solution, the problem reappears. According to (e), the problem is a program if its solution is a program. And the solution is a program if $x{\ge}0 \Rightarrow x'{=}0$ is a program. By saying "recursion is allowed" we break the impasse and declare that $x{\ge}0 \Rightarrow x'{=}0$ is a program. A computer executes it by behaving according to the solution, and whenever the problem is encountered again, the behavior is again according to the solution.

We must prove the refinement, so we do that now.

	if $x{=}0$ **then** ok **else** $(x{:=}\ x{-}1.\ \ x{\ge}0 \Rightarrow x'{=}0)$	Replace ok; Substitution Law
=	**if** $x{=}0$ **then** $x'{=}x$ **else** $x{-}1{\ge}0 \Rightarrow x'{=}0$	use context $x{=}0$ to modify the **then**-part and use context $x{\ne}0$ and $x{:}\ int$ to modify the **else**-part
=	**if** $x{=}0$ **then** $x{\ge}0 \Rightarrow x'{=}0$ **else** $x{\ge}0 \Rightarrow x'{=}0$	Case Idempotence
=	$x{\ge}0 \Rightarrow x'{=}0$	

End of Programs

A specification serves as a contract between a client who wants a computer to behave a certain way and a programmer who will program a computer to behave as desired. For this purpose, a specification must be written as clearly, as understandably, as possible. The programmer then refines the specification to obtain a program, which a computer can execute. Sometimes the clearest, most understandable specification is already a program. When that is so, there is no need for any other specification, and no need for refinement. However, the programming notations are only part of the specification notations: those that happen to be implemented. Specifiers should use whatever notations help to make their specifications clear, including but not limited to programming notations.

End of Specifications

4.1 Program Development

4.1.0 Refinement Laws

Once we have a specification, we refine it until we have a program. We have only five programming notations to choose from when we refine. Two of them, *ok* and assignment, are programs and require no further refinement. The other three solve the given refinement problem by raising new problems to be solved by further refinement. When these new problems are solved, their solutions will contribute to the solution of the original problem, according to the first of our refinement laws.

Refinement by Steps (Stepwise Refinement) (monotonicity, transitivity)
If $A \Leftarrow$ **if** b **then** C **else** D and $C \Leftarrow E$ and $D \Leftarrow F$ are theorems,
then $A \Leftarrow$ **if** b **then** E **else** F is a theorem.
If $A \Leftarrow B.C$ and $B \Leftarrow D$ and $C \Leftarrow E$ are theorems, then $A \Leftarrow D.E$ is a theorem.
If $A \Leftarrow B$ and $B \Leftarrow C$ are theorems, then $A \Leftarrow C$ is a theorem.

Refinement by Steps allows us to introduce one programming construct at a time into our ultimate solution. The next law allows us to break the problem into parts in a different way.

Refinement by Parts (monotonicity, conflation)
If $A \Leftarrow$ **if** b **then** C **else** D and $E \Leftarrow$ **if** b **then** F **else** G are theorems,
then $A \wedge E \Leftarrow$ **if** b **then** $C \wedge F$ **else** $D \wedge G$ is a theorem.
If $A \Leftarrow B.C$ and $D \Leftarrow E.F$ are theorems, then $A \wedge D \Leftarrow B \wedge E.\ C \wedge F$ is a theorem.
If $A \Leftarrow B$ and $C \Leftarrow D$ are theorems, then $A \wedge C \Leftarrow B \wedge D$ is a theorem.

When we add to our repertoire of programming operators in later chapters, the new operators must obey similar Refinement by Steps and Refinement by Parts laws. Our final refinement law is

Refinement by Cases
$P \Leftarrow$ **if** b **then** Q **else** R is a theorem if and only if
$P \Leftarrow b \wedge Q$ and $P \Leftarrow \neg b \wedge R$ are theorems.

As an example of Refinement by Cases, we can prove
$x' \le x \Leftarrow$ **if** $x=0$ **then** $x'=x$ **else** $x'<x$
by proving both
$x' \le x \Leftarrow x=0 \wedge x'=x$
and
$x' \le x \Leftarrow x \ne 0 \wedge x'<x$

End of Refinement Laws

4.1.1 List Summation

As an example of program development, let us do Exercise 142: write a program to find the sum of a list of numbers. Let L be the list of numbers, and let s be a number variable whose final value will be the sum of the items in L. Now s is a state variable, so it corresponds to two mathematical variables s and s'. Our solution does not change list L, so L is a state constant (which is a mathematical variable).

The first step is to express the problem as clearly and as simply as possible. One possibility is

$$s:= \Sigma L$$

We are assuming the expression to the right of the assignment symbol is not implemented, so this specification is not a program until we refine it. This specification says not only that s has the right final value, but also that all other variables are unchanged, and that makes it a little difficult to implement. So let's choose a weaker specification that is easier to implement.

$$s' = \Sigma L$$

The algorithmic idea is obvious: consider each item of the list in order, accumulating the sum. To do so we need an accumulator variable, and we may as well use s for that. We also need a variable to serve as index in the list, saying how many items have been considered; let us take natural variable n for that. We must begin by assigning 0 to both s and n to indicate that we have summed zero items so far. We complete the task by adding the remaining items (which means all of them) to the sum.

$$s' = \Sigma L \quad \Leftarrow \quad s:= 0.\ \ n:= 0.\ \ s' = s + \Sigma\, L\, [n;..\#L]$$

(Remember: list indexes start at 0, and the list $[n;..\#L]$ includes n and excludes $\#L$.) This theorem is easily proven by two applications of the Substitution Law. We consider that we have solved the original problem, but now we have a new problem to solve: $s' = s + \Sigma\, L\, [n;..\#L]$. When we refine this new problem, we must ignore the context in which it arose; in particular, we ignore that $s{=}0 \wedge n{=}0$. The new specification represents the problem when n items have been summed and the rest remain to be summed, for arbitrary n. One of the possible values for n is $\#L$, which means that all items have been summed. That suggests that we use Case Creation next.

$$s' = s + \Sigma\, L\, [n;..\#L] \quad \Leftarrow \quad \textbf{if } n{=}\#L \textbf{ then } n{=}\#L \Rightarrow s' = s + \Sigma\, L\, [n;..\#L]$$
$$\textbf{else } n{\neq}\#L \Rightarrow s' = s + \Sigma\, L\, [n;..\#L]$$

Now we have two new problems, but one is trivial.

$$n{=}\#L \Rightarrow s' = s + \Sigma\, L\, [n;..\#L] \quad \Leftarrow \quad ok$$

In the other problem, not all items have been summed $(n{\neq}\#L)$. That means there is at least one more item to be added to the sum, so let us add one more item to the sum. To complete the refinement, we must also add any remaining items.

$$n{\neq}\#L \Rightarrow s' = s + \Sigma\, L\, [n;..\#L] \quad \Leftarrow \quad s:= s{+}Ln.\ \ n:= n{+}1.\ \ s' = s + \Sigma\, L\, [n;..\#L]$$

This refinement is proven by two applications of the Substitution Law. The final specification has already been refined, so we have finished programming.

One point that deserves further attention is our use of $n{\neq}\#L$ to mean that not all items have been summed. We really need $n{<}\#L$ to say that there is at least one more item. The specification in which this appears

$$n{\neq}\#L \Rightarrow s' = s + \Sigma\, L\, [n;..\#L]$$

also uses the notation $n;..\#L$, which is defined only for $n{\le}\#L$. We may therefore consider that $n{\le}\#L$ is implicit in our use of the notation; this, together with $n{\neq}\#L$, tells us $n{<}\#L$ as required.

In our first refinement, we could have used a weaker specification to say that n items have been summed and the rest remain to be added. We could have said

$$s' = \Sigma L \quad \Leftarrow \quad s:= 0.\ \ n:= 0.\ \ 0{\le}n{\le}\#L \ \wedge\ s = \Sigma\, L\, [0;..n] \Rightarrow s' = s + \Sigma\, L\, [n;..\#L]$$

For those who were uncomfortable about the use of implicit information in the preceding paragraph, the first part of the antecedent $(0{\le}n{\le}\#L)$ makes the needed bound on n explicit. The second part of the antecedent $(s = \Sigma\, L\, [0;..n])$ is not used anywhere.

When a compiler translates a program into machine language, it treats each refined specification as just an identifier. For example, the summation program looks like

$A \Leftarrow s:= 0.\ n:= 0.\ B$
$B \Leftarrow$ **if** $n=\#L$ **then** C **else** D
$C \Leftarrow ok$
$D \Leftarrow s:= s+Ln.\ n:= n+1.\ B$

to a compiler. Using the Law of Refinement by Steps, a compiler can compile the calls to C and D in-line (macro-expansion) creating

$B \Leftarrow$ **if** $n=\#L$ **then** ok **else** $(s:= s+Ln.\ n:= n+1.\ B)$

So, for the sake of efficient execution, there is no need for us to put the pieces together, and we needn't worry about the number of refinements we use.

If we want to execute this program on a computer, we must translate it to a programming language that is implemented on that computer. For example, we can translate the summation program to C as follows.

```
void B (void) {if (n == sizeof(L)/sizeof(L[0])) ; else { s = s + L[n];  n = n+1;  B( ); }}
s = 0;  n = 0;  B( );
```

A call that is executed last in the solution of a refinement, as B is here, can be translated as just a branch (jump) machine instruction. Many compilers do a poor job of translating calls, so we might prefer to write "go to", which will then be translated as a branch instruction.

```
s = 0;  n = 0;
B: if (n == sizeof(L)/sizeof(L[0])) ; else { s = s + L[n];  n = n+1;  goto B; }
```

Most calls can be translated either as nothing (in-line), or as a branch, so we needn't worry about calls, even recursive calls, being inefficient.

End of List Summation

4.1.2 Binary Exponentiation

Now let's try Exercise 149: given natural variables x and y, write a program for $y' = 2^x$ without using exponentiation. Here is a solution that is neither the simplest nor the most efficient. It has been chosen to illustrate several points.

$y'=2^x \Leftarrow$ **if** $x=0$ **then** $x=0 \Rightarrow y'=2^x$ **else** $x>0 \Rightarrow y'=2^x$
$x=0 \Rightarrow y'=2^x \Leftarrow y:= 1.\ x:= 3$
$x>0 \Rightarrow y'=2^x \Leftarrow x>0 \Rightarrow y'=2^{x-1}.\ y'=2\times y$
$x>0 \Rightarrow y'=2^{x-1} \Leftarrow x'=x-1.\ y'=2^x$
$y'=2\times y \Leftarrow y:= 2\times y.\ x:= 5$
$x'=x-1 \Leftarrow x:= x-1.\ y:= 7$

The first refinement divides the problem into two cases; in the second case $x \ne 0$, and since x is natural, $x>0$. In the second refinement, since $x=0$, we want $y'=1$, which we get by the assignment $y:= 1$. The other assignment $x:= 3$ is superfluous, and our solution would be simpler without it; we have included it just to make the point that it is allowed by the specification. The next refinement makes $y'=2^x$ in two steps: first $y'=2^{x-1}$ and then double y. The antecedent $x>0$ ensures that 2^{x-1} will be natural. The last two refinements again contain superfluous assignments. Without the theory of programming, we would be very worried that these superfluous assignments might in some way make the result wrong. With the theory, we only need to prove these six refinements, and we are confident that execution will not give us a wrong answer.

This solution has been constructed to make it difficult to follow the execution. You can make the program look more familiar by replacing the nonprogramming notations with single letters.

$A \Leftarrow$ **if** $x=0$ **then** B **else** C
$B \Leftarrow y:= 1.\ x:= 3$
$C \Leftarrow D.\ E$
$D \Leftarrow F.\ A$
$E \Leftarrow y:= 2\times y.\ x:= 5$
$F \Leftarrow x:= x-1.\ y:= 7$

You can reduce the number of refinements by applying the Stepwise Refinement Law.

$A \Leftarrow$ **if** $x=0$ **then** $(y:= 1.\ x:= 3)$ **else** $(x:= x-1.\ y:= 7.\ A.\ y:= 2\times y.\ x:= 5)$

You can translate this into a programming language that is available on a computer near you. For example, in C it becomes

```
int x, y;
void A (void) {if (x==0) {y = 1;  x = 3;} else {x = x–1;  y = 7;  A ( );  y = 2*y;  x = 5;}}
```

You can then test it on a variety of x values. For example, execution of

```
x = 5;  A ( );  printf ("%i", y);
```

will print 32 . But you will find it easier to prove the refinements than to try to understand all possible executions of this program without any theory.

End of Binary Exponentiation

End of Program Development

4.2 Time

So far, we have talked only about the result of a computation, not about how long it takes. To talk about time, we just add a time variable. We do not change the theory at all; the time variable is treated just like any other variable, as part of the state. The state $\sigma = t; x; y; ...$ now consists of a time variable t and some memory variables x , y , ... The interpretation of t as time is justified by the way we use it. In an implementation, the time variable does not require space in the computer's memory; it simply represents the time at which execution occurs.

We use t for the initial time, the time at which execution starts, and t' for the final time, the time at which execution ends. To allow for nontermination we take the domain of time to be a number system extended with ∞ . The number system we extend can be the naturals, or the integers, or the rationals, or the reals, whichever we prefer.

Time cannot decrease, therefore a specification S with time is implementable if and only if

$\forall\sigma\cdot \exists\sigma'\cdot S \land t'\geq t$

For each initial state, there must be at least one satisfactory final state in which time has not decreased.

There are many ways to measure time. We present just two: real time and recursive time.

4.2.0 Real Time

In the real time measure, the time variable t is of type *xreal* . Real time has the advantage of measuring the actual execution time; for some applications, such as the control of a chemical or nuclear reaction, this is essential. It has the disadvantage of requiring intimate knowledge of the implementation (hardware and software).

To obtain the real execution time of a program, modify the program as follows.

- Replace each assignment $x:= e$ by
  $t:= t+$ (the time to evaluate and store e). $x:= e$
- Replace each conditional **if** b **then** P **else** Q by
  $t:= t+$ (the time to evaluate b and branch). **if** b **then** P **else** Q

- Replace each call P by
  $t:= t+$ (the time for the call and return). P
  For a call that is implemented "in-line", this time will be zero. For a call that is executed last in a refinement solution, it may be just the time for a branch. Sometimes it will be the time required to push a return address onto a stack and branch, plus the time to pop the return address and branch back.

- Each refined specification can include time. For example, let f be a function of the initial state σ . Then
  $t' = t + f\sigma$
  specifies that $f\sigma$ is the execution time,
  $t' \le t + f\sigma$
  specifies that $f\sigma$ is an upper bound on the execution time, and
  $t' \ge t + f\sigma$
  specifies that $f\sigma$ is a lower bound on the execution time.

We could place the time increase after each of the programming notations instead of before. By placing it before, we make it easier to use the Substitution Law.

In Subsection 4.0.4 we considered an example of the form

$P \Leftarrow$ **if** $x=0$ **then** ok **else** $(x:= x-1.\ P)$

Suppose that the **if** , the assignment, and the call each take time 1 . The refinement becomes

$P \Leftarrow$ $t:= t+1.$ **if** $x=0$ **then** ok **else** $(t:= t+1.\ x:= x-1.\ t:= t+1.\ P)$

This refinement is a theorem when

$P =$ **if** $x\ge0$ **then** $x'=0 \ \wedge\ t' = t+3\times x+1$ **else** $t'=\infty$

When x starts with a nonnegative value, execution of this program sets x to 0, and takes time $3\times x+1$ to do so; when x starts with a negative value, execution takes infinite time, and nothing is said about the final value of x . This is a reasonable description of the computation.

The same refinement

$P \Leftarrow$ $t:= t+1.$ **if** $x=0$ **then** ok **else** $(t:= t+1.\ x:= x-1.\ t:= t+1.\ P)$

is also a theorem for various other definitions of P , including the following three:

$P = x'=0$

$P =$ **if** $x\ge0$ **then** $t'=t+3\times x+1$ **else** $t'=\infty$

$P = x'=0 \ \wedge$ **if** $x\ge0$ **then** $t'=t+3\times x+1$ **else** $t'=\infty$

The first one ignores time, and the second one ignores the result. If we prove the refinement for the first one, and for the second one, then the Law of Refinement by Parts says that we have proven it for the last one also. The last one says that execution of this program always sets x to 0 ; when x starts with a nonnegative value, it takes time $3\times x+1$ to do so; when x starts with a negative value, it takes infinite time. It is strange to say that a result such as $x'=0$ is obtained at time infinity. To say that a result is obtained at time infinity is really just a way of saying that the result is never obtained. The only reason for saying it this strange way is so that we can divide the proof into two parts, the result and the timing, and then we get their conjunction for free. So we just ignore anything that a specification says about the values of variables at time infinity.

Even stranger things can be said about the values of variables at time infinity. Consider

$$Q \Leftarrow t:= t+1.\ Q$$

Three implementable specifications for which this is a theorem are

$$Q = t'=\infty$$
$$Q = x'=2 \land t'=\infty$$
$$Q = x'=3 \land t'=\infty$$

The first looks reasonable, but according to the last two we can show that the "final" value of x is 2, and also 3. But since $t'=\infty$, we are really saying in both cases that we never obtain a result.

End of Real Time

4.2.1 Recursive Time

The recursive time measure is more abstract than the real time measure; it does not measure the actual execution time. Its advantage is that we do not have to know any implementation details. In the recursive time measure, the time variable t has type *xint*, and

- each recursive call costs time 1;
- all else is free.

This measure neglects the time for "straight-line" and "branching" programs, charging only for loops.

In the recursive measure, our earlier example becomes

$$P \Leftarrow \textbf{if } x=0 \textbf{ then } ok \textbf{ else } (x:= x-1.\ t:= t+1.\ P)$$

which is a theorem for various definitions of P, including the following two:

$$P = \textbf{if } x\ge 0 \textbf{ then } x'=0 \land t' = t+x \textbf{ else } t'=\infty$$
$$P = x'=0 \land \textbf{if } x\ge 0 \textbf{ then } t' = t+x \textbf{ else } t'=\infty$$

The execution time, which was $3\times x + 1$ for nonnegative x in the real time measure, has become just x in the recursive time measure. The recursive time measure tells us less than the real time measure; it says only that the execution time increases linearly with x, but not what the multiplicative and additive constants are.

That example was a direct recursion: problem P was refined by a solution containing a call to P. Recursions can also be indirect. For example, problem A may be refined by a solution containing a call to B, whose solution contains a call to C, whose solution contains a call to A. In an indirect recursion, which calls are recursive? All of them? Or just one of them? Which one? The answer is that for recursive time it doesn't matter very much; the constants may be affected, but the form of the time expression is unchanged. The general rule of recursive time is that

- in every loop of calls, there must be a time increment of at least one time unit.

End of Recursive Time

Let us prove a refinement with time (Exercise 119(b)):

$$R \Leftarrow \textbf{if } x=1 \textbf{ then } ok \textbf{ else } (x:= div\ x\ 2.\ t:= t+1.\ R)$$

where x is an integer variable, and

$$R = x'=1 \land \textbf{if } x\ge 1 \textbf{ then } t' \le t + log\ x \textbf{ else } t'=\infty$$

In order to use Refinement by Parts even more effectively, we rewrite the **if then else** as a conjunction.

$$R = x'=1 \land (x\ge 1 \Rightarrow t' \le t + log\ x) \land (x<1 \Rightarrow t'=\infty)$$

This exercise uses the functions *div* (divide and round down) and *log* (binary logarithm). Execution of this program always sets x to 1; when x starts with a positive value, it takes logarithmic time; when x starts nonpositive, it takes infinite time. Thanks to Refinement by Parts, it is sufficient to verify the three conjuncts of R separately:

$$x'=1 \;\Leftarrow\; \textbf{if } x=1 \textbf{ then } ok \textbf{ else } (x:= div\ x\ 2.\ \ t:= t+1.\ \ x'=1)$$
$$x\ge 1 \Rightarrow t' \le t + log\ x \;\Leftarrow\; \textbf{if } x=1 \textbf{ then } ok$$
$$\textbf{else } (x:= div\ x\ 2.\ \ t:= t+1.\ \ x\ge 1 \Rightarrow t' \le t + log\ x)$$
$$x<1 \Rightarrow t'=\infty \;\Leftarrow\; \textbf{if } x=1 \textbf{ then } ok \textbf{ else } (x:= div\ x\ 2.\ \ t:= t+1.\ \ x<1 \Rightarrow t'=\infty)$$

We can apply the Substitution Law to rewrite these three parts as follows:

$$x'=1 \;\Leftarrow\; \textbf{if } x=1 \textbf{ then } x'=x \wedge t'=t \textbf{ else } x'=1$$
$$x\ge 1 \Rightarrow t' \le t + log\ x \;\Leftarrow\; \textbf{if } x=1 \textbf{ then } x'=x \wedge t'=t$$
$$\textbf{else } div\ x\ 2 \ge 1 \Rightarrow t' \le t + 1 + log\ (div\ x\ 2)$$
$$x<1 \Rightarrow t'=\infty \;\Leftarrow\; \textbf{if } x=1 \textbf{ then } x'=x \wedge t'=t \textbf{ else } div\ x\ 2 < 1 \Rightarrow t'=\infty$$

Now we break each of these three parts in two using Refinement by Cases. We must prove

$$x'=1 \;\Leftarrow\; x=1 \wedge x'=x \wedge t'=t$$
$$x'=1 \;\Leftarrow\; x\ne 1 \wedge x'=1$$

$$x\ge 1 \Rightarrow t' \le t + log\ x \;\Leftarrow\; x=1 \wedge x'=x \wedge t'=t$$
$$x\ge 1 \Rightarrow t' \le t + log\ x \;\Leftarrow\; x\ne 1 \wedge (div\ x\ 2 \ge 1 \Rightarrow t' \le t + 1 + log\ (div\ x\ 2))$$

$$x<1 \Rightarrow t'=\infty \;\Leftarrow\; x=1 \wedge x'=x \wedge t'=t$$
$$x<1 \Rightarrow t'=\infty \;\Leftarrow\; x\ne 1 \wedge (div\ x\ 2 < 1 \Rightarrow t'=\infty)$$

We'll prove each of these six implications in turn. First,

$(x'=1 \;\Leftarrow\; x=1 \wedge x'=x \wedge t'=t)$ by transitivity and specialization
$=\ \top$

Next,

$(x'=1 \;\Leftarrow\; x\ne 1 \wedge x'=1)$ by specialization
$=\ \top$

Next,

$(x\ge 1 \Rightarrow t' \le t + log\ x \;\Leftarrow\; x=1 \wedge x'=x \wedge t'=t)$ use the first Law of Portation to move the initial antecedent over to the solution side where it becomes a conjunct
$=\ t' \le t + log\ x \Leftarrow x=1 \wedge x'=x \wedge t'=t$ and note that $log\ 1 = 0$
$=\ \top$

Next comes the hardest one of the six.

$(x\ge 1 \Rightarrow t' \le t + log\ x \;\Leftarrow\; x\ne 1 \wedge (div\ x\ 2 \ge 1 \Rightarrow t' \le t + 1 + log\ (div\ x\ 2)))$
Again use the first Law of Portation to move the initial antecedent over to the solution side where it becomes a conjunct.
$=\ t' \le t + log\ x \Leftarrow x>1 \wedge (div\ x\ 2 \ge 1 \Rightarrow t' \le t + 1 + log\ (div\ x\ 2))$
Since x is an integer, $x>1 = div\ x\ 2 \ge 1$, so by the first Law of Discharge,
$=\ t' \le t + log\ x \Leftarrow x>1 \wedge t' \le t + 1 + log\ (div\ x\ 2)$
By the first Law of Portation, move $t' \le t + 1 + log\ (div\ x\ 2)$ over to the left side.
$=\ (t' \le t + 1 + log\ (div\ x\ 2) \Rightarrow t' \le t + log\ x) \Leftarrow x>1$
By a Connection Law, $(t'\le a \Rightarrow t'\le b) \Leftarrow a\le b$.
$\Leftarrow\ t + 1 + log\ (div\ x\ 2) \le t + log\ x \Leftarrow x>1$ subtract 1 from each side
$=\ t + log\ (div\ x\ 2) \le t + log\ x - 1 \Leftarrow x>1$ law of logarithms
$=\ t + log\ (div\ x\ 2) \le t + log\ (x/2) \Leftarrow x>1$ log and $+$ are monotonic for $x>0$
$\Leftarrow\ div\ x\ 2 \le x/2$ div is $/$ and then round down
$=\ \top$

The next one is easier.

$$(x<1 \Rightarrow t'=\infty \Longleftarrow x=1 \wedge x'=x \wedge t'=t) \qquad \text{Law of Portation}$$
$$= \quad t'=\infty \Leftarrow x<1 \wedge x=1 \wedge x'=x \wedge t'=t \qquad \text{Put } x<1 \wedge x=1 \text{ together, and first Base Law}$$
$$= \quad t'=\infty \Leftarrow \bot \qquad \text{last Base Law}$$
$$= \quad \top$$

And finally,

$$(x<1 \Rightarrow t'=\infty \Longleftarrow x\ne1 \wedge (div\ x\ 2 < 1 \Rightarrow t'=\infty)) \qquad \text{Law of Portation}$$
$$= \quad t'=\infty \Leftarrow x<1 \wedge (div\ x\ 2 < 1 \Rightarrow t'=\infty) \qquad \text{Discharge}$$
$$= \quad t'=\infty \Leftarrow x<1 \wedge t'=\infty \qquad \text{Specialization}$$
$$= \quad \top$$

And that completes the proof.

4.2.2 Termination

A specification is a contract between a customer who wants some software and a programmer who provides it. The customer can complain that the programmer has broken the contract if, when executing the program, the customer observes behavior contrary to the specification.

Here are four specifications, each of which says that variable x has final value 2 .

(a) $x'=2$

(b) $x'=2 \wedge t'<\infty$

(c) $x'=2 \wedge (t<\infty \Rightarrow t'<\infty)$

(d) $x'=2 \wedge t'\le t+1$

Specification (a) says nothing about when the final value is wanted. It can be refined, including recursive time, as follows:

$$x'=2 \Longleftarrow t:=t+1.\ x'=2$$

This infinite loop provides a final value for x at time ∞ ; or, to say the same thing in different words, it never provides a final value for x . It may be an unkind refinement, but the customer has no ground for complaint. The customer is entitled to complain when the computation delivers a final state in which $x'\ne2$, and it never will.

In order to rule out this unkind implementation, the customer might ask for specification (b), which insists that the final state be delivered at a finite time. The programmer has to reject (b) because it is unimplementable: (b) $\wedge\ t'\ge t$ is unsatisfiable for $t=\infty$. It may seem strange to reject a specification just because it cannot be satisfied with nondecreasing time when the computation starts at time ∞ . After all, the customer doesn't want to start at time ∞ . But suppose the customer uses the software in a dependent (sequential) composition following an infinite loop. Then the computation does start at time ∞ (in other words, it never starts), and we cannot expect it to stop before it starts. An implementable specification must be satisfiable with nondecreasing time for all initial states, even for initial time ∞ .

So the customer tries again with specification (c). This says that if the computation starts at a finite time, it must end at a finite time. This one is implementable, but surprisingly, it can be refined with exactly the same construction as (a)! Including recursive time,

$$x'=2 \wedge (t<\infty \Rightarrow t'<\infty) \Longleftarrow t:=t+1.\ x'=2 \wedge (t<\infty \Rightarrow t'<\infty)$$

The customer may not be happy, but again there is no ground for complaint. The customer is entitled to complain if and only if the computation delivers a final state in which $x'\ne2$ or it takes forever. But there is never a time when the customer can complain that the computation has taken

forever, so the circumstances for complaint are exactly the same for (c) as for (a). This fact is accurately reflected in the theory, which allows the same refinement constructions for (c) as for (a).

Finally, the customer changes the specification to (d), measuring time in seconds. Now the customer can complain if either $x' \neq 2$ or the computation takes more than a second. An infinite loop is no longer possible because

$$x'=2 \land t' \le t+1 \quad \Leftarrow \quad t:= t+1.\ x'=2 \land t' \le t+1$$

is not a theorem. We refine

$$x'=2 \land t' \le t+1 \quad \Leftarrow \quad x:= 2$$

Specification (d) gives a time bound, therefore more circumstances in which to complain, therefore fewer refinements. Execution provides the customer with the desired result within the time bound.

One can complain about a computation if and only if one observes behavior contrary to the specification. For that reason, specifying termination without a practical time bound is worthless.

End of Termination

4.2.3 Soundness and Completeness — optional

The theory of programming presented in this book is sound in the following sense. Let P be an implementable specification. If we can prove the refinement

$P \Leftarrow$ (something possibly involving recursive calls to P)

then observations of the corresponding computation(s) will never (in finite time) contradict P.

The theory is incomplete in the following sense. Even if P is an implementable specification, and observations of the computation(s) corresponding to

$P \Leftarrow$ (something possibly involving recursive calls to P)

never (in finite time) contradict P, the refinement might not be provable. But in that case, there is another implementable specification Q such that the refinements

$P \Leftarrow Q$

$Q \Leftarrow$ (something possibly involving recursive calls to Q)

are both provable, where the Q refinement is identical to the earlier unprovable P refinement except for the change from P to Q. In that weaker sense, the theory is complete. There cannot be a theory of programming that is both sound and complete in the stronger sense.

End of Soundness and Completeness

4.2.4 Linear Search

Exercise 153: Write a program to find the first occurrence of a given item in a given list. The execution time must be linear in the length of the list.

Let the list be L and the value we are looking for be x (these are not state variables). Our program will assign natural variable h (for "here") the index of the first occurrence of x in L if x is there. If x is not there, its "first occurrence" is not defined; it will be convenient to indicate that x is not in L by assigning h the length of L. The specification is

$$\neg\ x\colon L\,(0,..h') \ \land\ (Lh'=x \lor h'=\#L) \ \land\ t' \le t+\#L$$

First, let us consider just the part of the specification that talks about h' and leave the time for later. The idea, of course, is to look at each item in the list, in order, starting at item 0, until we either find x or run out of items. To start at item 0 we refine as follows:

$$\neg\ x\colon L\,(0,..h') \ \land\ (Lh'=x \lor h'=\#L) \ \Leftarrow$$
$$h:= 0.\ h \le \#L \ \Rightarrow\ \neg\ x\colon L\,(h,..h') \ \land\ (Lh'=x \lor h'=\#L)$$

The new problem is like the original problem except that it describes a linear search starting at index h, for any h such that $0 \le h \le \#L$, not just at index 0. Since h is a natural variable, we did not bother to write $0 \le h$, but we could have written it. We needed to generalize the starting index to describe the remaining problem as the search progresses. We can satisfy $\neg x: L(h,..h')$ by doing nothing, which means $h'=h$ and the list segment is empty. To obtain $Lh'=x \lor h'=\#L$, we need to test either $Lh=x$ or $h=\#L$. To test $Lh=x$ we need to know $h<\#L$, so we have to test $h=\#L$ first.

$$h \le \#L \;\Rightarrow\; \neg x: L(h,..h') \;\land\; (Lh'=x \lor h'=\#L) \;\Leftarrow$$
$$\quad \textbf{if } h=\#L \textbf{ then } ok \textbf{ else } h<\#L \;\Rightarrow\; \neg x: L(h,..h') \;\land\; (Lh'=x \lor h'=\#L)$$

In the remaining problem we are able to test $Lh=x$.

$$h<\#L \;\Rightarrow\; \neg x: L(h,..h') \;\land\; (Lh'=x \lor h'=\#L) \;\Leftarrow$$
$$\quad \textbf{if } Lh=x \textbf{ then } ok \textbf{ else } (h:= h+1.\;\; h \le \#L \;\Rightarrow\; \neg x: L(h,..h') \;\land\; (Lh'=x \lor h'=\#L))$$

Now for the timing:

$$t' \le t+\#L \;\Leftarrow\; h:= 0.\;\; h \le \#L \Rightarrow t' \le t+\#L-h$$
$$h \le \#L \Rightarrow t' \le t+\#L-h \;\Leftarrow\; \textbf{if } h=\#L \textbf{ then } ok \textbf{ else } h<\#L \Rightarrow t' \le t+\#L-h$$
$$h<\#L \Rightarrow t' \le t+\#L-h \;\Leftarrow\; \textbf{if } Lh=x \textbf{ then } ok$$
$$\qquad \textbf{else } (h:= h+1.\;\; t:= t+1.\;\; h \le \#L \Rightarrow t' \le t+\#L-h)$$

Refinement by Parts says that if the same refinement structure can be used for two specifications, then it can be used for their conjunction. If we add $t:= t+1$ to the refinements that were not concerned with time, it won't affect their proof, and then we have the same refinement structure for both $\neg x: L(0,..h') \;\land\; (Lh'=x \lor h'=\#L)$ and $t' \le t+\#L$, so we know it works for their conjunction, and that solves the original problem. We could have divided $\neg x: L(0,..h') \;\land\; (Lh'=x \lor h'=\#L)$ into parts also. And of course we should prove our refinements.

It is not really necessary to take such small steps in programming. We could have written

$$\neg x: L(0,..h') \;\land\; (Lh'=x \lor h'=\#L) \;\land\; t' \le t+\#L \;\Leftarrow$$
$$\quad h:= 0.\;\; h \le \#L \;\Rightarrow\; \neg x: L(h,..h') \;\land\; (Lh'=x \lor h'=\#L) \;\land\; t' \le t+\#L-h$$
$$h \le \#L \;\Rightarrow\; \neg x: L(h,..h') \;\land\; (Lh'=x \lor h'=\#L) \;\land\; t' \le t+\#L-h \;\Leftarrow$$
$$\quad \textbf{if } h = \#L \textbf{ then } ok$$
$$\quad \textbf{else if } L\,h = x \textbf{ then } ok$$
$$\quad \textbf{else } (h:= h+1.\;\; t:= t+1.\;\; h \le \#L \;\Rightarrow\; \neg x: L(h,..h') \land (Lh'=x \lor h'=\#L) \land t' \le t+\#L-h)$$

But now, suppose we learn that the given list L is known to be nonempty. To take advantage of this new information, we rewrite the first refinement

$$\neg x: L(0,..h') \;\land\; (Lh'=x \lor h'=\#L) \;\land\; t' \le t+\#L \;\Leftarrow$$
$$\quad h:= 0.\;\; h<\#L \;\Rightarrow\; \neg x: L(h,..h') \;\land\; (Lh'=x \lor h'=\#L) \;\land\; t' \le t+\#L-h$$

and that's all; the new problem is already solved if we haven't made our steps too large. (Using the recursive time measure, there is no advantage to rewriting the first refinement this way. Using the real time measure, there is a small advantage.) As a habit, we write information about constants once, rather than in every specification. Here, for instance, we should say $\#L>0$ once so that we can use it when we prove our refinements, but we did not repeat it in each specification.

We can sometimes improve the execution time (real measure) by a technique called the sentinel. We need list L to be a variable so we can catenate one value to the end of it. If we can do so cheaply enough, we should begin by catenating x. Then the search is sure to find x, and we can skip the test $h=\#L$ each iteration. The program, ignoring time, becomes

$$\neg x: L(0,..h') \;\land\; (Lh'=x \lor h'=\#L) \;\Leftarrow\; L:= L^{+}[x].\;\; h:= 0.\;\; Q$$
$$Q \;\Leftarrow\; \textbf{if } Lh=x \textbf{ then } ok \textbf{ else } (h:= h+1.\;\; Q)$$

where $Q \;=\; L(\#L-1) = x \land h<\#L \;\Rightarrow\; L'=L \;\land\; \neg x: L(h,..h') \;\land\; Lh'=x$.

End of Linear Search

4.2.5 Binary Search

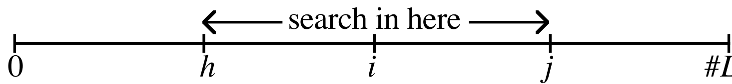
Exercise 154: Write a program to find a given item in a given nonempty sorted list. The execution time must be logarithmic in the length of the list. The strategy is to identify which half of the list contains the item if it occurs at all, then which quarter, then which eighth, and so on.

As in the previous subsection, let the list be L and the value we are looking for be x (these are not state variables). Our program will again assign natural variable h the index of an occurrence of x in L if x is there. But this time, let's indicate whether x is present in L by assigning boolean variable p the value $\top$ if it is and $\bot$ if not. Ignoring time for the moment, the problem is

$$x: L\,(0,..\#L) \;=\; p' \;\Longrightarrow\; Lh' = x$$

As the search progresses, we narrow the segment of the list that we need to search. Let us introduce natural variables i and j, and let specification R describe the search within the segment $h,..j$.

$$R \;=\; (x: L\,(h,..j) \;=\; p' \;\Longrightarrow\; Lh' = x)$$



We can now solve the problem.

$$(x: L\,(0,..\#L) \;=\; p' \;\Longrightarrow\; Lh' = x) \;\Longleftarrow\; h{:=}\,0.\;\; j{:=}\,\#L.\;\; h{<}j \Rightarrow R$$

$$h{<}j \Rightarrow R \;\Longleftarrow\; \textbf{if}\; j{-}h = 1\; \textbf{then}\; p{:=}\, Lh{=}x\; \textbf{else}\; j{-}h{\ge}2 \Rightarrow R$$

$$j{-}h{\ge}2 \Rightarrow R \;\Longleftarrow\; j{-}h{\ge}2 \;\Rightarrow\; h'{=}h{<}i'{<}j{=}j'.\;\; \textbf{if}\; Li{\le}x\; \textbf{then}\; h{:=}\, i\; \textbf{else}\; j{:=}\, i.\;\; h{<}j \Rightarrow R$$

To get the correct result, it does not matter how we choose i as long as it is properly between h and j. If we choose $i{:=}\, h{+}1$, we have a linear search. To obtain the best execution time in the worst case, we should choose i so it splits the segment $h;..j$ into halves. To obtain the best execution time on average, we should choose i so it splits the segment $h;..j$ into two segments in which there is an equal probability of finding x. In the absence of further information about probabilities, that again means splitting $h;..j$ into two segments of equal size.

$$j{-}h{\ge}2 \;\Rightarrow\; h'{=}h{<}i'{<}j{=}j' \;\Longleftarrow\; i{:=}\, div\,(h{+}j)\,2$$

After finding the mid-point i of the segment $h;..j$, it is tempting to test whether $Li{=}x$; if Li is the item we seek, we end execution right there, and this might improve the execution time. According to the recursive measure, the worst case time is not improved at all, and the average time is improved slightly by a factor of $(\#L)/(\#L{+}1)$ assuming equal probability of finding the item at each index and not finding it at all. And according to the real time measure, both the worst case and average execution times are a lot worse because the loop contains three tests instead of two.

For recursive execution time, put $t{:=}\, t{+}1$ before the final, recursive call. We will have to prove

$$T \;\Longleftarrow\; h{:=}\,0.\;\; j{:=}\,\#L.\;\; U$$

$$U \;\Longleftarrow\; \textbf{if}\; j{-}h = 1\; \textbf{then}\; p{:=}\, Lh{=}x\; \textbf{else}\; V$$

$$V \;\Longleftarrow\; i{:=}\, div\,(h{+}j)\,2.\;\; \textbf{if}\; Li{\le}x\; \textbf{then}\; h{:=}\, i\; \textbf{else}\; j{:=}\, i.\;\; t{:=}\, t{+}1.\;\; U$$

for a suitable choice of timing expressions T , U , V . If we do not see a suitable choice, we can always try executing the program a few times to see what we get. The worst case occurs when the item sought is larger than all items in the list. For this case we get

$\#L$	=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18 ...
$t'-t$	=	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5 ...

from which we define

$$T \;=\; t' \le t + ceil\,(log\,(\#L))$$
$$U \;=\; h<j \Rightarrow t' \le t + ceil\,(log\,(j-h))$$
$$V \;=\; j-h\ge 2 \Rightarrow t' \le t + ceil\,(log\,(j-h))$$

where $ceil$ is the function that rounds up.

We can identify three levels of care in programming. At the lowest level, one writes programs without bothering to write clear specifications and refinements. At the next level, one writes clear and precise specifications and refinements as we have just done for binary search; with practice, one can quickly see the correctness of the refinements without bothering to write formal proofs. At the highest level of care, one proves each refinement formally; to achieve this level, an automated theorem prover is very helpful.

Here are the proofs of the seven refinements in this subsection. For the first refinement

$$(x\colon L\,(0,..\#L) \;=\; p' \;\Longrightarrow\; Lh' = x) \;\Longleftarrow\; h:= 0.\; j:= \#L.\; h<j \Rightarrow R$$

we start with the right side.

$h:= 0.\; j:= \#L.\; h<j \Rightarrow R$ — replace R and then use Substitution Law twice

$=\;\; 0<\#L \Rightarrow (x\colon L\,(0,..\#L) \;=\; p' \;\Longrightarrow\; Lh' = x)$ — we are given that L is nonempty

$=\;\; (x\colon L\,(0,..\#L) \;=\; p' \;\Longrightarrow\; Lh' = x)$

The second refinement

$$h<j \Rightarrow R \;\Longleftarrow\; \textbf{if } j-h = 1 \textbf{ then } p:= Lh=x \textbf{ else } j-h\ge 2 \Rightarrow R$$

can be proven by cases. And its first case is

$(h<j \Rightarrow R \;\Longleftarrow\; j-h = 1 \;\wedge\; (p:= Lh=x))$ — portation

$=\;\; j-h = 1 \;\wedge\; (p:= Lh=x) \;\Rightarrow\; R$ — expand assignment and R

$=\;\; j-h = 1 \;\wedge\; p'=(Lh=x) \;\wedge\; h'=h \;\wedge\; i'=i \;\wedge\; j'=j \;\Rightarrow\; (x\colon L\,(h,..j) \;=\; p' \;\Longrightarrow\; Lh' = x)$

use the antecedent as context to simplify the consequent

$=\;\; j-h = 1 \;\wedge\; p'=(Lh=x) \;\wedge\; h'=h \;\wedge\; i'=i \;\wedge\; j'=j \;\Rightarrow\; (x=Lh \;=\; Lh=x \;\Longrightarrow\; Lh=x)$

Symmetry and Base and Reflexive Laws

$=\;\; \top$

The second case of the second refinement is

$(h<j \Rightarrow R \;\Longleftarrow\; j-h \neq 1 \;\wedge\; (j-h\ge 2 \;\Rightarrow\; R))$ — portation

$=\;\; j-h\ge 2 \wedge (j-h\ge 2 \;\Rightarrow\; R) \;\Rightarrow\; R$ — discharge

$=\;\; j-h\ge 2 \wedge R \;\Rightarrow\; R$ — specialization

$=\;\; \top$

The next refinement

$$j-h\ge 2 \Rightarrow R \;\Longleftarrow\; j-h\ge 2 \;\Rightarrow\; h'=h<i'<j=j'.$$
$$\textbf{if } Li\le x \textbf{ then } h:= i \textbf{ else } j:= i.$$
$$h<j \Rightarrow R$$

can be proven by cases. Using the distributive laws of dependent composition, its first case is

$$(j-h\ge 2 \Rightarrow R \ \Longleftarrow\ j-h\ge 2 \ \Rightarrow\ h'=h<i'<j=j'.\ Li\le x \land (h{:=}\ i.\ \ h<j \Rightarrow R))$$
Condition

$$\Longleftarrow \quad (j-h\ge 2 \Rightarrow R \ \Longleftarrow\ j-h\ge 2 \ \Rightarrow\ (h'=h<i'<j=j'.\ Li\le x \land (h{:=}\ i.\ \ h<j \Rightarrow R)))$$
Portation

$$= \quad j-h\ge 2 \land (j-h\ge 2 \ \Rightarrow\ (h'=h<i'<j=j'.\ Li\le x \land (h{:=}\ i.\ \ h<j \Rightarrow R))) \Rightarrow R$$
discharge $j-h\ge 2$ and specialize

$$\Longleftarrow \quad (h'=h<i'<j=j'.\ Li\le x \land (h{:=}\ i.\ \ h<j \Rightarrow R)) \Rightarrow R$$
expand first R and use Substitution

$$= \quad (h'=h<i'<j=j'.\ Li\le x \land (i<j \Rightarrow (x{:}\ L\,(i,..j)\ =\ p'\ \Longrightarrow\ Lh'=x))) \Rightarrow R$$
dependent composition

$$= \quad (\exists h'', i'', j'', p''\cdot\ \ h''=h<i''<j=j'' \land Li''\le x \land (i''<j'' \Rightarrow (x{:}\ L\,(i'',..j'')\ =\ p'\ \Longrightarrow\ Lh'=x))) \Rightarrow R$$
eliminate p'' , h'' , and j'' by one-point, and rename i'' to i

$$= \quad (\exists i\cdot\ h<i<j \land Li\le x \land (i<j \Rightarrow (x{:}\ L\,(i,..j)\ =\ p'\ \Longrightarrow\ Lh'=x))) \Rightarrow R$$
use context $i<j$ to discharge

$$= \quad (\exists i\cdot\ h<i<j \land Li\le x \land (x{:}\ L\,(i,..j)\ =\ p'\ \Longrightarrow\ Lh'=x)) \Rightarrow R$$
If $h<i$ and $Li\le x$ and L is sorted, then $x{:}\ L\,(i,..j)\ =\ x{:}\ L\,(h,..j)$

$$= \quad (\exists i\cdot\ h<i<j \land Li\le x \land (x{:}\ L\,(h,..j)\ =\ p'\ \Longrightarrow\ Lh'=x)) \Rightarrow R$$
note that $x{:}\ L\,(h,..j)\ =\ p'\ \Longrightarrow\ Lh'=x$ is R
since it doesn't use i , bring it outside the scope of the quantifier

$$= \quad (\exists i\cdot\ h<i<j \land Li\le x) \land R \Rightarrow R$$
specialize

$$= \quad \top$$

Its second case

$$j-h\ge 2 \Rightarrow R \ \Longleftarrow\ j-h\ge 2 \ \Rightarrow\ h'=h<i'<j=j'.\ Li>x \land (j{:=}\ i.\ \ h<j \Rightarrow R)$$

is proven just like its first case.

The next refinement is

$$(j-h\ge 2 \ \Rightarrow\ h'=h<i'<j=j' \ \Longleftarrow\ i{:=}\ div\,(h+j)\ 2)$$
expand assignment

$$= \quad (j-h\ge 2 \ \Rightarrow\ h'=h<i'<j=j' \ \Longleftarrow\ i'=div\,(h+j)\ 2 \ \land\ p'=p \ \land\ h'=h \ \land\ j'=j)$$
use the equations in the antecedent as context to simplify the consequent

$$= \quad (j-h\ge 2 \ \Rightarrow\ h=h<div\,(h+j)\ 2<j=j \ \Longleftarrow\ i'=div\,(h+j)\ 2 \land p'=p \land h'=h \land j'=j)$$
simplify $h=h$ and $j=j$ and use the properties of div

$$= \quad (j-h\ge 2 \ \Rightarrow\ \top \ \Longleftarrow\ i'=div\,(h+j)\ 2 \ \land\ p'=p \ \land\ h'=h \ \land\ j'=j)$$
base law twice

$$= \quad \top$$

The next refinement is

$$(T \ \Longleftarrow\ h{:=}\,0.\ j{:=}\,\#L.\ U)$$
replace T and U

$$= \quad (t'\le t+ceil\,(log\,(\#L)) \ \Longleftarrow\ h{:=}\,0.\ j{:=}\,\#L.\ h<j \Rightarrow t'\le t+ceil\,(log\,(j-h)))$$
Substitution Law twice

$$= \quad (t'\le t+ceil\,(log\,(\#L)) \ \Longleftarrow\ 0<\#L \Rightarrow t'\le t+ceil\,(log\,(\#L-0)))$$

$$= \quad \top$$

The next refinement

$$U \ \Longleftarrow\ \textbf{if}\ j-h=1\ \textbf{then}\ p{:=}\ Lh=x\ \textbf{else}\ V$$

can be proven by cases. And its first case is

$$(U \ \Longleftarrow\ j-h=1 \ \land\ (p{:=}\ Lh=x))$$
expand U and the assignment

$$= \quad (h<j \Rightarrow t'\le t+ceil\,(log\,(j-h)) \ \Longleftarrow\ j-h=1 \land p'=(Lh=x) \land h'=h \land i'=i \land j'=j \land t'=t)$$
use main antecedent as context in main consequent

$$= \quad (h<j \Rightarrow t\le t+ceil\,(log\ 1) \ \Longleftarrow\ j-h=1 \land p'=(Lh=x) \land h'=h \land i'=i \land j'=j \land t'=t)$$
Use $log\ 1=0$

$$= \quad (h<j \Rightarrow \top \ \Longleftarrow\ j-h=1 \land p'=(Lh=x) \land h'=h \land i'=i \land j'=j \land t'=t)$$
base law twice

$$= \quad \top$$

Its second case is

$(U \Leftarrow j{-}h \neq 1 \ \land\ V)$ — expand U and V

$= \ (h<j \Rightarrow t' \le t + ceil\ (log\ (j{-}h)) \Leftarrow j{-}h{\neq}1 \land (j{-}h{\ge}2 \Rightarrow t' \le t + ceil\ (log\ (j{-}h))))$ — portation

$= \ h<j \land j{-}h{\neq}1 \land (j{-}h{\ge}2 \Rightarrow t' \le t + ceil\ (log\ (j{-}h))) \Rightarrow t' \le t + ceil\ (log\ (j{-}h))$ — simplify

$= \ j{-}h{\ge}2 \land (j{-}h{\ge}2 \Rightarrow t' \le t + ceil\ (log\ (j{-}h))) \Rightarrow t' \le t + ceil\ (log\ (j{-}h))$ — discharge

$= \ j{-}h{\ge}2 \ \land\ t' \le t + ceil\ (log\ (j{-}h)) \Rightarrow t' \le t + ceil\ (log\ (j{-}h))$ — specialization

$= \ \top$

Before we prove the next refinement, we prove two little theorems first.

if *even* $(h+j)$
then ($div\ (h+j)\ 2 < j$
$= \ (h+j)/2 < j$
$= \ j{-}h > 0$
$\Leftarrow \ j{-}h \ge 2$)
else ($div\ (h+j)\ 2 < j$
$= \ (h+j{-}1)/2 < j$
$= \ j{-}h > -1$
$\Leftarrow \ j{-}h \ge 2$)

if *even* $(h+j)$
then $1 + ceil\ (log\ (j - div\ (h+j)\ 2))$
$= \ ceil\ (1 + log\ (j - (h+j)/2))$
$= \ ceil\ (log\ (j{-}h))$
else $1 + ceil\ (log\ (j - div\ (h+j)\ 2))$
$= \ ceil\ (1 + log\ (j - (h+j{-}1)/2))$
$= \ ceil\ (log\ (j{-}h{+}1))$ — If $h+j$ is odd then $j{-}h$ is odd and can't be a power of 2
$= \ ceil\ (log\ (j{-}h))$

Finally, the last refinement

$V \Leftarrow i{:=}\ div\ (h+j)\ 2.$ **if** $Li{\le}x$ **then** $h{:=}\ i$ **else** $j{:=}\ i.\ \ t{:=}\ t{+}1.\ \ U$

can be proven in two cases. First case:

$(V \Leftarrow i{:=}\ div\ (h+j)\ 2.\ \ Li{\le}x \ \land\ (h{:=}\ i.\ \ t{:=}\ t{+}1.\ \ U))$ — drop $Li{\le}x$ and replace U

$\Leftarrow \ (V \Leftarrow i{:=}\ div\ (h+j)\ 2.\ \ h{:=}\ i.\ \ t{:=}\ t{+}1.\ \ h<j \Rightarrow t' \le t + ceil\ (log\ (j{-}h)))$ — then use Substitution Law three times

$= \ (V \Leftarrow div\ (h+j)\ 2 < j \ \Rightarrow\ t' \le t + 1 + ceil\ (log\ (j - div\ (h+j)\ 2)))$ — use the two little theorems

$\Leftarrow \ (V \Leftarrow j{-}h \ge 2 \ \Rightarrow\ t' \le t + ceil\ (log\ (j{-}h)))$ — definition of V, reflexive Law

$= \ \top$

And the second case

$V \Leftarrow i{:=}\ div\ (h+j)\ 2.\ \ Li{>}x \ \land\ (j{:=}\ i.\ \ t{:=}\ t{+}1.\ \ U)$

is proven just like the first.

End of Binary Search

4.2.6 Fast Exponentiation

Exercise 151: Given rational variables x and z and natural variable y, write a program for $z' = x^y$ that runs fast without using exponentiation.

This specification does not say how fast the execution should be; let's make it as fast as we can. The idea is to accumulate a product, using variable z as accumulator. Define

$P \;=\; z' = z \times x^y$

We can solve the problem as follows, though this solution does not give the fastest possible computation.

$z'=x^y \;\Leftarrow\; z:= 1.\ P$

$P \;\Leftarrow\;$ **if** $y=0$ **then** ok **else** $y>0 \Rightarrow P$

$y>0 \Rightarrow P \;\Leftarrow\; z:= z \times x.\ y:= y-1.\ P$

To speed up the computation, we change our refinement of $y>0 \Rightarrow P$ to test whether y is even or odd; in the odd case we make no improvement but in the even case we can cut y in half.

$y>0 \Rightarrow P \;\Leftarrow\;$ **if** $even\ y$ **then** $even\ y \wedge y>0 \Rightarrow P$ **else** $odd\ y \Rightarrow P$

$even\ y \wedge y>0 \Rightarrow P \;\Leftarrow\; x:= x \times x.\ y:= y/2.\ P$

$odd\ y \Rightarrow P \;\Leftarrow\; z:= z \times x.\ y:= y-1.\ P$

Each of these refinements is easily proven.

We have made the major improvement, but there are still several minor speedups. We make them partly as an exercise in achieving the greatest speed possible, and mainly as an example of program modification. To begin, if y is even and greater than 0, it is at least 2; after cutting it in half, it is at least 1; let us not waste that information. We re-refine

$even\ y \wedge y>0 \Rightarrow P \;\Leftarrow\; x:= x \times x.\ y:= y/2.\ y>0 \Rightarrow P$

If y is initially odd and 1 is subtracted, then it must become even; let us not waste that information. We re-refine

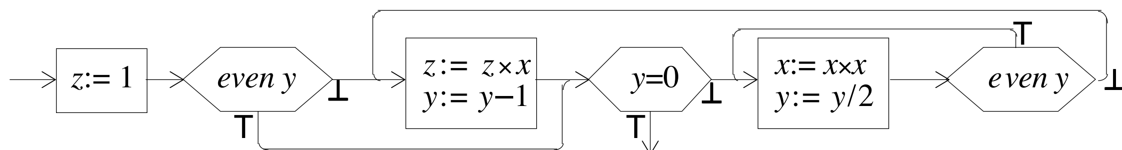
$odd\ y \Rightarrow P \;\Leftarrow\; z:= z \times x.\ y:= y-1.\ even\ y \Rightarrow P$

$even\ y \Rightarrow P \;\Leftarrow\;$ **if** $y = 0$ **then** ok **else** $even\ y \wedge y>0 \Rightarrow P$

And one more very minor improvement: if the program is used to calculate x^0 less often than x to an odd power (a reasonable assumption), it would be better to start with the test for evenness rather than the test for zeroness. We re-refine

$P \;\Leftarrow\;$ **if** $even\ y$ **then** $even\ y \Rightarrow P$ **else** $odd\ y \Rightarrow P$

Program modification, whether to gain speed or for any other purpose, can be dangerously error-prone when practiced without the proper theory. Try writing this program in your favorite standard programming language, starting with the first simple solution, and making the same modifications. The first modification introduces a new case within a loop; the second modification changes one of the cases into an inner loop; the next modification changes the outer loop into a case within the inner loop, with an intermediate exit; the final modification changes the loop entry-point to a choice of two entry-points. The flow chart looks like this.



Without the theory, this sort of program surgery is bound to introduce a few bugs. With the theory we have a better chance of making the modifications correctly because each new refinement is an easy theorem.

Before we consider time, here is the fast exponentiation program again.

$z'=x^y \;\Leftarrow\; z:= 1.\; P$
$P \;\Leftarrow\;$ **if** $even\ y$ **then** $even\ y \Rightarrow P$ **else** $odd\ y \Rightarrow P$
$even\ y \Rightarrow P \;\Leftarrow\;$ **if** $y=0$ **then** ok **else** $even\ y \wedge y>0 \Rightarrow P$
$odd\ y \Rightarrow P \;\Leftarrow\; z:= z\times x.\; y:= y-1.\; even\ y \Rightarrow P$
$even\ y \wedge y>0 \Rightarrow P \;\Leftarrow\; x:= x\times x.\; y:= y/2.\; y>0 \Rightarrow P$
$y>0 \Rightarrow P \;\Leftarrow\;$ **if** $even\ y$ **then** $even\ y \wedge y>0 \Rightarrow P$ **else** $odd\ y \Rightarrow P$

In the recursive time measure, every loop of calls must include a time increment. In this program, a single time increment charged to the call $y>0 \Rightarrow P$ does the trick.

$even\ y \wedge y>0 \Rightarrow P \;\Leftarrow\; x:= x\times x.\; y:= y/2.\; t:= t+1.\; y>0 \Rightarrow P$

To help us decide what time bounds we might try to prove, we can execute the program on some test cases. We find, for each natural n, that $y: 2^n,..2^{n+1} \;\Rightarrow\; t' = t+n$, plus the isolated case $y=0 \;\Rightarrow\; t'=t$. We therefore propose the timing specification

if $y=0$ **then** $t'=t$ **else** $t' = t + floor\ (log\ y)$

where $floor$ is the function that rounds down. We can prove this is the exact execution time, but it is easier to prove the less precise specification T defined as

$T \;=\;$ **if** $y=0$ **then** $t'=t$ **else** $t' \le t + log\ y$

To do so, we need to refine T with exactly the same refinement structure that we used to refine the result $z'=x^y$ so that we can conjoin the result and timing specifications according to Refinement by Parts. We can prove

$T \;\Leftarrow\; z:= 1.\; T$
$T \;\Leftarrow\;$ **if** $even\ y$ **then** T **else** $y>0 \Rightarrow T$
$T \;\Leftarrow\;$ **if** $y=0$ **then** ok **else** $y>0 \Rightarrow T$
$y>0 \Rightarrow T \;\Leftarrow\; z:= z\times x.\; y:= y-1.\; T$
$y>0 \Rightarrow T \;\Leftarrow\; x:= x\times x.\; y:= y/2.\; t:= t+1.\; y>0 \Rightarrow T$
$y>0 \Rightarrow T \;\Leftarrow\;$ **if** $even\ y$ **then** $y>0 \Rightarrow T$ **else** $y>0 \Rightarrow T$

It does not matter that specifications T and $y>0 \Rightarrow T$ are refined more than once. When we conjoin these specifications with the previous result specifications, we find that each specification is refined only once.

The timing can be written as a conjunction

$(y=0 \Rightarrow t'=t) \wedge (y>0 \Rightarrow t' \le t + log\ y)$

and it is tempting to try to prove those two parts separately. Unfortunately we cannot prove the second part of the timing by itself. Separating a specification into parts is not always a successful strategy.

End of Fast Exponentiation

4.2.7 Fibonacci Numbers

In this subsection, we tackle Exercise 217. The definition of the Fibonacci numbers

$$fib\ 0 = 0$$
$$fib\ 1 = 1$$
$$fib\ (n+2) = fib\ n + fib\ (n+1)$$

immediately suggests a recursive function definition

$$fib = 0\to 0 \mid 1\to 1 \mid \langle n\colon nat+2\to fib\ (n-2) + fib\ (n-1)\rangle$$
$$= \langle n\colon nat\to \mathbf{if}\ n<2\ \mathbf{then}\ n\ \mathbf{else}\ fib\ (n-2) + fib\ (n-1)\rangle$$

We did not include functions in our programming language, so we still have some work to do. Besides, the functional solution we have just given has exponential execution time, and we can do much better.

For $n \ge 2$, we can find a Fibonacci number if we know the previous pair of Fibonacci numbers. That suggests we keep track of a pair of numbers. Let x, y, and n be natural variables. We refine

$$x' = fib\ n \Leftarrow P$$

where P is the problem of finding a pair of Fibonacci numbers.

$$P = x' = fib\ n \wedge y' = fib\ (n+1)$$

When $n=0$, the solution is easy. When $n\ge 1$, we can decrease it by 1, find a pair of Fibonacci numbers at that previous argument, and then move x and y along one place.

$$P \Leftarrow \mathbf{if}\ n=0\ \mathbf{then}\ (x:= 0.\ y:= 1)\ \mathbf{else}\ (n:= n-1.\ P.\ x'=y \wedge y' = x+y)$$

To move x and y along we need another variable. We could use a new variable, but we already have n; is it safe to use n for this purpose? The specification $x'=y \wedge y' = x+y$ allows n to change, so we can use it if we want.

$$x'=y \wedge y' = x+y \Leftarrow n:= x.\ x:= y.\ y:= n+y$$

The time for this solution is linear. To prove it, we keep the same refinement structure, but we replace the specifications with new ones concerning time. We replace P by $t' = t+n$ and add $t:= t+1$ in front of its use; we also change $x'=y \wedge y = x+y$ into $t'=t$.

$$t' = t+n \Leftarrow \mathbf{if}\ n=0\ \mathbf{then}\ (x:= 0.\ y:= 1)\ \mathbf{else}\ (n:= n-1.\ t:= t+1.\ t' = t+n.\ t'=t)$$
$$t'=t \Leftarrow n:= x.\ x:= y.\ y:= n+y$$

Linear time is a lot better than exponential time, but we can do even better. Exercise 217 asks for a solution with logarithmic time. To get it, we need to take the hint offered in the exercise and use the equations

$$fib(2\times k + 1) = fib\ k^2 + fib(k+1)^2$$
$$fib(2\times k + 2) = 2 \times fib\ k \times fib(k+1) + fib(k+1)^2$$

These equations allow us to find a pair $fib(2\times k + 1), fib(2\times k + 2)$ in terms of a previous pair $fib\ k, fib(k+1)$ at half the argument. We refine

$$P \Leftarrow \mathbf{if}\ n=0\ \mathbf{then}\ (x:= 0.\ y:= 1)$$
$$\mathbf{else\ if}\ even\ n\ \mathbf{then}\ even\ n \wedge n>0 \Rightarrow P$$
$$\mathbf{else}\ odd\ n \Rightarrow P$$

Let's take the last new problem first. If n is odd, we can cut it down from $2\times k + 1$ to k by the assignment $n:= (n-1)/2$, then call P to obtain $fib\ k$ and $fib(k+1)$, then use the equations to obtain $fib(2\times k + 1)$ and $fib(2\times k + 2)$.

$$odd\ n \Rightarrow P \Leftarrow n:= (n-1)/2.\ P.\ x' = x^2 + y^2 \wedge y' = 2\times x\times y + y^2$$

The case $even\ n \wedge n>0$ is a little harder. We can decrease n from $2\times k + 2$ to k by the assignment $n:= n/2 - 1$, then call P to obtain $fib\ k$ and $fib(k+1)$, then use the equations to obtain $fib(2\times k + 1)$ and $fib(2\times k + 2)$ as before, but this time we want $fib(2\times k + 2)$ and $fib(2\times k + 3)$. We can get $fib(2\times k + 3)$ as the sum of $fib(2\times k + 1)$ and $fib(2\times k + 2)$.

$even\ n \land n{>}0 \Rightarrow P \ \Leftarrow\ n{:=}\ n/2-1.\ P.\ x' = 2{\times}x{\times}y + y^2 \ \land\ y' = x^2 + y^2 + x'$

The remaining two problems to find x' and y' in terms of x and y require another variable as before, and as before, we can use n .

$x' = x^2 + y^2 \ \land\ y' = 2{\times}x{\times}y + y^2 \ \Leftarrow\ n{:=}\ x.\ x{:=}\ x^2 + y^2.\ y{:=}\ 2{\times}n{\times}y + y^2$

$x' = 2{\times}x{\times}y + y^2 \ \land\ y' = x^2 + y^2 + x' \ \Leftarrow\ n{:=}\ x.\ x{:=}\ 2{\times}x{\times}y + y^2.\ y{:=}\ n^2 + y^2 + x$

To prove that this program is now logarithmic time, we define time specification

$T \ = \ t' \le t + log\ (n+1)$

and we put $t{:=}\ t+1$ before calls to T . We must now prove

$T \ \Leftarrow\ \textbf{if}\ n{=}0\ \textbf{then}\ (x{:=}\ 0.\ y{:=}\ 1)\ \textbf{else if}\ even\ n\ \textbf{then}\ even\ n \land n{>}0 \Rightarrow T\ \textbf{else}\ odd\ n \Rightarrow T$

$odd\ n \Rightarrow T \ \Leftarrow\ n{:=}\ (n-1)/2.\ t{:=}\ t+1.\ T.\ t'{=}t$

$even\ n \land n{>}0 \Rightarrow T \ \Leftarrow\ n{:=}\ n/2-1.\ t{:=}\ t+1.\ T.\ t'{=}t$

$t'{=}t \ \Leftarrow\ n{:=}\ x.\ x{:=}\ x^2 + y^2.\ y{:=}\ 2{\times}n{\times}y + y^2$

$t'{=}t \ \Leftarrow\ n{:=}\ x.\ x{:=}\ 2{\times}x{\times}y + y^2.\ y{:=}\ n^2 + y^2 + x$

The first one and last two are easy. Here are the other two.

$(odd\ n \Rightarrow t' \le t + log\ (n+1)) \ \Leftarrow\ (n{:=}\ (n-1)/2.\ t{:=}\ t+1.\ t' \le t + log\ (n+1).\ t'{=}t)$

$=\ (odd\ n \Rightarrow t' \le t + log\ (n+1)) \ \Leftarrow\ t' \le t + 1 + log\ ((n-1)/2+1)$

note that $(a \Rightarrow b) \Leftarrow c \ = \ a \Rightarrow (b \Leftarrow c)$

$=\ odd\ n \Rightarrow (t' \le t + log\ (n+1) \ \Leftarrow\ t' \le t + 1 + log\ ((n-1)/2+1))$ connection law

$\Leftarrow\ odd\ n \ \Rightarrow\ 1 + log\ ((n-1)/2+1) \le log\ (n+1)$ logarithm law

$=\ odd\ n \ \Rightarrow\ log\ (n-1+2) \le log\ (n+1)$ arithmetic

$=\ odd\ n \ \Rightarrow\ log\ (n+1) \le log\ (n+1)$ reflexivity and base

$=\ \top$

$(even\ n \land n{>}0 \Rightarrow t' \le t + log\ (n+1)) \Leftarrow (n{:=}\ n/2-1.\ t{:=}\ t+1.\ t' \le t + log\ (n+1).\ t'{=}t)$

by the same steps

$=\ even\ n \land n{>}0 \ \Rightarrow\ 1 + log\ (n/2-1+1) \le log\ (n+1)$

$=\ even\ n \land n{>}0 \ \Rightarrow\ log\ n \le log\ (n+1)$

$=\ \top$

End of Fibonacci Numbers

Finding the execution time of any program can always be done by transforming the program into a function that expresses the execution time. To illustrate how, we do Exercise 216 (roller coaster), which is a famous program whose execution time is considered to be unknown. Let n be a natural variable. Then, including recursive time,

$n'{=}1 \ \Leftarrow\ \textbf{if}\ n{=}1\ \textbf{then}\ ok$
$\quad \textbf{else if}\ even\ n\ \textbf{then}\ (n{:=}\ n/2.\ t{:=}\ t+1.\ n'{=}1)$
$\quad \textbf{else}\ (n{:=}\ 3{\times}n + 1.\ t{:=}\ t+1.\ n'{=}1)$

It is not even known whether the execution time is finite for all $n{>}0$.

We can express the execution time as $f\,n$, where function f must satisfy

$t'{=}t{+}fn \ \Leftarrow\ \textbf{if}\ n{=}1\ \textbf{then}\ ok$
$\quad \textbf{else if}\ even\ n\ \textbf{then}\ (n{:=}\ n/2.\ t{:=}\ t+1.\ t'{=}t{+}fn)$
$\quad \textbf{else}\ (n{:=}\ 3{\times}n + 1.\ t{:=}\ t+1.\ t'{=}t{+}fn)$

which can be simplified to

$f\,n \ = \ \textbf{if}\ n{=}1\ \textbf{then}\ 0$
$\quad \textbf{else if}\ even\ n\ \textbf{then}\ 1 + f\,(n/2)$
$\quad \textbf{else}\ 1 + f\,(3{\times}n + 1)$

Thus we have an exact definition of the execution time. So why is the execution time considered to be unknown?

If the execution time of some program is n^2, we consider that the execution time of that program is known. Why is n^2 accepted as a time bound, and $f\ n$ as defined above not accepted? Before answering, we suggest several non-reasons. The reason is not that f is defined recursively; the square function is defined in terms of multiplication, and multiplication is defined recursively. The reason cannot be that n^2 is well behaved (finite, monotonic, and smooth), while f jumps around wildly; every jump and change of value in f is there to fit the original program's execution time perfectly, and we shouldn't disqualify f just because it is a perfect bound. One might propose the length of time it takes to compute the time bound as a reason to reject f. Since it takes exactly as long to compute the time bound $f\ n$ as to run the program, we might as well just run the original program and look at our watch and say that's the time bound. But $log\ log\ n$ is accepted as a time bound even though it takes longer than $log\ log\ n$ to compute $log\ log\ n$.

The reason seems to be that function f is unfamiliar; it has not been well studied and we don't know much about it. If it were as well studied and familiar as square, we would accept it as a time bound.

We earlier looked at linear search in which we have to find the first occurrence of a given item in a given list. Suppose now that the list L is infinitely long, and we are told that there is at least one occurrence of the item x in the list. The desired result can be simplified to

$$\neg\ x: L\ (0,..h') \ \land\ Lh'=x$$

and the program can be simplified to

$$\neg\ x: L\ (0,..h') \ \land\ Lh'=x \ \Leftarrow\ h:= 0.\ \neg\ x: L\ (h,..h') \ \land\ Lh'=x$$
$$\neg\ x: L\ (h,..h') \ \land\ Lh'=x \ \Leftarrow\ \mathbf{if}\ Lh=x\ \mathbf{then}\ ok\ \mathbf{else}\ (h:= h+1.\ \neg\ x: L\ (h,..h') \ \land\ Lh'=x)$$

Adding recursive time, we can prove

$$t'=t+h' \ \Leftarrow\ h:= 0.\ t'=t+h'-h$$
$$t'=t+h'-h \ \Leftarrow\ \mathbf{if}\ Lh=x\ \mathbf{then}\ ok\ \mathbf{else}\ (h:= h+1.\ t:= t+1.\ t'=t+h'-h)$$

The execution time is h'. Is this acceptable as a time bound? It gives us no indication of how long to wait for a result. On the other hand, there is nothing more to say about the execution time. The defect is in the given information: that x occurs somewhere, with no indication where.

End of Time

4.3 Space

Our example to illustrate space calculation is Exercise 212: the problem of the Towers of Hanoi. There are 3 towers and n disks. The disks are graduated in size; disk 0 is the smallest and disk $n-1$ is the largest. Initially tower A holds all n disks, with the largest disk on the bottom, proceding upwards in order of size to the smallest disk on top. The task is to move all the disks from tower A to tower B, but you can move only one disk at a time, and you must never put a larger disk on top of a smaller one. In the process, you can make use of tower C as intermediate storage.

Our solution is *MovePile* "A" "B" "C" where we refine *MovePile* as follows.

$MovePile\ from\ to\ using \Leftarrow$ **if** $n=0$ **then** ok
else ($n:= n-1.$
$MovePile\ from\ using\ to.$
$MoveDisk\ from\ to.$
$MovePile\ using\ to\ from.$
$n:= n+1$)

Procedure *MovePile* moves all n disks, one at a time, never putting a larger disk on top of a smaller one. Its first parameter *from* is the tower where the n disks are initially; its second parameter *to* is the tower where the n disks are finally; its last parameter *using* is the tower used as intermediate storage. It accomplishes its task as follows. If there are any disks to move, it starts by ignoring the bottom disk ($n:= n-1$). Then a recursive call moves the remaining pile (all but the bottom disk, one at a time, never putting a larger disk on top of a smaller one) from the *from* tower to the *using* tower (using the *to* tower as intermediate storage). Then *MoveDisk* causes a robot arm to move the bottom disk. If you don't have a robot arm, then *MoveDisk* can just print out what the arm should do:

"Move disk "; *nat2text n*; " from tower "; *from*; " to tower "; *to*

Then a recursive call moves the remaining pile (all but the bottom disk, one at a time, never putting a larger disk on top of a smaller one) from the *using* tower to the *to* tower (using the *from* tower as intermediate storage). And finally n is restored to its original value.

To formalize *MovePile* and *MoveDisk* and to prove that the rules are obeyed and the disks end in the right place, we need to describe formally the position of the disks on the towers. But that is not the point of this section. Our concern is just the time and space requirements, so we will ignore the disk positions and the parameters *from* , *to* , and *using* . All we can prove at the moment is that if *MoveDisk* satisfies $n'=n$, so does *MovePile* .

To measure time, we add a time variable t , and use it to count disk moves. We suppose that *MoveDisk* takes time 1 , and that is all it does that we care about at the moment, so we replace it by $t:= t+1$. We now prove that the execution time is $2^n - 1$ by replacing *MovePile* with the specification $t:= t + 2^n - 1$. We prove

$t:= t + 2^n - 1 \Leftarrow$ **if** $n=0$ **then** ok
else ($n:= n-1.$
$t:= t + 2^n - 1.$
$t:= t+1.$
$t:= t + 2^n - 1.$
$n:= n+1$)

by cases. First case, starting with its right side:

$n=0 \wedge ok$ — expand ok
$= \quad n=0 \wedge n'=n \wedge t'=t$ — arithmetic
$\Rightarrow \quad t:= t + 2^n - 1$

Second case, starting with its right side:

$n>0 \wedge (n:= n-1.\ t:= t + 2^n - 1.\ t:= t+1.\ t:= t + 2^n - 1.\ n:= n+1)$
drop conjunct $n>0$; expand final assignment
$\Rightarrow \quad n:= n-1.\ t:= t + 2^n - 1.\ t:= t+1.\ t:= t + 2^n - 1.\ n'=n+1 \wedge t'=t$
use substitution law repeatedly from right to left
$= \quad n'=n-1+1 \wedge t'=t+2^{n-1}-1+1+2^{n-1}-1$ — simplify
$= \quad n'=n \wedge t'=t+2^n-1$
$= \quad t:= t + 2^n - 1$

To talk about the memory space used by a computation, we just add a space variable s. Like the time variable t, s is not part of the implementation, but only used in specifying and calculating space requirements. We use s for the space occupied initially at the start of execution, and s' for the space occupied finally at the end of execution. Any program may be used as part of a larger program, and it may not be the first part, so we cannot assume that the initial space occupied is 0, just as we cannot assume that a computation begins at time 0. In our example, the program calls itself recursively, and the recursive invocations begin at different times with different occupied space from the main (nonrecursive) invocation.

To allow for the possibility that execution endlessly consumes space, we take the domain of space to be the natural numbers extended with ∞. Wherever space is being increased, we insert $s{:=}\ s+$(the increase) to adjust s appropriately, and wherever space is being decreased, we insert $s{:=}\ s-$(the decrease). In our example, the recursive calls are not the last action in the refinement; they require that a return address be pushed onto a stack at the start of the call, and popped off at the end. Considering only space, ignoring time and disk movements, we can prove

$$
\begin{aligned}
s'=s \;\Leftarrow\;\; & \textbf{if } n=0 \textbf{ then } ok\\
& \textbf{else } (\;\; n{:=}\ n-1.\\
& \qquad s{:=}\ s+1.\;\; s'=s.\;\; s{:=}\ s-1.\\
& \qquad ok.\\
& \qquad s{:=}\ s+1.\;\; s'=s.\;\; s{:=}\ s-1.\\
& \qquad n{:=}\ n+1\;)
\end{aligned}
$$

which says that the space occupied is the same at the end as at the start.

It is comforting to know there are no "space leaks", but this does not tell us much about the space usage. There are two measures of interest: the maximum space occupied, and the average space occupied.

4.3.0 Maximum Space

Let m be the maximum space occupied before the start of execution (remember that any program may be part of a larger program that started execution earlier), and m' be the maximum space occupied by the end of execution. Wherever space is being increased, we insert $m{:=}\ max\ m\ s$ to keep m current. There is no need to adjust m at a decrease in space. In our example, we want to prove that the maximum space occupied is n. However, in a larger context, it may happen that the starting space is not 0, so we specify $m' = s+n$. We can assume that at the start $m \ge s$, since m is supposed to be the maximum value of s, but it may happen that the starting value of m is already greater than $s+n$, so the specification becomes $m\ge s \Rightarrow (m{:=}\ max\ m\ (s+n))$.

$$
\begin{aligned}
& m\ge s \Rightarrow (m{:=}\ max\ m\ (s+n)) \;\Leftarrow\\
& \qquad \textbf{if } n=0 \textbf{ then } ok\\
& \qquad \textbf{else } (\;\; n{:=}\ n-1.\\
& \qquad\qquad s{:=}\ s+1.\;\; m{:=}\ max\ m\ s.\;\; m\ge s \Rightarrow (m{:=}\ max\ m\ (s+n)).\;\; s{:=}\ s-1.\\
& \qquad\qquad ok.\\
& \qquad\qquad s{:=}\ s+1.\;\; m{:=}\ max\ m\ s.\;\; m\ge s \Rightarrow (m{:=}\ max\ m\ (s+n)).\;\; s{:=}\ s-1.\\
& \qquad\qquad n{:=}\ n+1\;)
\end{aligned}
$$

Before proving this, let's simplify the long line that occurs twice.

$s{:=}\ s+1.\ \ m{:=}\ max\ m\ s.\ \ m{\ge}s \Rightarrow (m{:=}\ max\ m\ (s+n)).\ \ s{:=}\ s-1$

Use a Condition Law, and expand final assignment

$\Rightarrow\quad s{:=}\ s+1.\ \ m{:=}\ max\ m\ s.\ \ m{\ge}s \Rightarrow (m{:=}\ max\ m\ (s+n).\ \ s'{=}s-1 \land m'{=}m \ \land\ n'{=}n)$

Use Substitution Law

$=\quad s{:=}\ s+1.\ \ m{:=}\ max\ m\ s.\ \ m{\ge}s\ \Rightarrow\ s'{=}s-1\ \land\ m' = max\ m\ (s+n)\ \land\ n'{=}n$

Use Substitution Law

$=\quad s{:=}\ s+1.\ \ (max\ m\ s){\ge}s\ \Rightarrow\ s'{=}s-1\ \land\ m' = max\ (max\ m\ s)\ (s+n)\ \land\ n'{=}n$

Simplify antecedent to $\top$. Also max is associative

$=\quad s{:=}\ s+1.\ \ s'{=}s-1\ \land\ m' = max\ m\ (s+n)\ \land\ n'{=}n$ use Substitution Law

$=\quad s'{=}s\ \land\ m' = max\ m\ (s+1+n)\ \land\ n'{=}n$

$=\quad m{:=}\ max\ m\ (s+1+n)$

The proof of the refinement proceeds in the usual two cases. First,

$n{=}0 \land ok$

$=\quad n'{=}n{=}0\ \land\ s'{=}s\ \land\ m'{=}m$

$\Rightarrow\quad m{\ge}s \Rightarrow (m{:=}\ max\ m\ (s+n))$

And second,

$n{>}0 \land (\ n{:=}\ n-1.$
$\quad s{:=}\ s+1.\ \ m{:=}\ max\ m\ s.\ \ m{\ge}s \Rightarrow (m{:=}\ max\ m\ (s+n)).\ \ s{:=}\ s-1.$
$\quad ok.$
$\quad s{:=}\ s+1.\ \ m{:=}\ max\ m\ s.\ \ m{\ge}s \Rightarrow (m{:=}\ max\ m\ (s+n)).\ \ s{:=}\ s-1.$
$\quad n{:=}\ n+1\ )$ Drop $n{>}0$ and ok. Simplify long lines. Expand final assignment.

$\Rightarrow\quad n{:=}\ n-1.\ \ m{:=}\ max\ m\ (s+1+n).\ \ m{:=}\ max\ m\ (s+1+n).\ \ n'{=}n+1 \land s'{=}s \land m'{=}m$

use Substitution Law three times

$=\quad n'{=}n\ \land\ s'{=}s\ \land\ m' = max\ (max\ m\ (s+n))\ (s+n)$ associative and idempotent laws

$=\quad n'{=}n\ \land\ s'{=}s\ \land\ m' = max\ m\ (s+n$

$\Rightarrow\quad m{\ge}s \Rightarrow (m{:=}\ max\ m\ (s+n))$

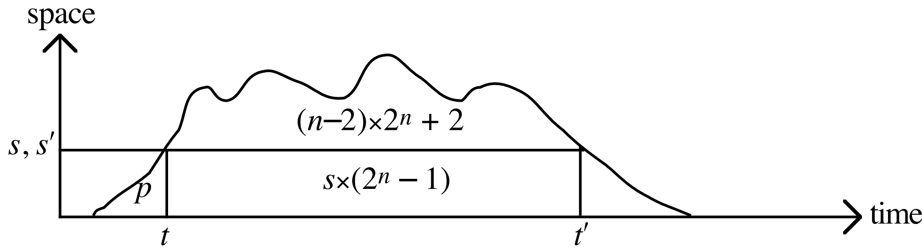
End of Maximum Space

4.3.1 Average Space

To find the average space occupied during a computation, we find the cumulative space-time product, and then divide by the execution time. Let p be the cumulative space-time product at the start of execution, and p' be the cumulative space-time product at the end of execution. We still need variable s, which we adjust exactly as before. We do not need variable t; however, an increase in p occurs where there would be an increase in t, and the increase is s times the increase in t. In the example, where t was increased by 1, we now increase p by s. We prove

$p{:=}\ p + s{\times}(2^n - 1) + (n-2){\times}2^n + 2\ \Leftarrow$
if $n{=}0$ **then** ok
else $(\ \ n{:=}\ n-1.$
$\quad s{:=}\ s+1.\ \ p{:=}\ p + s{\times}(2^n - 1) + (n-2){\times}2^n + 2.\ \ s{:=}\ s-1.$
$\quad p{:=}\ p+s.$
$\quad s{:=}\ s+1.\ \ p{:=}\ p + s{\times}(2^n - 1) + (n-2){\times}2^n + 2.\ \ s{:=}\ s-1.$
$\quad n{:=}\ n+1\ )$

In the specification $p{:=}\ p + s{\times}(2^n - 1) + (n-2){\times}2^n + 2$, the term $s{\times}(2^n - 1)$ is the product of the initial space s and total time $2^n - 1$; it is the increase in the space-time product due to the surrounding computation (which is 0 if s is 0). The additional amount $(n-2){\times}2^n + 2$ is due to our computation. The average space due to our computation is this additional amount divided by the execution time. Thus the average space occupied by our computation is $n + n/(2^n - 1) - 2$.



The proof, as usual, in two parts:

$n=0 \wedge ok$ — expand ok
$= \quad n=0 \wedge n'=n \wedge s'=s \wedge p'=p$ — arithmetic
$\Rightarrow \quad n'=n \wedge s'=s \wedge p' = p + s\times(2^n - 1) + (n-2)\times 2^n + 2$
$= \quad p:= p + s\times(2^n - 1) + (n-2)\times 2^n + 2$

$n>0 \wedge (\ n:= n-1.\ \ s:= s+1.\ \ p:= p + s\times(2^n - 1) + (n-2)\times 2^n + 2.\ \ s:= s-1.\ \ n:= n+1.$
$\quad p:= p+s.$
$\quad n:= n-1.\ \ s:= s+1.\ \ p:= p + s\times(2^n - 1) + (n-2)\times 2^n + 2.\ \ s:= s-1.\ \ n:= n+1\ )$
drop conjunct $n>0$; expand final assignment
$\Rightarrow \quad n:= n-1.\ \ s:= s+1.\ \ p:= p + s\times(2^n - 1) + (n-2)\times 2^n + 2.\ \ s:= s-1.\ \ n:= n+1.\ p:= p+s.$
$\quad n:= n-1.\ s:= s+1.\ p:= p + s\times(2^n-1) + (n-2)\times 2^n + 2.\ s:= s-1.\ n'=n+1 \wedge s'=s \wedge p'=p$
use substitution law 10 times from right to left
$= \quad n'=n \wedge s'=s$
$\wedge \quad p' = p + (s+1)\times(2^{n-1}-1) + (n-3)\times 2^{n-1} + 2 + s + (s+1)\times(2^{n-1}-1) + (n-3)\times 2^{n-1} + 2$
simplify
$= \quad n'=n \wedge s'=s \wedge p' = p + s\times(2^n - 1) + (n-2)\times 2^n + 2$
$= \quad p:= p + s\times(2^n - 1) + (n-2)\times 2^n + 2$

Instead of proving that the average space is exactly $n + n/(2^n - 1) - 2$, it is easier to prove that the average space is bounded above by n . To do so, instead of proving that the space-time product is $s\times(2^n-1) + (n-2)\times 2^n + 2$, we would prove it is at most $(s+n)\times(2^n-1)$. But we leave that as Exercise 212(f).

Putting together all the proofs for the Towers of Hanoi problem, we have

MovePile $\Leftarrow$ **if** $n=0$ **then** ok
else ($n:= n-1.$
$s:= s+1.\ \ m:= max\ m\ s.\ \ MovePile.\ \ s:= s-1.$
$t:= t+1.\ \ p:= p+s.\ \ ok.$
$s:= s+1.\ \ m:= max\ m\ s.\ \ MovePile.\ \ s:= s-1.$
$n:= n+1$)

where *MovePile* is the specification

$n'=n$
$\wedge \quad t' = t + 2^n - 1$
$\wedge \quad s'=s$
$\wedge \quad (m\ge s \Rightarrow m' = max\ m\ (s+n))$
$\wedge \quad p' = p + s\times(2^n - 1) + (n-2)\times 2^n + 2$

End of Average Space

End of Space

End of Program Theory

5 Programming Language

We have been using a very simple programming language consisting of only ok , assignment, **if then else** , dependent (sequential) composition, and refined specifications. In this chapter we enrich our repertoire by considering some of the notations found in some popular languages. We will not consider concurrency (independent composition) and interaction (input and output) just yet; they get their own chapters later.

5.0 Scope

5.0.0 Variable Declaration

The ability to declare a new state variable within a local scope is so useful that it is provided by every decent programming language. A declaration may look something like this:

var x: T

where x is the variable being declared, and T , called the type, indicates what values x can be assigned. A variable declaration applies to what follows it, according to the precedence table on the final page of the book. In program theory, it is essential that each of our notations apply to all specifications, not just to programs. That way we can introduce a local variable as part of the programming process, before its scope is refined.

We can express a variable declaration together with the specification to which it applies as a boolean expression in the initial and final state.

var x: $T\cdot P \quad = \quad \exists x, x'\text{: } T\cdot P$

Specification P is an expression in the initial and final values of all nonlocal (already declared) variables plus the newly declared local variable. Specification **var** x: $T\cdot P$ is an expression in the nonlocal variables only. For a variable declaration to be implementable, its type must be nonempty. As a simple example, suppose the nonlocal variables are integer variables y and z . Then

var x: $int\cdot\ x:= 2.\ y:= x+z$

$= \quad \exists x, x'\text{: } int\cdot\ x'=2 \ \land\ y' = 2+z \ \land\ z'=z$

$= \quad y' = 2+z \ \land\ z'=z$

According to our definition of variable declaration, the initial value of the local variable is an arbitrary value of its type.

var x: $int\cdot\ y:= x$

$= \quad \exists x, x'\text{: } int\cdot\ x'=x \land y'=x \land z'=z$

$= \quad z'=z$

which says that z is unchanged. Variable x is not mentioned because it is a local variable, and variable y is not mentioned because its final value is unknown. However

var x: $int\cdot\ y:= x–x$

$= \quad y'=0 \land z'=z$

In some languages, a newly declared variable has a special value called “the undefined value” which cannot participate in any expressions. To write such declarations as boolean expressions, we introduce the expression $undefined$ but we do not give any axioms about it, so nothing can be proven about it. Then

var x: $T\cdot P \quad = \quad \exists x\text{: } undefined\cdot \exists x'\text{: } T, undefined\cdot P$

For this kind of variable declaration, it is not necessary for the type to be nonempty.

An initializing assignment is easily defined in the same way.

$\textbf{var } x\text{: } T := e\cdot\ P \quad = \quad \exists x\text{: } e\cdot \exists x'\text{: } T\cdot P$

assuming e is of type T .

If we are accounting for space usage, a variable declaration should be accompanied by an increase to the space variable s at the start of the scope of the declaration, and a corresponding decrease to s at the end of the scope.

As in many programming languages, we can declare several variables in one declaration. For example,

$\textbf{var } x, y, z\text{: } T\cdot P \quad = \quad \exists x, x', y, y', z, z'\text{: } T\cdot P$

End of Variable Declaration

It is a service to the world to make variable declarations as local as possible. That way, the state space outside the local scope is not polluted with unwanted variables. Inside the local scope, there are all the nonlocal variables plus the local ones; there are more variables to keep track of locally.

5.0.1 Variable Suspension

We may wish, temporarily, to narrow our focus to a part of the state space. If the part is x and y , we indicate this with the notation

$\textbf{frame } x, y$

It applies to what follows it, according to the precedence table on the final page of the book, just like **var** . The **frame** notation is the formal way of saying "and all other variables (even the ones we cannot say because they are covered by local declarations) are unchanged". This is similar to the "import" statement of some languages, though not identical. If the state variables not included in the frame are w and z , then

$\textbf{frame } x, y\cdot P \quad = \quad P \land w'=w \land z'=z$

Within P the state variables are x and y . It allows P to refer to w and z , but only as local constants (mathematical variables, not state variables; there is no w' and no z'). Time and space variables are implicitly assumed to be in all frames, even though they may not be listed explicitly.

End of Variable Suspension

The definitions of ok and assignment using state variables

$ok \quad = \quad x'=x \land y'=y \land ...$

$x:= e \quad = \quad x'=e \land y'=y \land ...$

were partly informal, using three dots to say "and other conjuncts for other state variables". If we had defined **frame** first, we could have defined them formally as follows:

$ok \quad = \quad \textbf{frame}\cdot \top$

$x:= e \quad = \quad \textbf{frame } x\cdot\ x'=e$

We specified the list summation problem in the previous chapter as $s' = \Sigma L$. We took s to be a state variable, and L to be a constant. We might have preferred the specification $s:= \Sigma L$ saying that s has the right final value and that all other variables are unchanged, but our solution included a variable n which began at 0 and ended at $\#L$. We now have the formal notations needed.

$s:= \Sigma L \quad = \quad \textbf{frame } s\cdot\ \ \textbf{var } n\text{: } nat\cdot\ \ s' = \Sigma L$

First we reduce the state space to s ; if L was a state variable, it is now a constant. Next we introduce local variable n . Then we proceed as before.

End of Scope

5.1 Data Structures

5.1.0 Array

In most popular programming languages there is the notion of subscripted variable, or indexed variable, usually called an array. Each element of an array is a variable. Element 2 of array A can be assigned the value 3 by a notation such as

$$A(2):= 3$$

Perhaps the brackets are square; let us dispense with the brackets. We can write an array element assignment as a boolean expression in the initial and final state as follows. Let A be an array name, let i be any expression of the index type, and let e be any expression of the element type. Then

$$Ai:= e \quad = \quad A'i=e \ \land \ (\forall j\cdot j\neq i \Rightarrow A'j=Aj) \ \land \ x'=x \ \land \ y'=y \ \land \ ...$$

This says that after the assignment, element i of A equals e, all other elements of A are unchanged, and all other variables are unchanged. If you are unsure of the placement of the primes, consider the example

$$\begin{array}{ll} & A(A2):= 3 \\ = & A'(A2) = 3 \ \land \ (\forall j\cdot j\neq A2 \Rightarrow A'j=Aj) \ \land \ x'=x \ \land \ y'=y \ \land \ ... \end{array}$$

The Substitution Law

$$x:= e.\ P \quad = \quad (\text{for } x \text{ substitute } e \text{ in } P)$$

is very useful, but unfortunately it does not work for array element assignment. For example,

$$A2:= 3.\ \ i:= 2.\ \ Ai:= 4.\ \ Ai=A2$$

should equal $\top$, because $i=2$ just before the final boolean expression, and $A2=A2$ certainly equals $\top$. If we try to apply the Substitution Law, we get

$$\begin{array}{lll} & A2:= 3.\ \ i:= 2.\ \ Ai:= 4.\ \ Ai=A2 & \text{invalid use of substitution law} \\ = & A2:= 3.\ \ i:= 2.\ \ 4=A2 & \text{valid use of substitution law} \\ = & A2:= 3.\ \ 4=A2 & \text{invalid use of substitution law} \\ = & 4=3 & \\ = & \bot & \end{array}$$

Here is a second example of the failure of the Substitution Law for array elements.

$$A2:= 2.\ \ A(A2):= 3.\ \ A2=2$$

This should equal $\bot$ because $A2=3$ just before the final boolean expression. But the Substitution Law says

$$\begin{array}{lll} & A2:= 2.\ \ A(A2):= 3.\ \ A2=2 & \text{invalid use of substitution law} \\ = & A2:= 2.\ \ A2=2 & \text{invalid use of substitution law} \\ = & 2=2 & \\ = & \top & \end{array}$$

The Substitution Law works only when the assignment has a simple name to the left of $:=$. Fortunately we can always rewrite an array element assignment in that form.

$$\begin{array}{ll} & Ai:= e \\ = & A'i=e \ \land \ (\forall j\cdot j\neq i \Rightarrow A'j=Aj) \ \land \ x'=x \ \land \ y'=y \ \land \ ... \\ = & A' = i\rightarrow e \mid A \ \land \ x'=x \ \land \ y'=y \ \land \ ... \\ = & A:= i\rightarrow e \mid A \end{array}$$

Let us look again at the examples for which the Substitution Law did not work, this time using the notation $A:= i \to e \mid A$.

$$
\begin{aligned}
& A:= 2\to 3 \mid A.\ \ i:= 2.\ \ A:= i\to 4 \mid A.\ \ Ai = A2 \\
= \quad & A:= 2\to 3 \mid A.\ \ i:= 2.\ \ (i\to 4 \mid A)i = (i\to 4 \mid A)2 \\
= \quad & A:= 2\to 3 \mid A.\ \ (2\to 4 \mid A)2 = (2\to 4 \mid A)2 \\
= \quad & A:= 2\to 3 \mid A.\ \ \top \\
= \quad & \top
\end{aligned}
$$

$$
\begin{aligned}
& A:= 2\to 2 \mid A.\ \ A:= A2\to 3 \mid A.\ \ A2=2 \\
= \quad & A:= 2\to 2 \mid A.\ \ (A2\to 3 \mid A)2 = 2 \\
= \quad & ((2\to 2 \mid A)2\to 3 \mid 2\to 2 \mid A)\ 2 = 2 \\
= \quad & (2\to 3 \mid 2\to 2 \mid A)\ 2 = 2 \\
= \quad & 3 = 2 \\
= \quad & \bot
\end{aligned}
$$

The only thing to remember about array element assignment is this: change $Ai:= e$ to $A:= i\to e \mid A$ before applying any programming theory. A two-dimensional array element assignment $Aij:= e$ must be changed to $A:= (i; j)\to e \mid A$, and similarly for more dimensions.

End of Array

5.1.1 Record

Without inventing anything new, we can already build records, also known as structures, similar to those found in several languages. Let us define *person* as follows.

$$
\begin{aligned}
person \ = \quad & \text{"name"} \to text \\
\mid \quad & \text{"age"} \to nat
\end{aligned}
$$

We declare

var p: *person*

and assign p as follows.

p:= "name" → "Josh" | "age" → 17

In languages with records (or structures), a component (or field) is assigned the same way we make an array element assignment. For example,

p "age":= 18

Just as for array element assignment, the Substitution Law does not work for record components. And the solution is also the same; just rewrite it like this:

p:= "age" → 18 | p

No new theory is needed for records.

End of Record

End of Data Structures

5.2 Control Structures

5.2.0 While Loop

The **while**-loop of several languages has a syntax similar to

while b **do** P

where b is boolean and P is a specification. To execute it, evaluate b , and if its value is $\bot$ then you're done, but if its value is $\top$ then execute P and start over. We do not define the **while**-loop as a specification the way we have defined previous programming notations. Instead, if W is an implementable specification, we consider

$W \Leftarrow$ **while** b **do** P

to be an abbreviation of

$W \Leftarrow$ **if** b **then** $(P.\ W)$ **else** ok

For example, to prove

$s' = s + \Sigma\, L\, [n;..\#L] \ \wedge\ t' = t + \#L - n \ \Leftarrow$
while $n \neq \#L$ **do** $(s{:=}\ s + Ln.\ n{:=}\ n+1.\ t{:=}\ t+1)$

prove instead

$s' = s + \Sigma\, L\, [n;..\#L] \ \wedge\ t' = t + \#L - n \ \Leftarrow$
if $n \neq \#L$ **then** $(s{:=}\ s + Ln.\ n{:=}\ n+1.\ t{:=}\ t+1.\ s' = s + \Sigma\, L\, [n;..\#L] \ \wedge\ t' = t+\#L-n)$
else ok

During programming, we may happen to refine a specification W by **if** b **then** $(P.\ W)$ **else** ok . If so, we may abbreviate the refinement using a while-loop. This is particularly valuable when the implementation of call is poor, and does not use a branch instruction in this situation.

This account of **while**-loops is adequate for practical purposes: it tells us how we can use them in programming. But it does not allow us to prove as much as we might like; for example, we cannot prove

while b **do** P $=$ **if** b **then** $(P.$ **while** b **do** $P)$ **else** ok

A different account of **while**-loops is given in Chapter 6.

Exercise 265: Consider the following program in natural variables x and y .

while $\neg\ x{=}y{=}0$ **do**
if $y{>}0$ **then** $y{:=}\ y-1$
else $(x{:=}\ x-1.$ **var** $n{:}\ nat\cdot\ y{:=}\ n)$

This loop decreases y until it is 0 ; then it decreases x by 1 and assigns an arbitrary natural number to y ; then again it decreases y until it is 0 ; and again it decreases x by 1 and assigns an arbitrary natural number to y ; and so on until both x and y are 0 . The problem is to find a time bound. So we introduce time variable t , and rewrite the loop in refinement form.

$P \Leftarrow$ **if** $x{=}y{=}0$ **then** ok
else if $y{>}0$ **then** $(y{:=}\ y-1.\ t{:=}\ t+1.\ P)$
else $(x{:=}\ x-1.\ (\exists n\cdot\ y{:=}\ n).\ t{:=}\ t+1.\ P)$

The execution time depends on x and on y and on the arbitrary values assigned to y . That means we need n to be nonlocal so we can refer to it in the specification P . But a nonlocal n would have a single arbitrary initial value that would be assigned to y every time x is decreased, whereas in our computation y may be assigned different arbitrary values every time x is decreased. So we change n into a function f of x . (Variable x never repeats a value; if it did repeat, we would have to make f be a function of time.)

Let $f{:}\ nat \to nat$. We say nothing more about f , so it is a completely arbitrary function from nat to nat . Introducing f gives us a way to refer to the arbitrary values, but does not say anything about when or how those arbitrary values are chosen. Let $s = \Sigma f\,[0;..x]$, which says s is the sum of the first x values of f . We prove

$t' = t{+}x{+}y{+}s \ \Leftarrow$ **if** $x{=}y{=}0$ **then** ok
else if $y{>}0$ **then** $(y{:=}\ y-1.\ t{:=}\ t+1.\ t' = t{+}x{+}y{+}s)$
else $(x{:=}\ x-1.\ y{:=}\ fx.\ t{:=}\ t+1.\ t' = t{+}x{+}y{+}s)$

The proof is in three cases.

$x=y=0 \land ok$
$\Longrightarrow\quad x=y=s=0 \land t'=t$
$\Longrightarrow\quad t' = t+x+y+s$

$y>0 \land (y:= y-1.\ t:= t+1.\ t' = t+x+y+s)$ substitution law twice
$=\quad y>0 \land t' = t+1+x+y-1+s$
$\Longrightarrow\quad t' = t+x+y+s$

$x>0 \land y=0 \land (x:= x-1.\ y:= f\,x.\ t:= t+1.\ t' = t+x+y+s)$ substitution law 3 times
$=\quad x>0 \land y=0 \land t' = t+1+x-1+f\,(x-1)+\Sigma f\,[0;..x-1]$
$\Longrightarrow\quad t' = t+x+y+s$

The execution time of the program is $x + y +$ (the sum of x arbitrary natural numbers) .

End of While Loop

5.2.1 Loop with Exit

Some languages provide a command to jump out of the middle of a loop. The syntax for a loop in such a language might be

loop P **end**

with the additional syntax

exit when b

allowed within P , where b is boolean. Sometimes the word "break" is used instead of "exit". As in Subsection 5.2.0, we consider refinement by a loop with exits to be an alternative notation. For example, if L is an implementable specification, then

$L \Leftarrow$ **loop**
  A.
  exit when b.
  C
end

is an alternative notation for

$L \Leftarrow A.$ **if** b **then** ok **else** $(C.\ L)$

Programmers who use loop constructs sometimes find that they reach their goal deep within several nested loops. The problem is how to get out. A boolean variable can be introduced for the purpose of recording whether the goal has been reached, and tested at each iteration of each level of loop to decide whether to continue or exit. Or a **go to** can be used to jump directly out of all the loops, saving all tests. Or perhaps the programming language provides a specialized **go to** for this purpose: **exit** n **when** b which means exit n loops when b is satisfied. For example, we may have something like this:

$P \Leftarrow$ **loop**
  A.
  loop
    B.
    exit 2 **when** c.
    D
  end.
  E
end

The refinement structure corresponding to this loop is

$P \Leftarrow A.\ Q$

$Q \Leftarrow B.$ **if** c **then** ok **else** $(D.\ Q)$

for some appropriately defined Q . It has often been suggested that every loop should have a specification, but the loop construct does not require it. The refinement structure does require it.

The preceding example had a deep exit but no shallow exit, leaving E stranded in a dead area. Here is an example with both deep and shallow exits.

$P \Leftarrow$ **loop**
    A.
    exit 1 **when** b.
    C.
    loop
        D.
        exit 2 **when** e.
        F.
        exit 1 **when** g.
        H
    end.
    I
end

The refinement structure corresponding to this loop is

$P \Leftarrow A.$ **if** b **then** ok **else** $(C.\ Q)$

$Q \Leftarrow D.$ **if** e **then** ok **else** $(F.$ **if** g **then** $(I.\ P)$ **else** $(H.\ Q))$

for some appropriately defined Q .

Loops with exits can always be translated easily to a refinement structure. But the reverse is not true; some refinement structures require the introduction of new variables and even whole data structures to encode them as loops with exits.

End of Exit Loop

5.2.2 Two-Dimensional Search

To illustrate the preceding subsection, we can do Exercise 157: Write a program to find a given item in a given 2-dimensional array. The execution time must be linear in the product of the dimensions.

Let the array be A , let its dimensions be n by m , and let the item we seek be x . We will indicate the position of x in A by the final values of natural variables i and j . If x occurs more than once, any of its positions will do. If it does not occur, we will indicate that by assigning i and j the values n and m respectively. The problem, except for time, is then P where

$P =$ **if** $x\colon A\ (0,..n)\ (0,..m)$ **then** $x = A\ i'\ j'$ **else** $i'=n \land j'=m$

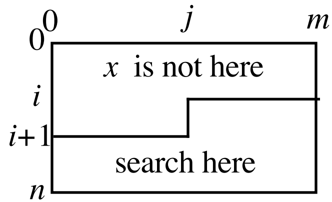
We may as well search row 0 first, then row 1 , and so on. Accordingly, we define Q to specify the search from row i onward:

$Q =$ **if** $x\colon A\ (i,..n)\ (0,..m)$ **then** $x = A\ i'\ j'$ **else** $i'=n \land j'=m$

Within each row, we search the columns in order, and so we define R to specify the search from row i column j onward:

$R =$ **if** $x\colon A\ i\ (j,..m),\ A\ (i+1,..n)\ (0,..m)$ **then** $x = A\ i'\ j'$ **else** $i'=n \land j'=m$

The expression $A\ i\ (j,..m),\ A\ (i+1,..n)\ (0,..m)$ represents the items in the bottom region of the following picture:



We now solve the problem in five easy pieces.

$P \Leftarrow i:= 0.\ \ i{\le}n \Rightarrow Q$

$i{\le}n \Rightarrow Q \Leftarrow$ **if** $i{=}n$ **then** $j:= m$ **else** $i{<}n \Rightarrow Q$

$i{<}n \Rightarrow Q \Leftarrow j:= 0.\ \ i{<}n \wedge j{\le}m \Rightarrow R$

$i{<}n \wedge j{\le}m \Rightarrow R \Leftarrow$ **if** $j{=}m$ **then** $(i:= i{+}1.\ \ i{\le}n \Rightarrow Q)$ **else** $i{<}n \wedge j{<}m \Rightarrow R$

$i{<}n \wedge j{<}m \Rightarrow R \Leftarrow$ **if** $A\ i\ j = x$ **then** *ok* **else** $(j:= j{+}1.\ \ i{<}n \wedge j{\le}m \Rightarrow R)$

It is easier to see the execution pattern when we retain only enough information for execution. The non-program specifications are needed for understanding the purpose, and for proof, but not for execution. To a compiler, the program appears as follows:

$P \Leftarrow i:= 0.\ \ L0$
$L0 \Leftarrow$ **if** $i{=}n$ **then** $j:= m$ **else** $(j:= 0.\ \ L1)$
$L1 \Leftarrow$ **if** $j{=}m$ **then** $(i:= i{+}1.\ \ L0)$
else if $A\ i\ j = x$ **then** *ok*
else $(j:= j{+}1.\ \ L1)$

In C, this is

```
     i = 0;
L0:  if (i==n) j = m;
     else {     j = 0;
           L1:  if (j==m) {i = i+1;  goto L0;}
                else if (A[i][j]==x) ;
                else {j = j+1;  goto L1;}
           }
```

To add recursive time, we put $t:= t{+}1$ just after $i:= i{+}1$ and after $j:= j{+}1$. Or, to be a little more clever, we can get away with a single time increment placed just before the test $j{=}m$. We also change the five specifications we are refining to refer to time. The time remaining is at most the area remaining to be searched.

$t' \le t + n{\times}m \Leftarrow i:= 0.\ \ i{\le}n \Rightarrow t' \le t + (n{-}i){\times}m$

$i{\le}n \Rightarrow t' \le t + (n{-}i){\times}m \Leftarrow$ **if** $i{=}n$ **then** $j:= m$ **else** $i{<}n \Rightarrow t' \le t + (n{-}i){\times}m$

$i{<}n \Rightarrow t' \le t + (n{-}i){\times}m \Leftarrow j:= 0.\ \ i{<}n \wedge j{\le}m \Rightarrow t' \le t + (n{-}i){\times}m - j$

$i{<}n \wedge j{\le}m \Rightarrow t' \le t + (n{-}i){\times}m - j \Leftarrow$
$t:= t{+}1.$
if $j{=}m$ **then** $(i:= i{+}1.\ \ i{\le}n \Rightarrow t' \le t + (n{-}i){\times}m)$
else $i{<}n \wedge j{<}m \Rightarrow t' \le t + (n{-}i){\times}m - j$

$i<n \wedge j<m \Rightarrow t' \le t + (n-i)\times m - j \Leftarrow$
  if $A\ i\ j = x$ **then** *ok*
  else $(j{:=}\ j+1.\ \ i<n \wedge j\le m \Rightarrow t' \le t + (n-i)\times m - j)$

End of Two-Dimensional Search

5.2.3 For Loop

Let us use the syntax

for $i{:=}\ m;..n$ **do** P

where i is a fresh name, m and n are integer expressions such that $m\le n$, and P is a specification, as an almost-typical notation for controlled iteration. The difference from popular languages is just that iteration continues up to but excluding $i=n$. To avoid some thorns, let us say also that i is not a state variable (so it cannot be assigned within P), and that the initial values of m and n control the iteration (so the number of iterations is $n-m$).

As with the previous loop constructs, we will not define the **for**-loop as a specification, but instead show how it is used in refinement. Let F be a function of two integer variables whose result is an implementable specification. Then

$Fmn \Leftarrow$ **for** $i{:=}\ m;..n$ **do** P

is an abbreviation of the three refinements

$Fii \Leftarrow m\le i\le n \wedge ok$
$Fi(i+1) \Leftarrow m\le i<n \wedge P$
$Fik \Leftarrow m\le i<j<k\le n \wedge (Fij.\ Fjk)$

If $m=n$ there are no iterations, and specification Fmn must be satisfied by doing nothing ok. The body of the loop has to do one iteration $Fi(i+1)$. Finally, Fmn must be satisfied by first doing the iterations from m to an intermediate index j, and then doing the rest of the iterations from j to n.

For example, let the state consist of integer variable x, and let F be defined as

$F = \langle i, j{:}\ nat \rightarrow x' = x\times 2^{j-i}\rangle$

Then we can solve the exponentiation problem $x'=2^n$ in two refinements:

$x'=2^n \Leftarrow x{:=}\ 1.\ \ F0n$
$F0n \Leftarrow$ **for** $i{:=}\ 0;..n$ **do** $x{:=}\ 2\times x$

The first refinement is proven by the Substitution Law. To prove the second, we must prove three theorems

$Fii \Leftarrow 0\le i\le n \wedge ok$
$Fi(i+1) \Leftarrow 0\le i<n \wedge (x{:=}\ 2\times x)$
$Fik \Leftarrow 0\le i<j<k\le n \wedge (Fij.\ Fjk)$

all of which are easy.

The recursive time measure requires each loop to contain a time increment of at least one time unit. In general, the time taken by the body of a **for** loop may be a function f of the iteration i. Using $t' = t + \Sigma i{:}\ m,..n\cdot fi$ as **for**-loop specification Fmn, the **for**-loop rule tells us

$t' = t + \Sigma i{:}\ m,..n\cdot fi \Leftarrow$ **for** $i{:=}\ m;..n$ **do** $t' = t+fi$

When the body takes constant time c, this simplifies to

$t' = t + (n-m)\times c \Leftarrow$ **for** $i{:=}\ m;..n$ **do** $t' = t+c$

A typical use of the **for**-loop rule is to do something to each item in a list. For example, Exercise 268 asks us to add 1 to each item in a list. The specification is

$$\#L'=\#L \;\wedge\; \forall i\colon 0,..\#L\cdot L'i=Li+1$$

Now we need a specification Fik that describes an arbitrary segment of iterations: adding 1 to each item from index i to index k .

$$Fik \;=\; \#L'=\#L \wedge (\forall j\colon i,..k\cdot L'j=Lj+1) \wedge (\forall j\colon (0,..i), (k,..\#L)\cdot L'j=Lj)$$

To prove

$$F\,0\,(\#L) \;\Leftarrow\; \textbf{for } i{:=}\;0;..\#L \textbf{ do } L{:=}\; i{\to}Li+1 \mid L$$

we must prove three theorems:

$$Fii \;\Leftarrow\; 0{\le}i{\le}\#L \wedge ok$$
$$Fi(i+1) \;\Leftarrow\; 0{\le}i{<}\#L \wedge (L{:=}\; i{\to}Li+1 \mid L)$$
$$Fik \;\Leftarrow\; 0{\le}i{<}j{<}k{\le}\#L \wedge (Fij.\; Fjk)$$

Sometimes the **for**-loop specification Fmn has the form $Im{\Rightarrow}I'n$, where I is a function of one variable whose result is a precondition, and I' is the function whose result is the corresponding postcondition. When I is applied to the **for**-loop index, condition Ii is called an invariant. An advantage of this form of specification is that both $Fii \Leftarrow ok$ and $Fik \Leftarrow (Fij.\; Fjk)$ are automatically satisfied. Not all **for**-loop specifications can be put in this form; neither the timing nor the previous example (add 1 to each item) can be. But the earlier exponential example can be put in this form. Define

$$I \;=\; \langle i\colon nat{\to}x=2^i\rangle$$

Then the solution is

$$x'=2^n \;\Leftarrow\; x{:=}\;1.\; I0{\Rightarrow}I'n$$
$$I0{\Rightarrow}I'n \;\Leftarrow\; \textbf{for } i{:=}\;0;..n \textbf{ do } Ii{\Rightarrow}I'(i+1)$$
$$Ii{\Rightarrow}I'(i+1) \;\Leftarrow\; x{:=}\;2{\times}x$$

As another example of the invariant form of the **for**-loop rule, here is Exercise 186(a): Given a list of integers, possibly including negatives, write a program to find the minimum sum of any segment (sublist of consecutive items). Let L be the list. Formally, the problem is P where

$$P \;=\; s' \;=\; MIN\; i, j\cdot \Sigma\, L\, [i;..j]$$

where $0 \le i \le j \le \#L$. The condition $I\,k$ will say that s is the minimum sum of any segment up to index k . For $k=0$ there is only one segment, the empty segment, and its sum is 0 . When $k=\#L$ all segments are included and we have the desired result. To go from $I\,k$ to $I\,(k+1)$ we have to consider those segments that end at index $k+1$. We could find the sum of each new segment, then take the minimum of those sums and of s to be the new value of s . But we can do better. Each segment ending at index $k+1$ is a one-item extension of a segment ending at index k with one exception: the empty segment ending at $k+1$.

```
               k    k+1
               ↓     ↓
[ 4 ; –2 ; –8 ; 7 ; 3 ; 0 ; –1 ]
```

If we know the minimum sum c of any segment ending at k , then $min\;(c + L\;k)\;0$ is the minimum sum of any segment ending at $k+1$. So we define, for $0 \le k \le \#L$,

$$I\,k \;=\; s \;=\; (MIN\; i\colon 0,..k+1\cdot MIN\; j\colon i,..k+1\cdot \Sigma\, L\,[i;..j])$$
$$\wedge\;\; c \;=\; (MIN\; i\colon 0,..k+1\cdot \Sigma\, L\,[i;..k])$$

Now the program is easy.

$$P \;\Leftarrow\; s{:=}\;0.\;\; c{:=}\;0.\;\; I\,0 \Rightarrow I'(\#L)$$
$$I\,0 \Rightarrow I'(\#L) \;\Leftarrow\; \textbf{for } k{:=}\;0;..\#L \textbf{ do } I\,k \Rightarrow I'(k+1)$$
$$I\,k \Rightarrow I'(k+1) \;\Leftarrow\; c{:=}\;min\;(c + L\,k)\;0.\;\; s{:=}\;min\;c\;s$$

End of For Loop

5.2.4 Go To

Programming texts often warn that the **go to** is harmful, and should be avoided, but it causes no more problem for proof than loop constructs. For example, suppose the fast exponentiation program $z'=x^y$ of Subsection 4.2.6 were written as follows (using colon for labeling).

A: $z:= 1$. **if** *even* y **then go to** C
  else B: ($z:= z\times x$. $y:= y-1$.
    C: **if** $y=0$ **then go to** E
     else D: ($x:= x\times x$. $y:= y/2$. **if** *even* y **then go to** D **else go to** B)).
E:

Straight from the program, what needs to be proven is the following:

$A \Leftarrow$ $z:= 1$. **if** *even* y **then** C **else** B
$B \Leftarrow$ $z:= z\times x$. $y:= y-1$. C
$C \Leftarrow$ **if** $y=0$ **then** E **else** D
$D \Leftarrow$ $x:= x\times x$. $y:= y/2$. **if** *even* y **then** D **else** B

for appropriate formalizations of the labels (specifically, $A = z'=x^y$, $B = odd\ y \Rightarrow z'=z\times x^y$, $C = even\ y \Rightarrow z'=z\times x^y$, $D = even\ y \wedge y>0 \Rightarrow z'=z\times x^y$, and $E = ok$). The difficulty with **go to** , as with loop constructs, is inventing the specifications.

End of Go To

End of Control Structures

5.3 Time and Space Dependence

Some programming languages provide a clock, or a delay, or other time-dependent features. Our examples have used the time variable as a ghost, or auxiliary variable, never affecting the course of a computation. It was used as part of the theory, to prove something about the execution time. Used for that purpose only, it did not need representation in a computer. But if there is a readable clock available as a time source during a computation, it can be used to affect the computation. The assignment $deadline:= t + 5$ is allowed, as is **if** $t \le deadline$ **then** ... **else** ... . But the assignment $t:= 5$ is not allowed. We can look at the clock, but not reset it arbitrarily; all clock changes must correspond to the passage of time (according to some measure). (A computer operator may need to set the clock sometimes, but that is not part of the theory of programming.)

We may occasionally want to specify the passage of time. For example, we may want the computation to "wait until time w ". Let us invent a notation for it, and define it formally as

wait until w $=$ $t:= max\ t\ w$

Because we are not allowed to reset the clock, $t:= max\ t\ w$ is not acceptable as a program until we refine it. Letting time be an extended integer and using recursive time,

wait until w $\Leftarrow$ **if** $t\ge w$ **then** *ok* **else** ($t:= t+1$. **wait until** w)

and we obtain a busy-wait loop. We can prove this refinement by cases. First,

$t\ge w \wedge ok$
$= \quad t\ge w \wedge (t:= t)$
$\Rightarrow \quad t:= max\ t\ w$

And second,

$t<w \wedge (t:= t+1.\ t:= max\ t\ w)$
  In the left conjunct, use t: *xint* . In the right conjunct, use the Substitution Law.
$= \quad t+1 \le w \wedge (t:= max\ (t+1)\ w)$
$= \quad t+1 \le w \wedge (t:= w)$
$= \quad t<w \wedge (t:= max\ t\ w)$
$\Rightarrow \quad t:= max\ t\ w$

In programs that depend upon time, we should use the real time measure, rather than the recursive time measure. We also need to be more careful where we place our time increments. And we need a slightly different definition of **wait until** w , but we leave that as Exercise 275(b).

Our space variable s , like the time variable t , has so far been used to prove things about space usage, not to affect the computation. But if a program has space usage information available to it, there is no harm in using that information. Like t , s can be read but not written arbitrarily. All changes to s must correspond to changes in space usage.

End of Time and Space Dependence

5.4 Assertions optional

5.4.0 Checking

As a safety check, some programming languages include the notation

assert b

where b is boolean, to mean something like "I believe b is true". If it comes at the beginning of a procedure or method, it may use the word **precondition** ; if at the end, it may use the word **postcondition** ; if it comes at the start or end of a loop, it may use the word **invariant** ; these are all the same construct. It is executed by checking that b is true; if it is, execution continues normally, but if not, an error message is printed and execution is suspended. The intention is that in a correct program, the asserted expressions will always be true, and so all assertions are redundant. All error checking requires redundancy, and assertions help us to find errors and prevent subsequent damage to the state variables. But it's not free; it costs execution time.

Assertions are defined as follows.

assert b $=$ **if** b **then** ok **else** (*print* "error". **wait until** ∞)

If b is true, **assert** b is the same as ok . If b is false, execution cannot proceed in finite time to any following actions. Assertions are an easy way to make programs more robust.

End of Checking

5.4.1 Backtracking

If P and Q are implementable specifications, so is $P \lor Q$. The disjunction can be implemented by choosing one of P or Q and satisfying it. Normally this choice is made as a refinement, either $P \lor Q \Leftarrow P$ or $P \lor Q \Leftarrow Q$. We could save this programming step by making disjunction a programming connective, perhaps using the notation **or** . For example,

$x := 0$ **or** $x := 1$

would be a program whose execution assigns either 0 or 1 to x . This would leave the choice of disjunct to the programming language implementer.

The next construct radically changes the way we program. We introduce the notation

ensure b

where b is boolean, to mean something like "make b be true". We define it as follows.

ensure b $=$ **if** b **then** ok **else** $b' \land ok$

$=$ $b' \land ok$

Like **assert** b , **ensure** b is equal to ok if b is true. But when b is false, there is a problem: the computation must make b true without changing anything. This is unimplementable (unless b is identically true). However, in combination with other constructs, the whole may be implementable. Consider the following example in variables x and y .

$x:= 0$ **or** $x:= 1$. **ensure** $x=1$
$= \quad \exists x'', y''\cdot (x''=0 \land y''=y \ \lor\ x''=1 \land y''=y) \ \land\ x'=1 \land x'=x'' \land y'=y''$
$= \quad x'=1 \land y'=y$
$= \quad x:= 1$

Although an implementation is given a choice between $x:= 0$ and $x:= 1$, it must choose the right one to satisfy a later condition. It can do so by making either choice (as usual), and when faced with a later **ensure** whose condition is false, it must backtrack and make another choice. Since choices can be nested within choices, a lot of bookkeeping is necessary.

Several popular programming languages, such as Prolog, feature backtracking. They may state that choices are made in a particular order (we have omitted that complication). Two warnings should accompany such languages. First, it is the programmer's responsibility to show that a program is implementable; the language does not guarantee it. Alternatively, the implementation does not guarantee that computations will satisfy the program, since it is sometimes impossible to satisfy it. The second warning is that the time and space calculations do not work.

End of Backtracking

End of Assertions

5.5 Subprograms

5.5.0 Result Expression

Let P be a specification and e be an expression in unprimed variables. Then

P **result** e

is an expression of the initial state. It expresses the result of executing P and then evaluating e. For example, the following expresses an approximation to the base of the natural logarithms.

var *term*, *sum*: *rat* := 1·
for i:= 1;..15 **do** (*term*:= *term*/*i*. *sum*:= *sum*+*term*)
result *sum*

The axiom for the **result** expression is

$x' = (P$ **result** $e) \;=\; P.\ x'=e$

where x is any state variable of the right type.

The example introduces local variables *term* and *sum*, and no other variables are reassigned. So clearly the nonlocal state is unchanged. But consider

$y:= y+1$ **result** y

The result is as if the assignment $y:= y+1$ were executed, then y is the result, except that the value of variable y is unchanged.

$x:= (y:= y+1$ **result** $y)$
$= \quad x' = (y:= y+1$ **result** $y) \ \land\ y'=y$
$= \quad (y:= y+1.\ x'=y) \ \land\ y'=y$
$= \quad x' = y+1 \ \land\ y'=y$
$= \quad x:= y+1$

The expression P **result** e can be implemented as follows. Replace each nonlocal variable within P and e that is assigned within P by a fresh local variable initialized to the value of the nonlocal variable. Then execute P and evaluate e. In the implementation of some programming languages, the introduction of fresh local variables for this purpose is not done, so the evaluation of an expression may cause a state change. State changes resulting from the evaluation of an expression are called “side-effects”. With side-effects, mathematical reasoning is not possible. For example,

we cannot say $x+x = 2\times x$, nor even $x=x$, since x might be $(y{:=}\ y+1\ \textbf{result}\ y)$, and each evaluation results in an integer that is 1 larger than the previous evaluation. Side effects are easily avoided; a programmer can introduce the necessary local variables if the language implementation fails to do so. Some programming languages forbid assignments to nonlocal variables within expressions, so the programmer is required to introduce the necessary local variables.

If a programming language allows side-effects, we have to get rid of them before using any theory. For example,

$x{:=}\ (P\ \textbf{result}\ e)$ becomes $(P.\ x{:=}\ e)$

after renaming local variables within P as necessary to avoid clashes with nonlocal variables, and allowing the scope of variables declared in P to extend through $x{:=}\ e$. For another example,

$x{:=}\ y + (P\ \textbf{result}\ e)$ becomes $(\textbf{var}\ z{:=}\ y\cdot\ P.\ x{:=}\ z+e)$

with similar provisos.

The recursive time measure that we have been using neglects the time for expression evaluation. This is reasonable in some applications for expressions consisting of a few operations implemented in computer hardware. For expressions using operations not implemented in hardware (perhaps list catenation) it is questionable. For **result** expressions containing loops, it is unreasonable. But allowing a **result** expression to increase a time variable would be a side-effect, so here is what we do. We first include time in the **result** expression for the purpose of calculating a time bound. Then we remove the time variable from the result expression (to get rid of the side-effect) and we put a time increment in the program that uses the **result** expression.

End of Result Expression

5.5.1 Function

In many popular programming languages, a function is a combination of assertion about the result, name of the function, parameters, scope control, and **result** expression. It's a “package deal”. For example, in C, the binary exponential function looks like this:

```
int bexp (int n)
{ int r = 1;
  int i;
  for (i=0; i<n; i++) r = r*2;
  return r; }
```

In our notations, this would be

$$bexp \;=\; \langle\, n{:}\ int\rightarrow\ \textbf{var}\ r{:}\ int := 1\cdot\ \textbf{for}\ i{:=}\ 0;..n\ \textbf{do}\ r{:=}\ r\times 2.\ \textbf{assert}\ r{:}\ int\ \textbf{result}\ r\,\rangle$$

We present these programming features separately so that they can be understood separately. They can be combined in any way desired, as in the example. The harm in providing one construct for the combination is its complexity. Programmers trained with these languages may be unable to separate the issues and realize that naming, parameterization, assertions, local scope, and **result** expressions are independently useful.

Even the form of function we are using in this book could be both simplified and generalized. Stating the domain of a parameter is a special case of axiom introduction, which can be separated from name introduction (see Exercise 90).

End of Function

5.5.2 Procedure

The procedure (or void function, or method), as it is found in many languages, is a “package deal” like the function. It combines name declaration, parameterization, and local scope. The comments of the previous subsection apply here too. There are also some new issues.

To use our theory for program development, not just verification, we must be able to talk about a procedure whose body is an unrefined specification, not yet a program. For example, we may want a procedure P with parameter x defined as

$$P \;=\; \langle x{:}\ int \rightarrow a' < x < b' \rangle$$

that assigns variables a and b values that lie on opposite sides of a value to be supplied as argument. We can use procedure P before we refine its body. For example,

$$P\,3 \;=\; a' < 3 < b'$$
$$P\,(a+1) \;=\; a' < a+1 < b'$$

The body is easily refined as

$$a' < x < b' \;\Leftarrow\; a{:}=x-1.\ \ b{:}=x+1$$

Our choice of refinement does not alter our definition of P ; it is of no use when using P . The users don't need to know the implementation, and the implementer doesn't need to know the uses.

A procedure and argument can be translated to a local variable and initial value.

$$\langle p{:}\ D\rightarrow B\rangle\, a \;=\; (\mathbf{var}\ p{:}\ D := a\cdot\ B) \qquad \text{if } B \text{ doesn't use } p' \text{ or } p{:}=$$

This translation suggests that a parameter is really just a local variable whose initial value will be supplied as an argument. In many popular programming languages, that is exactly the case. This is an unfortunate confusion of specification and implementation. The decision to create a parameter, and the choice of its domain, are part of a procedural specification, and are of interest to a user of the procedure. The decision to create a local variable, and the choice of its domain, are normally part of refinement, part of the process of implementation, and should not be of concern to a user of the procedure. When a parameter is assigned a value within a procedure body, it is acting as a local variable and no longer has any connection to its former role as parameter.

Another kind of parameter, usually called a reference parameter or **var** parameter, stands for a nonlocal variable to be supplied as argument. Here is an example, using $\langle *\ \rangle$ to introduce a reference parameter.

$$\langle *x{:}\ int\rightarrow a{:}=3.\ \ b{:}=4.\ \ x{:}=5\rangle\, a$$
$$=\quad a{:}=3.\ \ b{:}=4.\ \ a{:}=5$$
$$=\quad a'=5 \wedge b'=4$$

Reference parameters can be used only when the body of the procedure is pure program, not using any other specification notations. For the above example, if we had written

$$\langle *x{:}\ int\rightarrow a'=3 \wedge b'=4 \wedge x'=5\rangle\, a$$

we could not just replace x with a , nor even x' with a' . Furthermore, we cannot do any reasoning about the procedure body until after the procedure has been applied to its arguments. The following example has a procedure body that is equivalent to the previous example,

$$\langle *x{:}\ int\rightarrow x{:}=5.\ \ b{:}=4.\ \ a{:}=3\rangle\, a$$
$$=\quad a{:}=5.\ \ b{:}=4.\ \ a{:}=3$$
$$=\quad a'=3 \wedge b'=4$$

but the result is different. Reference parameters prevent the use of specification, and they prevent any reasoning about the procedure by itself. We must apply our programming theory separately for each call. This contradicts the purpose of procedures.

End of Procedure

End of Subprograms

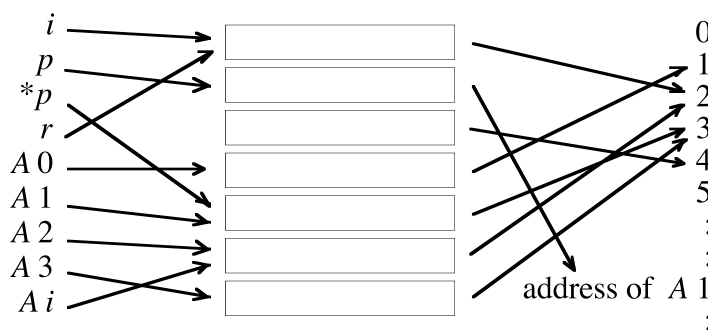
5.6 Alias optional

Many popular programming languages present us with a model of computation in which there is a memory consisting of a large number of individual storage cells. Each cell contains a value. Via the programming language, cells have names. Here is a standard sort of picture.

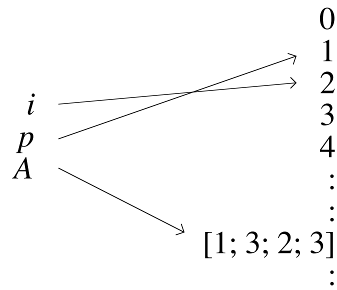
name	value
r, i	2
p	address of $A\ 1$
	4
$A\ 0$	1
$*p$, $A\ 1$	3
$A\ i$, $A\ 2$	2
$A\ 3$	3

In the picture, p is a pointer variable that currently points to array element $A\ 1$, and $*p$ is p dereferenced; so $*p$ and $A\ 1$ refer to the same memory cell. Since variable i currently has value 2 , $A\ i$ and $A\ 2$ refer to the same cell. And r is a reference parameter for which variable i has been supplied as argument, so r and i refer to the same cell. We see that a cell may have zero, one, two, or more names. When a cell has two or more names that are visible at the same time, the names are said to be "aliases".

As we have seen with arrays and with reference parameters, aliasing prevents us from applying our theory of programming. Some programming languages prohibit aliasing. Unfortunately, aliasing is difficult to detect, especially during program construction before a specification has been fully refined as a program. To most people, prohibitions and restrictions are distasteful. To avoid the prohibition, we have a choice: we can complicate our theory of programming to handle aliasing, or we can simplify our model of computation to eliminate it. If we redraw our picture slightly, we see that there are two mappings: one from names to cells, and one from cells to values.



An assignment such as p:= address of $A\ 3$ or i:= 4 can change both mappings at once. An assignment to one name can change the value indirectly referred to by another name. To simplify the picture and eliminate the possibility of aliasing, we eliminate the cells and allow a richer space of values. Here is the new picture.



Pointer variables can be replaced by index variables dedicated to one structure so that they can be implemented as addresses. Reference parameters are unnecessary if functions can return structured values. The simpler picture is perfectly adequate, and the problem of aliasing disappears.

End of Alias

5.7 Probabilistic Programming optional

Probability Theory has been developed using the arbitrary convention that a probability is a real number between 0 and 1 inclusive

$prob = §r: real· 0≤r≤1$

with 1 representing "certainly true", 0 representing "certainly false", 1/2 representing "equally likely true or false", and so on. Accordingly, for this section only, we add the axioms

$\top = 1$

$\bot = 0$

With these axioms, boolean operators can be expressed arithmetically. For example, $\neg x = 1-x$, $x \wedge y = x \times y$, and $x \vee y = x - x \times y + y$.

A distribution is an expression whose value (for all assignments of values to its variables) is a probability, and whose sum (over all assignments of values to its variables) is 1 . (For the sake of simplicity, we consider only distributions over boolean and integer variables; for rational and real variables, summations become integrals.) For example, if $n: nat+1$, then 2^{-n} is a distribution because

$(\forall n: nat+1· 2^{-n}: prob) \wedge (\Sigma n: nat+1· 2^{-n})=1$

If we have two variables $n, m: nat+1$, then 2^{-n-m} is a distribution because

$(\forall n, m: nat+1· 2^{-n-m}: prob) \wedge (\Sigma n, m: nat+1· 2^{-n-m})=1$

A distribution can be used to tell the frequency of occurrence of values of its variables. For example, 2^{-n} says that n has value 3 one-eighth of the time. Distribution 2^{-n-m} says that the state in which n has value 3 and m has value 1 occurs one-sixteenth of the time. A distribution can also be used to say what values we expect or predict variables to have. Distribution 2^{-n} says that n is equally as likely to have the value 1 as it is to not have the value 1 , and twice as likely to have the value 1 as it is to have the value 2 . A distribution can also be used to specify the probability that we want for the value of each variable.

Suppose we have one natural state variable n . The specification $n' = n+1$ tells us that, for any given initial value n , the final value n' is $n+1$. Stated differently, it says the final value n' equals the initial value $n+1$ with probability 1 , and equals any other value with probability 0 . For any values of n and n' , the value of $n' = n+1$ is either $\top$ (1) or $\bot$ (0), so it is a probability. The specification $n' = n+1$ is not a distribution of n and n' because there are infinitely many pairs of values that give $n' = n+1$ the value $\top$ or 1 , and so

$\Sigma n, n'· n' = n+1 = \infty$

But for any fixed value of n , there is a single value of n' that gives $n' = n+1$ the value $\top$ or 1 ,

and so

$\Sigma n' \cdot\ n' = n+1 \quad = \quad 1$

For any fixed value of n , $n' = n+1$ is a one-point distribution of n' . Similarly, any implementable deterministic specification is a one-point distribution of the final state.

We generalize our programming notations to allow probabilistic operands as follows.

ok	$=$	$(x'=x) \times (y'=y) \times ...$
$x{:=}\ e$	$=$	$(x'=e) \times (y'=y) \times ...$
if b **then** P **else** Q	$=$	$b \times P\ +\ (1-b) \times Q$
$P.\,Q$	$=$	$\Sigma x'', y'', ...\cdot$ (for $x', y', ...$ substitute $x'', y'', ...$ in P) $\times$ (for $x, y, ...$ substitute $x'', y'', ...$ in Q)

Since $\bot=0$ and $\top=1$, the definitions of ok and assignment have not changed; they have just been expressed arithmetically. If b , P , and Q are boolean, the definitions of **if** b **then** P **else** Q and $P.Q$ have not changed. But now they apply not only to $\bot$ and $\top$, that is to 0 and 1 , but also to values between 0 and 1 . In other words, they apply to probabilities. If b is a probability of the initial state, and P and Q are distributions of the final state, then **if** b **then** P **else** Q is a distribution of the final state. If P and Q are distributions of the final state, then $P.Q$ is a distribution of the final state. For example,

if $1/3$ **then** $x{:=}\ 0$ **else** $x{:=}\ 1$

means that with probability $1/3$ we assign x the value 0 , and with the remaining probability $2/3$ we assign x the value 1 . In one variable x ,

if $1/3$ **then** $x{:=}\ 0$ **else** $x{:=}\ 1$

$= \quad 1/3 \times (x'=0) + (1 - 1/3) \times (x'=1)$

Let us evaluate this expression using the value 0 for x' .

$$\begin{aligned} & 1/3 \times (0=0) + (1 - 1/3) \times (0=1) \\ = \quad & 1/3 \times 1 + 2/3 \times 0 \\ = \quad & 1/3 \end{aligned}$$

which is the probability that x has final value 0 . Let us evaluate this expression using the value 1 for x' .

$$\begin{aligned} & 1/3 \times (1=0) + (1 - 1/3) \times (1=1) \\ = \quad & 1/3 \times 0 + 2/3 \times 1 \\ = \quad & 2/3 \end{aligned}$$

which is the probability that x has final value 1 . Let us evaluate this expression using the value 2 for x' .

$$\begin{aligned} & 1/3 \times (2=0) + (1 - 1/3) \times (2=1) \\ = \quad & 1/3 \times 0 + 2/3 \times 0 \\ = \quad & 0 \end{aligned}$$

which is the probability that x has final value 2 .

Here is a slightly more elaborate example in one variable x .

if $1/3$ **then** $x{:=}\ 0$ **else** $x{:=}\ 1$.
if $x{=}0$ **then if** $1/2$ **then** $x{:=}\ x{+}2$ **else** $x{:=}\ x{+}3$
else if $1/4$ **then** $x{:=}\ x{+}4$ **else** $x{:=}\ x{+}5$

$$\begin{aligned} = \quad & \Sigma x''\cdot\ ((x''=0)/3 + (x''=1)\times 2/3) \\ & \quad \times\ (\ (x''=0) \times ((x' = x''+2)/2 + (x' = x''+3)/2) \\ & \quad\quad +\ (1 - (x''=0)) \times ((x' = x''+4)/4 + (x' = x''+5)\times 3/4)) \\ = \quad & (x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2 \end{aligned}$$

After the first line, x might be 0 or 1 . If it is 0 , then with probability $1/2$ we add 2 , and with the remaining probability $1/2$ we add 3 ; otherwise (if x is not 0) with probability $1/4$ we add 4 and with the remaining probability $3/4$ we add 5 . The sum is much easier than it looks

because all values for x'' other than 0 and 1 make a 0 contribution to the sum. The final line says that the resulting value of variable x is 2 with probability $1/6$, 3 with probability $1/6$, 5 with probability $1/6$, 6 with probability $1/2$, and any other value with probability 0 .

Let P be any distribution of final states, and let e be any number expression over initial states. After execution of P , the average value of e is $(P.e)$. For example, the average value of n^2 as n varies over $nat+1$ according to distribution 2^{-n} is

$$
\begin{aligned}
& 2^{-n'}.\ n^2 \\
= \quad & \Sigma n''\!: nat+1\cdot\ 2^{-n''} \times n''^2 \\
= \quad & 6
\end{aligned}
$$

After execution of the previous example, the average value of x is

$$
\begin{aligned}
& \textbf{if } 1/3 \textbf{ then } x:= 0 \textbf{ else } x:= 1. \\
& \textbf{if } x{=}0 \textbf{ then if } 1/2 \textbf{ then } x:= x+2 \textbf{ else } x:= x+3 \\
& \textbf{else if } 1/4 \textbf{ then } x:= x+4 \textbf{ else } x:= x+5. \\
& x \\
= \quad & (x'{=}2)/6 + (x'{=}3)/6 + (x'{=}5)/6 + (x'{=}6)/2.\ x \\
= \quad & \Sigma x''\cdot\ ((x''{=}2)/6 + (x''{=}3)/6 + (x''{=}5)/6 + (x''{=}6)/2) \times x'' \\
= \quad & 1/6 \times 2 + 1/6 \times 3 + 1/6 \times 5 + 1/2 \times 6 \\
= \quad & 4 + 2/3
\end{aligned}
$$

Let P be any distribution of final states, and let b be any boolean expression over initial states. After execution of P , the probability that b is true is $(P.b)$. Probability is just the average value of a boolean expression. For example, after execution of the previous example, the probability that x is greater than 3 is

$$
\begin{aligned}
& \textbf{if } 1/3 \textbf{ then } x:= 0 \textbf{ else } x:= 1. \\
& \textbf{if } x{=}0 \textbf{ then if } 1/2 \textbf{ then } x:= x+2 \textbf{ else } x:= x+3 \\
& \textbf{else if } 1/4 \textbf{ then } x:= x+4 \textbf{ else } x:= x+5. \\
& x{>}3 \\
= \quad & (x'{=}2)/6 + (x'{=}3)/6 + (x'{=}5)/6 + (x'{=}6)/2.\ x{>}3 \\
= \quad & \Sigma x''\cdot\ ((x''{=}2)/6 + (x''{=}3)/6 + (x''{=}5)/6 + (x''{=}6)/2) \times (x''{>}3) \\
= \quad & 1/6 \times (2{>}3) + 1/6 \times (3{>}3) + 1/6 \times (5{>}3) + 1/2 \times (6{>}3) \\
= \quad & 2/3
\end{aligned}
$$

Most of the laws, including all distribution laws and the Substitution Law, apply without change to probabilistic specifications and programs. For example, the previous two calculations could begin by distributing the final line (x in the first one, $x>3$ in the second) back into the **then**- and **else**-parts that increase x , then distribute **if** $x=0$ ... back into the **then**- and **else**-parts that initialize x , then use the Substitution Law six times, thus avoiding the need to sum.

5.7.0 Random Number Generators

Many programming languages provide a random number generator (sometimes called a “pseudo-random number generator”). The usual notation is functional, and the usual result is a value whose distribution is uniform (constant) over a nonempty finite range. If $n\colon nat+1$, we use the notation $rand\ n$ for a generator that produces natural numbers uniformly distributed over the range $0,..n$. So $rand\ n$ has value r with probability $(r\colon 0,..n) / n$.

Functional notation for a random number generator is inconsistent. Since $x=x$ is a law, we should be able to simplify $rand\ n = rand\ n$ to $\top$, but we cannot because the two occurrences of $rand\ n$ might generate different numbers. Since $x+x = 2\times x$ is a law, we should be able to simplify $rand\ n + rand\ n$ to $2 \times rand\ n$, but we cannot. To restore consistency, we replace each use of $rand$ with a fresh variable before we do anything else. We can replace $rand\ n$ with integer variable r whose value has probability $(r\text{: }0,..n) / n$. Or, if you prefer, we can replace $rand\ n$ with variable $r\text{: }0,..n$ whose value has probability $1/n$. (This is a mathematical variable, or in other words, a state constant; there is no r'.) For example, in one state variable x,

$$
\begin{array}{lll}
 & x{:=}\ rand\ 2.\ \ x{:=}\ x + rand\ 3 & \text{replace one } rand \text{ with } r \text{ and one with } s\\
= & \Sigma r\text{: }0,..2\cdot\ \Sigma s\text{: }0,..3\cdot\ (x{:=}\ r)/2.\ \ (x{:=}\ x + s)/3 & \text{factor twice}\\
= & (\Sigma r\text{: }0,..2\cdot\ \Sigma s\text{: }0,..3\cdot\ (x{:=}\ r.\ \ x{:=}\ x + s))/6 & \text{replace final assignment, Substitution Law}\\
= & (\Sigma r\text{: }0,..2\cdot\ \Sigma s\text{: }0,..3\cdot\ (x' = r{+}s))\ /\ 6 & \text{sum}\\
= & ((x' = 0{+}0) + (x' = 0{+}1) + (x' = 0{+}2) + (x' = 1{+}0) + (x' = 1{+}1) + (x' = 1{+}2))\ /\ 6 & \\
= & (x'{=}0)\ /\ 6\ +\ (x'{=}1)\ /\ 3 + (x'{=}2)\ /\ 3\ +\ (x'{=}3)\ /\ 6 &
\end{array}
$$

which says that x' is 0 with probability $1/6$, 1 with probability $1/3$, 2 with probability $1/3$, 3 with probability $1/6$, and any other value with probability 0.

Whenever $rand$ occurs in the context of a simple equation, such as $r = rand\ n$, we don't need to introduce a variable for it, since one is supplied. We just replace the deceptive equation with $(r\text{: }0,..n) / n$. For example, in one variable x,

$$
\begin{array}{lll}
 & x{:=}\ rand\ 2.\ \ x{:=}\ x + rand\ 3 & \text{replace assignments}\\
= & (x'\text{: }0,..2)/2.\ \ (x'\text{: }x{+}(0,..3))/3 & \text{dependent composition}\\
= & \Sigma x''\cdot\ (x''\text{: }0,..2)/2 \times (x'\text{: }x''{+}(0,..3))/3 & \text{sum}\\
= & 1/2 \times (x'\text{: }0,..3)/3\ +\ 1/2 \times (x'\text{: }1,..4)/3 & \\
= & (x'{=}0)\ /\ 6\ +\ (x'{=}1)\ /\ 3 + (x'{=}2)\ /\ 3\ +\ (x'{=}3)\ /\ 6 &
\end{array}
$$

as before. And **if** $rand\ 2$ **then** A **else** B can be replaced by **if** $1/2$ **then** A **else** B.

Although $rand$ produces uniformly distributed natural numbers, it can be transformed into many different distributions. We just saw that $rand\ 2 + rand\ 3$ has value n with distribution $((n{=}0) + (n{=}3))\ /\ 6\ +\ ((n{=}1) + (n{=}2))\ /\ 3$. As another example, $rand\ 8 < 3$ has boolean value b with distribution

$$
\begin{array}{ll}
 & \Sigma r\text{: }0,..8\cdot\ (b = (r{<}3))\ /\ 8\\
= & (b{=}\top) \times 3/8\ +\ (b{=}\bot) \times 5/8\\
= & 5/8 - b/4
\end{array}
$$

which says that b is $\top$ with probability $3/8$, and $\bot$ with probability $5/8$.

Exercise 281 is a simplified version of blackjack. You are dealt a card from a deck; its value is in the range 1 through 13 inclusive. You may stop with just one card, or have a second card if you want. Your object is to get a total as near as possible to 14, but not over 14. Your strategy is to take a second card if the first is under 7. Assuming each card value has equal probability (actually, the second card drawn has a diminished probability of having the same value as the first card drawn, but let's ignore that complication), we represent a card as $(rand\ 13) + 1$. In one variable x, the game is

$$
\begin{array}{lll}
 & x{:=}\ (rand\ 13) + 1.\ \ \textbf{if}\ x{<}7\ \textbf{then}\ x{:=}\ x + (rand\ 13) + 1\ \textbf{else}\ ok & \\
 & & \text{replace } rand \text{ and } ok\\
= & (x'\text{: }(0,..13){+}1)/13.\ \ \textbf{if}\ x{<}7\ \textbf{then}\ (x'\text{: }x{+}(0,..13){+}1)/13\ \textbf{else}\ x'{=}x & \text{replace . and } \textbf{if}\\
= & \Sigma x''\cdot\ (x''\text{: }1,..14)/13 \times ((x''{<}7){\times}(x'\text{: }x''{+}1,..x''{+}14)/13 + (x''{\ge}7){\times}(x'{=}x'')) & \\
 & & \text{by several omitted steps}\\
= & ((2{\le}x'{<}7){\times}(x'{-}1) + (7{\le}x'{<}14){\times}19 + (14{\le}x'{<}20){\times}(20{-}x'))\ /\ 169 &
\end{array}
$$

That is the distribution of x' if we use the "under 7" strategy. We can similarly find the distribution of x' if we use the "under 8" strategy, or any other strategy. But which strategy is best? To compare two strategies, we play both of them at once. Player x will play "under n" and player y will play "under $n{+}1$" using exactly the same cards c and d (the result would be no different if they used different cards, but it would require more variables). Here is the new game, followed by the condition that x wins:

$$c:= (\mathit{rand}\ 13) + 1.\ \ d:= (\mathit{rand}\ 13) + 1.$$
$$\textbf{if } c < n \textbf{ then } x:= c{+}d \textbf{ else } x:= c.\ \ \textbf{if } c < n{+}1 \textbf{ then } y:= c{+}d \textbf{ else } y:= c.$$
$$y{<}x{\le}14 \vee x{\le}14{<}y \qquad \text{Replace } \mathit{rand} \text{ and use the Functional-Imperative Law twice.}$$
$$= \quad (c'{:}\ (0,..13){+}1 \ \wedge\ d'{:}\ (0,..13){+}1 \ \wedge\ x'{=}x \ \wedge\ y'{=}y) / 13 / 13.$$
$$x:= \textbf{if } c < n \textbf{ then } c{+}d \textbf{ else } c.\ \ y:= \textbf{if } c < n{+}1 \textbf{ then } c{+}d \textbf{ else } c.$$
$$y{<}x{\le}14 \vee x{\le}14{<}y \qquad \text{Use the Substitution Law twice.}$$
$$= \quad (c'{:}\ (0,..13){+}1 \ \wedge\ d'{:}\ (0,..13){+}1 \ \wedge\ x'{=}x \ \wedge\ y'{=}y) / 169.$$
$$(\textbf{if } c < n{+}1 \textbf{ then } c{+}d \textbf{ else } c) < (\textbf{if } c < n \textbf{ then } c{+}d \textbf{ else } c) \le 14$$
$$\vee\ \ (\textbf{if } c < n \textbf{ then } c{+}d \textbf{ else } c) \le 14 < (\textbf{if } c < n{+}1 \textbf{ then } c{+}d \textbf{ else } c)$$
$$= \quad (c'{:}\ (0,..13){+}1 \ \wedge\ d'{:}\ (0,..13){+}1 \ \wedge\ x'{=}x \ \wedge\ y'{=}y) / 169.\ \ c{=}n \wedge d{>}14{-}n$$
$$= \quad \Sigma c'', d'', x'', y''\cdot$$
$$(c''{:}\ (0,..13){+}1 \ \wedge\ d''{:}\ (0,..13){+}1 \ \wedge\ x''{=}x \ \wedge\ y''{=}y) / 169 \times (c''{=}n \wedge d''{>}14{-}n)$$
$$= \quad \Sigma d''{:}\ 1,..14\cdot (d''{>}14{-}n)/169$$
$$= \quad (n{-}1) / 169$$

The probability that x wins is $(n{-}1) / 169$. By similar calculations we can find that the probability that y wins is $(14{-}n) / 169$, and the probability of a tie is $12/13$. For $n{<}8$, "under $n{+}1$" beats "under n". For $n{\ge}8$, "under n" beats "under $n{+}1$". So "under 8" beats both "under 7" and "under 9".

Exercise 282 asks: If you repeatedly throw a pair of six-sided dice until they are equal, how long does it take? The program is

$$u'{=}v' \ \Longleftarrow\ u:= (\mathit{rand}\ 6) + 1.\ \ v:= (\mathit{rand}\ 6) + 1.\ \ \textbf{if } u{=}v \textbf{ then } \mathit{ok} \textbf{ else } (t:= t{+}1.\ \ u'{=}v')$$

Each iteration, with probability $5/6$ we keep going, and with probability $1/6$ we stop. So we offer the hypothesis that (for finite t) the execution time has the distribution

$$(t'{\ge}t) \times (5/6)^{t'-t} \times 1/6$$

To prove it, let's start with the implementation.

$$u:= (\mathit{rand}\ 6) + 1.\ \ v:= (\mathit{rand}\ 6) + 1. \qquad \text{replace } \mathit{rand} \text{ and}$$
$$\textbf{if } u{=}v \textbf{ then } t'{=}t \textbf{ else } (t:= t{+}1.\ \ (t'{\ge}t) \times (5/6)^{t'-t} \times 1/6) \qquad \text{Substitution Law}$$
$$= \quad (u'{:}\ 1,..7 \ \wedge\ v'{=}v \ \wedge\ t'{=}t)/6.\ \ (u'{=}u \ \wedge\ v'{:}\ 1,..7 \ \wedge\ t'{=}t)/6. \qquad \text{replace first .}$$
$$\textbf{if } u{=}v \textbf{ then } t'{=}t \textbf{ else } (t'{\ge}t{+}1) \times (5/6)^{t'-t-1} / 6 \qquad \text{and simplify}$$
$$= \quad (u', v'{:}\ 1,..7 \ \wedge\ t'{=}t)/36. \qquad \text{replace remaining .}$$
$$\textbf{if } u{=}v \textbf{ then } t'{=}t \textbf{ else } (t'{\ge}t{+}1) \times (5/6)^{t'-t-1} / 6 \qquad \text{and replace } \textbf{if}$$
$$= \quad \Sigma u'', v''{:}\ 1,..7\cdot\ \Sigma t''\cdot\ (t''{=}t)/36 \times \ (\ \ (u''{=}v'') \times (t'{=}t'')$$
$$+ (u''{\ne}v'') \times (t'{\ge}t''{+}1) \times (5/6)^{t'-t''-1} / 6) \qquad \text{sum}$$
$$= \quad (6 \times (t'{=}t)\ +\ 30 \times (t'{\ge}t{+}1) \times (5/6)^{t'-t-1} / 6) / 36 \qquad \text{combine}$$
$$= \quad (t'{\ge}t) \times (5/6)^{t'-t} \times 1/6$$

which is the distribution we hypothesized, and that completes the proof.

The average value of t' is

$$(t'{\ge}t) \times (5/6)^{t'-t} \times 1/6.\ \ t \ = \ t{+}5$$

so on average it takes 5 additional throws of the dice (after the first) to get an equal pair.

End of Random Number Generators

Probability problems are notorious for misleading even professional mathematicians. Informal reasoning to arrive at a probability distribution, as is standard in studies of probability, is essential to forming a reasonable hypothesis. But hypotheses are sometimes wrong. We write the hypothesis as a probabilistic specification, we refine it as a program, and we prove our refinements exactly as we did with boolean specifications. Sometimes wrong hypotheses can be traced to a wrong understanding of the problem. Formalization as a program makes one's understanding clear. Proof shows that a hypothesized probability distribution is correct for the program. Informal arguments are replaced by formal proof.

Probabilistic specifications can also be interpreted as “fuzzy” specifications. For example, $(x'=0)/3 + (x'=1)\times 2/3$ could mean that we will be one-third satisfied if the result x' is 0, two-thirds satisfied if it is 1, and completely unsatisfied if it is anything else.

5.7.1 Information optional

There is a close connection between information and probability. If a boolean expression has probability p of being true, and you evaluate it, and it turns out to be true, then the amount of information in bits that you have just learned is $info\ p$, defined as

$$info\ p \ = \ -\ log\ p$$

where log is the binary (base 2) logarithm. For example, $even\ (rand\ 8)$ has probability $1/2$ of being true. If we evaluate it and find that it is true, we have just learned

$$info\ (1/2) \ = \ -\ log\ (1/2) \ = \ log\ 2 \ = \ 1$$

bit of information; we have learned that the rightmost bit of the random number we were given is 0. If we find that $even\ (rand\ 8)$ is false, then we have learned that $\neg\ even\ (rand\ 8)$ is true, and since it also has probability $1/2$, we have also gained one bit; we have learned that the rightmost bit of the random number is 1. If we test $rand\ 8 = 5$, which has probability $1/8$ of being true, and we find that it is true, we learn

$$info\ (1/8) \ = \ -\ log\ (1/8) \ = \ log\ 8 \ = \ 3$$

bits, which is the entire random number in binary. If we find that $rand\ 8 = 5$ is false, we learn

$$info\ (7/8) \ = \ -\ log\ (7/8) \ = \ log\ 8 - log\ 7 \ = \ 3 - 2.80736 \ = \ 0.19264 \text{ approximately}$$

bits; we learn that the random number isn't 5, but it could be any of 7 others. Suppose we test $rand\ 8 < 8$. Since it is certain to be true, there is really no point in making this test; we learn

$$info\ 1 \ = \ -\ log\ 1 \ = \ -0 \ = \ 0$$

When an **if** b **then** P **else** Q occurs within a loop, b is tested repeatedly. Suppose b has probability p of being true. When it is true, we learn $info\ p$ bits, and this happens with probability p. When it is false, we learn $info\ (1-p)$ bits, and this happens with probability $(1-p)$. The average amount of information gained, called the entropy, is

$$entro\ p \ = \ p \times info\ p \ + \ (1-p) \times info\ (1-p)$$

For example, $entro\ (1/2) \ = \ 1$, and $entro\ (1/8) \ = \ entro\ (7/8) \ = \ 0.54356$ approximately. Since $entro\ p$ is at its maximum when $p=1/2$, we learn most on average, and make the most efficient use of the test, if its probability is near $1/2$. For example, in the binary search problem of Chapter 4, we could have divided the remaining search interval anywhere, but for the best average execution time, we split it into two parts having equal probabilities of finding the item we seek. And in the fast exponentiation problem, it is better on average to test $even\ y$ rather than $y=0$ if we have a choice.

End of Information

End of Probabilistic Programming

5.8 Functional Programming optional

Most of this book is about a kind of programming that is sometimes called "imperative", which means that a program describes a change of state (or "commands" a computer to change state in a particular way). This section presents an alternative: a program is a function from its input to its output. More generally, a specification is a function from possible inputs to desired outputs, and programs (as always) are implemented specifications. We take away assignment and dependent composition from our programming notations, and we add functions.

To illustrate, we look once again at the list summation problem (Exercise 142). This time, the specification is $\langle L\text{: } [*rat] \to \Sigma L \rangle$. Assuming Σ is not an implemented operator, we still have some programming to do. We introduce variable n to indicate how much of the list has been summed; initially n is 0 .

$$\Sigma L \;=\; \langle n\text{: } 0,..\#L+1 \to \Sigma\, L\, [n;..\#L] \rangle\; 0$$

It saves some copying to write " $\Sigma L =$ ... " rather than " $\langle L\text{: } [*rat] \to \Sigma L \rangle =$ ... ", but we must remember the domain of L . At first sight, the domain of n is annoying; it seems to be one occasion when an interval notation that includes both endpoints would be preferable. On second look, it's trying to tell us something useful: the domain is really composed of two parts that must be treated differently.

$$0,..\#L+1 \;=\; (0,..\#L), \#L$$

We divide the function into a selective union

$$\langle n\text{: } 0,..\#L+1 \to \Sigma\, L\, [n;..\#L] \rangle \;=\; \langle n\text{: } 0,..\#L \to \Sigma\, L\, [n;..\#L] \rangle \mid \langle n\text{: } \#L \to \Sigma\, L\, [n;..\#L] \rangle$$

and continue with each part separately. In the left part, we have $n<\#L$, and in the right part $n=\#L$.

$$\langle n\text{: } 0,..\#L \to \Sigma\, L\, [n;..\#L] \rangle \;=\; \langle n\text{: } 0,..\#L \to Ln + \Sigma\, L\, [n+1;..\#L] \rangle$$

$$\langle n\text{: } \#L \to \Sigma\, L\, [n;..\#L] \rangle \;=\; \langle n\text{: } \#L \to 0 \rangle$$

This time we copied the domain of n to indicate which part of the selective union is being considered. The one remaining problem is solved by recursion.

$$\Sigma\, L\, [n+1;..\#L] \;=\; \langle n\text{: } 0,..\#L+1 \to \Sigma\, L\, [n;..\#L] \rangle\; (n+1)$$

In place of the selective union we could have used **if then else** ; they are related by the law

$$\langle v\text{: } A \to x \rangle \mid \langle v\text{: } B \to y \rangle \;=\; \langle v\text{: } A, B \to \textbf{if } v\text{: } A \textbf{ then } x \textbf{ else } y \rangle$$

When we are interested in the execution time rather than the result, we replace the result of each function with its time according to some measure. For example, in the list summation problem, we might decide to charge time 1 for each addition and 0 for everything else. The specification becomes $\langle L\text{: } [*rat] \to \#L \rangle$, meaning for any list, the execution time is its length. We now must make exactly the same programming steps as before. The first step was to introduce variable n ; we do the same now, but we choose a new result for the new function to indicate its execution time.

$$\#L \;=\; \langle n\text{: } 0,..\#L+1 \to \#L-n \rangle\; 0$$

The second step was to decompose the function into a selective union; we do so again.

$$\langle n\text{: } 0,..\#L+1 \to \#L-n \rangle \;=\; \langle n\text{: } 0,..\#L \to \#L-n \rangle \mid \langle n\text{: } \#L \to \#L-n \rangle$$

The left side of the selective union became a function with one addition in it, so our timing function must become a function with a charge of 1 in it. To make the equation correct, the time for the remaining summation must be adjusted.

$$\langle n\text{: } 0,..\#L \to \#L-n \rangle \;=\; \langle n\text{: } 0,..\#L \to 1 + \#L-n-1 \rangle$$

The right side of the selective union became a function with a constant result; according to our measure, its time must be 0 .

$$\langle n\text{: } \#L \to \#L-n \rangle \;=\; \langle n\text{: } \#L \to 0 \rangle$$

The remaining problem was solved by a recursive call; the corresponding call solves the remaining time problem.

$\#L-n-1 \quad = \quad \langle n\colon 0,..\#L+1 \rightarrow \#L-n\rangle\ (n+1)$

And that completes the proof that execution time (according to this measure) is the length of the list.

In the recursive time measure, we charge nothing for any operation except recursive call, and we charge 1 for that. Let's redo the timing proof with this measure. Again, the time specification is $\langle L\colon [*rat] \rightarrow \#L\rangle$.

$$\#L \ = \ \langle n\colon 0,..\#L+1 \rightarrow \#L-n\rangle\ 0$$
$$\langle n\colon 0,..\#L+1 \rightarrow \#L-n\rangle \ = \ \langle n\colon 0,..\#L \rightarrow \#L-n\rangle \mid \langle n\colon \#L \rightarrow \#L-n\rangle$$
$$\langle n\colon 0,..\#L \rightarrow \#L-n\rangle \ = \ \langle n\colon 0,..\#L \rightarrow \#L-n\rangle$$
$$\langle n\colon \#L \rightarrow \#L-n\rangle \ = \ \langle n\colon \#L \rightarrow 0\rangle$$
$$\#L-n \ = \ 1 + \langle n\colon 0,..\#L+1 \rightarrow \#L-n\rangle\ (n+1)$$

5.8.0 Function Refinement

In imperative programming, we can write a nondeterministic specification such as $x'\colon 2, 3, 4$ that allows the result to be any one of several possibilities. In functional programming, a nondeterministic specification is a bunch consisting of more than one element. The specification $2, 3, 4$ allows the result to be any one of those three numbers.

Functional specifications can be classified the same way as imperative specifications, based on the number of satisfactory outputs for each input.

Functional specification S is unsatisfiable for domain element x :	$¢\, Sx < 1$
Functional specification S is satisfiable for domain element x :	$¢\, Sx \ge 1$
Functional specification S is deterministic for domain element x :	$¢\, Sx \le 1$
Functional specification S is nondeterministic for domain element x :	$¢\, Sx > 1$

Functional specification S is satisfiable for domain element x :	$\exists y\cdot\ y\colon Sx$
Functional specification S is implementable:	$\forall x\cdot\ \exists y\cdot\ y\colon Sx$

(x is quantified over the domain of S , and y is quantified over the range of S .) Implementability can be restated as $\forall x\cdot\ Sx \ne null$.

Consider the problem of searching for an item in a list of integers. Our first attempt at specification might be

$$\langle L\colon [*int] \rightarrow \langle x\colon int \rightarrow \S n\colon 0,..\#L\cdot\ Ln = x\rangle\rangle$$

which says that for any list L and item x , we want an index of L where x occurs. If x occurs several times in L , any of its indexes will do. Unfortunately, if x does not occur in L , we are left without any possible result, so this specification is unimplementable. We must decide what we want when x does not occur in L ; let's say any natural that is not an index of L will do.

$$\langle L\colon [*int] \rightarrow \langle x\colon int \rightarrow \mathbf{if}\ x\colon L\ (0,..\#L)\ \mathbf{then}\ \S n\colon 0,..\#L\cdot\ Ln = x\ \mathbf{else}\ \#L,..\infty\rangle\rangle$$

This specification is implementable, and often nondeterministic.

Functional refinement is similar to imperative refinement. An imperative specification is a boolean expression, and imperative refinement is reverse implication. Functional specification is a function, and functional refinement is the reverse of the function ordering. Functional specification P (the problem) is refined by functional specification S (the solution) if and only if $S\colon P$. To refine, we can either decrease the choice of result, or increase the domain. Now we have a most annoying notational problem. Typically, we like to write the problem on the left, then the refinement symbol, then the solution on the right; we want to write $S\colon P$ the other way round. Inclusion is antisymmetric, so its symbol should not be symmetric, but unfortunately it is. Let us write $::$ for “backwards colon”, so that “ P is refined by S ” is written $P\colon\colon S$.

To refine our search specification, we create a linear search program, starting the search with index 0 and increasing the index until either x is found or L is exhausted. First we introduce the index.

$$(\textbf{if } x: L\ (0,..\#L)\ \textbf{then } \S n: 0,..\#L\cdot\ Ln = x\ \textbf{else } \#L,..\infty)::$$
$$\langle i: nat\rightarrow \textbf{if } x: L\ (i,..\#L)\ \textbf{then } \S n: i,..\#L\cdot\ Ln = x\ \textbf{else } \#L,..\infty\rangle\ 0$$

The two sides of this refinement are equal, so we could have written $=$ instead of $::$. We could have been more precise about the domain of i , and then we probably would decompose the function into a selective union, as we did in the previous problem. But this time let's use an **if then else** .

$$(\textbf{if } x: L\ (i,..\#L)\ \textbf{then } \S n: i,..\#L\cdot\ Ln = x\ \textbf{else } \#L,..\infty)::$$
$$\textbf{if } i = \#L\ \textbf{then } \#L$$
$$\textbf{else if } x = Li\ \textbf{then } i$$
$$\textbf{else } \langle i: nat\rightarrow \textbf{if } x: L\ (i,..\#L)\ \textbf{then } \S n: i,..\#L\cdot\ Ln = x\ \textbf{else } \#L,..\infty\rangle\ (i+1)$$

The timing specification, recursive measure, is $\langle L\rightarrow\langle x\rightarrow 0,..\#L+1\rangle\rangle$, which means that the time is less than $\#L+1$. To prove that this is the execution time, we must prove

$$0,..\#L+1::\ \ \langle i: nat\rightarrow 0,..\#L-i+1\rangle\ 0$$

and

$$0,..\#L-i+1::\quad \textbf{if } i = \#L\ \textbf{then } 0$$
$$\textbf{else if } x = Li\ \textbf{then } 0$$
$$\textbf{else } 1 + \langle i: nat\rightarrow 0,..\#L-i+1\rangle\ (i+1)$$

As this example illustrates, the steps in a functional refinement are the same as the steps in an imperative refinement for the same problem, including the resolution of nondeterminism and timing. But the notations are different.

End of Function Refinement

Functional and imperative programming are not really competitors; they can be used together. We cannot ignore imperative programming if ever we want to pause, to stop computing for a while, and resume later from the same state. Imperative programming languages all include a functional (expression) sublanguage, so we cannot ignore functional programming either.

At the heart of functional programming we have the Application Axiom

$$\langle v: D\rightarrow b\rangle\ x\ \ =\ \ (\text{for } v \text{ substitute } x \text{ in } b\ )$$

At the heart of imperative programming we have the Substitution Law

$$x:= e.\ P\ \ =\ \ (\text{for } x \text{ substitute } e \text{ in } P\ )$$

Functional programming and imperative programming differ mainly in the notation they use for substitution.

End of Functional Programming

End of Programming Language

6 Recursive Definition

6.0 Recursive Data Definition

In this section we are concerned with the definition of infinite bunches. Our first example is *nat* , the natural numbers. It was defined in Chapter 2 using axioms called construction and induction. Now we take a closer look at these axioms.

6.0.0 Construction and Induction

To define *nat* , we need to say what its elements are. We can start by saying that 0 is an element

$0: nat$

and then say that for every element of *nat* , adding 1 gives an element

$nat+1: nat$

These axioms are called the *nat* construction axioms, and 0 and $nat+1$ are called the *nat* constructors. Using these axioms, we can "construct" the elements of *nat* as follows.

	$\top$	by the axiom, $0: nat$
$\Rightarrow$	$0: nat$	add 1 to each side
$\Rightarrow$	$0+1: nat+1$	by arithmetic, $0+1 = 1$; by the axiom, $nat+1: nat$
$\Rightarrow$	$1: nat$	add 1 to each side
$\Rightarrow$	$1+1: nat+1$	by arithmetic, $1+1 = 2$; by the axiom, $nat+1: nat$
$\Rightarrow$	$2: nat$	

and so on.

From the construction axioms we can prove $2: nat$ but we cannot prove $\neg\ -2: nat$. That is why we need the induction axiom. The construction axioms tell us that the natural numbers are in *nat* , and the induction axiom tells us that nothing else is. Here is the *nat* induction axiom.

$0: B\ \wedge\ B+1: B\ \Rightarrow\ nat: B$

We have introduced *nat* as a constant, like *null* and 0 . It is not a variable, and cannot be instantiated. But B is a variable, to be instantiated at will.

The two construction axioms can be combined into one, and induction can be restated, as follows:

$0, nat+1: nat$ — *nat* construction

$0, B+1: B\ \Rightarrow\ nat: B$ — *nat* induction

There are many bunches satisfying the inclusion $0, B+1: B$, such as: the naturals, the integers, the integers and half-integers, the rationals. Induction says that of all these bunches, *nat* is the smallest.

We have presented *nat* construction and *nat* induction using bunch notation. We now present equivalent axioms using predicate notation. We begin with induction.

In predicate notation, the *nat* induction axiom can be stated as follows: If $P: nat\rightarrow bool$,

$P0 \wedge \forall n: nat\cdot Pn \Rightarrow P(n+1)\ \ \Longrightarrow\ \ \forall n: nat\cdot Pn$

We prove first that the bunch form implies the predicate form.

	$0: B\ \wedge\ B+1: B\ \Rightarrow\ nat: B$	Let $B = \S n: nat\cdot Pn$. Then $B: nat$,
$\Longrightarrow$	$0: B\ \wedge (\forall n: nat\cdot n: B\ \Rightarrow\ n+1: B)\ \Rightarrow\ \forall n: nat\cdot n: B$	and $\forall n: nat\cdot (n: B)=Pn$.
$=$	$P0 \wedge (\forall n: nat\cdot Pn \Rightarrow P(n+1))\ \Rightarrow\ \forall n: nat\cdot Pn$	

The reverse is proven similarly.

$$P0 \wedge (\forall n: nat \cdot Pn \Rightarrow P(n+1)) \;\Rightarrow\; \forall n: nat \cdot Pn$$

For arbitrary bunch B , let $P = \langle n: nat \rightarrow n: B\rangle$. Then again $\forall n: nat \cdot Pn = (n: B)$.

$$\Rightarrow\; 0: B \;\wedge\; (\forall n: nat \cdot n: B \;\Rightarrow\; n+1: B) \;\Rightarrow\; \forall n: nat \cdot n: B$$
$$=\; 0: B \;\wedge\; (\forall n: nat`B \cdot n+1: B) \;\Rightarrow\; \forall n: nat \cdot n: B$$
$$=\; 0: B \;\wedge\; (nat`B)+1: B \;\Rightarrow\; nat: B$$
$$\Rightarrow\; 0: B \;\wedge\; B+1: B \;\Rightarrow\; nat: B$$

Therefore the bunch and predicate forms of *nat* induction are equivalent.

The predicate form of *nat* construction can be stated as follows: If $P: nat \rightarrow bool$,

$$P0 \wedge \forall n: nat \cdot Pn \Rightarrow P(n+1) \;\Leftarrow\; \forall n: nat \cdot Pn$$

This is the same as induction but with the main implication reversed. We prove first that the bunch form implies the predicate form.

$\forall n: nat \cdot Pn$ — domain change using *nat* construction, bunch version

$\Rightarrow\; \forall n: 0, nat+1 \cdot Pn$ — axiom about $\forall$

$=\; (\forall n: 0 \cdot Pn) \wedge (\forall n: nat+1 \cdot Pn)$ — One-Point Law and variable change

$=\; P0 \wedge \forall n: nat \cdot P(n+1)$

$\Rightarrow\; P0 \wedge \forall n: nat \cdot Pn \Rightarrow P(n+1)$

And now we prove that the predicate form implies the bunch form .

$P0 \wedge (\forall n: nat \cdot Pn \Rightarrow P(n+1)) \;\Leftarrow\; \forall n: nat \cdot Pn$ — Let $P = \langle n: nat \rightarrow n: nat\rangle$

$\Rightarrow\; 0: nat \wedge (\forall n: nat \cdot n: nat \;\Rightarrow\; n+1: nat) \;\Leftarrow\; \forall n: nat \cdot n: nat$

$=\; 0: nat \wedge (\forall n: nat \cdot n+1: nat) \;\Leftarrow\; \top$

$=\; 0: nat \;\wedge\; nat+1: nat$

A corollary is that *nat* can be defined by the single axiom

$$P0 \wedge \forall n: nat \cdot Pn \Rightarrow P(n+1) \;=\; \forall n: nat \cdot Pn$$

There are other predicate versions of induction; here is the usual one again plus three more.

$$P0 \wedge \forall n: nat \cdot Pn \Rightarrow P(n+1) \;\Rightarrow\; \forall n: nat \cdot Pn$$
$$P0 \vee \exists n: nat \cdot \neg Pn \wedge P(n+1) \;\Leftarrow\; \exists n: nat \cdot Pn$$
$$\forall n: nat \cdot Pn \Rightarrow P(n+1) \;\Rightarrow\; \forall n: nat \cdot P0 \Rightarrow Pn$$
$$\exists n: nat \cdot \neg Pn \wedge P(n+1) \;\Leftarrow\; \exists n: nat \cdot \neg P0 \wedge Pn$$

The first version says that to prove P of all naturals, prove it of 0 , and assuming it of natural n , prove it of $n+1$. In other words, you get to all naturals by starting at 0 and repeatedly adding 1 . The second version is obtained from the first by the duality laws and a renaming. The next is the prettiest; it says that if you can “go” from any natural to the next, then you can “go” from 0 to any natural.

Here are two laws that are consequences of induction.

$$\forall n: nat \cdot (\forall m: nat \cdot m<n \Rightarrow Pm) \Rightarrow Pn \;\Rightarrow\; \forall n: nat \cdot Pn$$
$$\exists n: nat \cdot (\forall m: nat \cdot m<n \Rightarrow \neg Pm) \wedge Pn \;\Leftarrow\; \exists n: nat \cdot Pn$$

The first is like the first version of induction, except that the base case $P0$ is not explicitly stated, and the step uses the assumption that all previous naturals satisfy P , rather than just the one previous natural. The last one says that if there is a natural with property P then there is a first natural with property P (all previous naturals don't have it).

Proof by induction does not require any special notation or format. For example, Exercise 288 asks us to prove that the square of an odd natural number is $8\times m + 1$ for some natural m. Quantifying over nat,

	$\forall n\cdot \exists m\cdot (2\times n + 1)^2 = 8\times m + 1$	various number laws
$=$	$\forall n\cdot \exists m\cdot 4\times n\times(n+1) + 1 = 8\times m + 1$	various number laws
$=$	$\forall n\cdot \exists m\cdot n\times(n+1) = 2\times m$	the usual predicate form of induction
$\Leftarrow$	$(\exists m\cdot 0\times(0+1) = 2\times m)$	generalization and
	$\wedge\ (\forall n\cdot (\exists m\cdot n\times(n+1) = 2\times m) \Rightarrow (\exists l\cdot (n+1)\times(n+2) = 2\times l))$	distribution
$\Leftarrow$	$0\times(0+1) = 2\times 0$	arithmetic and
	$\wedge\ (\forall n, m\cdot n\times(n+1) = 2\times m \Rightarrow (\exists l\cdot (n+1)\times(n+2) = 2\times l))$	generalization
$\Leftarrow$	$\forall n, m\cdot n\times(n+1) = 2\times m \Rightarrow (n+1)\times(n+2) = 2\times(m+n+1)$	various number laws
$=$	$\top$	

Now that we have an infinite bunch, it is easy to define others. For example, we can define pow to be the powers of 2 either by the equation

$$pow = 2^{nat}$$

or by using the solution quantifier

$$pow = \S p\colon nat\cdot \exists m\colon nat\cdot p = 2^m$$

But let us do it the same way we defined nat. The pow construction axiom is

$$1, 2\times pow\colon pow$$

and the pow induction axiom is

$$1, 2\times B\colon B \Rightarrow pow\colon B$$

Induction is not just for nat. In predicate form, we can define pow with the axiom

$$P1 \wedge \forall p\colon pow\cdot Pp \Rightarrow P(2\times p) = \forall p\colon pow\cdot Pp$$

We can define the bunch of integers as

$$int = nat, -nat$$

or equivalently we can use the construction and induction axioms

$$0, int+1, int-1\colon int$$

$$0, B+1, B-1\colon B \Rightarrow int\colon B$$

or we can use the axiom

$$P0 \wedge (\forall i\colon int\cdot Pi \Rightarrow P(i+1)) \wedge (\forall i\colon int\cdot Pi \Rightarrow P(i-1)) = \forall i\colon int\cdot Pi$$

Whichever we choose as axiom(s), the others are theorems.

Similarly we can define the bunch of rationals as

$$rat = int/(nat+1)$$

or equivalently by the construction and induction axioms

$$1, rat+rat, rat-rat, rat\times rat, rat/(\S r\colon rat\cdot r\neq 0)\colon rat$$

$$1, B+B, B-B, B\times B, B/(\S b\colon B\cdot b\neq 0)\colon B \Rightarrow rat\colon B$$

or with the axiom (quantifying over rat, of course)

$$\begin{aligned} & P1 \\ & \wedge (\forall r, s\cdot Pr \wedge Ps \Rightarrow P(r+s)) \\ & \wedge (\forall r, s\cdot Pr \wedge Ps \Rightarrow P(r-s)) \\ & \wedge (\forall r, s\cdot Pr \wedge Ps \Rightarrow P(r\times s)) \\ & \wedge (\forall r, s\cdot Pr \wedge Ps \wedge s\neq 0 \Rightarrow P(r/s)) \\ = \ & \forall r\cdot Pr \end{aligned}$$

As the examples suggest, we can define a bunch by construction and induction axioms using any number of constructors. To end this subsection, we define a bunch using zero constructors. In general, we have one construction axiom per constructor, so there aren't any construction axioms. But there is still an induction axiom. With no constructors, the antecedent becomes trivial and disappears, and we are left with the induction axiom

$null: B$

where $null$ is the bunch being defined. As always, induction says that, apart from elements due to construction axioms, nothing else is in the bunch being defined. This is exactly how we defined $null$ in Chapter 2. The predicate form of $null$ induction is

$\forall x: null\cdot\ P\ x$

End of Construction and Induction

6.0.1 Least Fixed-Points

We have defined nat by a construction axiom and an induction axiom

$0, nat+1: nat$ — nat construction

$0, B+1: B \Rightarrow nat: B$ — nat induction

We now prove two similar-looking theorems:

$nat = 0, nat+1$ — nat fixed-point construction

$B = 0, B+1 \Rightarrow nat: B$ — nat fixed-point induction

A fixed-point of a function f is an element x of its domain such that f maps x to itself: $x = fx$. A least fixed-point of f is a smallest such x . Fixed-point construction has the form

name = (expression involving *name*)

and so it says that *name* is a fixed-point of the expression on the right. Fixed-point induction tells us that *name* is the smallest bunch satisfying fixed-point construction, and in that sense it is the least fixed-point of the constructor.

We first prove nat fixed-point construction. It is stronger than nat construction, so the proof will also have to use nat induction. Let us start there.

$\top$ — nat induction axiom

$=\ 0, B+1: B \Rightarrow nat: B$ — replace B with $0, nat+1$

$\Rightarrow\ 0, (0, nat+1)+1: 0, nat+1 \Rightarrow nat: 0, nat+1$ — strengthen the antecedent by cancelling the "0"s and "+1"s from the two sides of the first ":"

$\Rightarrow\ 0, nat+1: nat \Rightarrow nat: 0, nat+1$ — the antecedent is the nat construction axiom, so we can delete it, and use it again to strengthen the consequent

$=\ nat = 0, nat+1$

We prove nat fixed-point induction just by strengthening the antecedent of nat induction.

In similar fashion we can prove that pow , int , and rat are all least fixed-points of their constructors. In fact, we could have defined nat and each of these bunches as least fixed-points of their constructors. It is quite common to define a bunch of strings by a fixed-point construction axiom called a grammar. For example,

exp = "x", *exp*; "+"; *exp*

In this context, union is usually denoted by | and catenation is usually denoted by nothing. The other axiom, to say that *exp* is the least of the fixed-points, is usually stated informally by saying that only constructed elements are included.

End of Least Fixed-Points

6.0.2 Recursive Data Construction

Recursive construction is a procedure for constructing least fixed-points from constructors. It usually works, but not always. We seek the smallest solution of

$name$ = (expression involving $name$)

Here are the steps of the procedure.

0. Construct a sequence of bunches $name_0$ $name_1$ $name_2$ ... beginning with

$name_0 = null$

and continuing with

$name_{n+1}$ = (expression involving $name_n$)

We can thus construct a bunch $name_n$ for any natural number n .

1. Next, try to find an expression for $name_n$ that may involve n but does not involve $name$.

$name_n$ = (expression involving n but not $name$)

2. Now form a bunch $name_\infty$ by replacing n with ∞ .

$name_\infty$ = (expression involving neither n nor $name$)

3. The bunch $name_\infty$ is usually the least fixed-point of the constructor, but not always, so we must test it. First we test to see if it is a fixed-point.

$name_\infty$ = (expression involving $name_\infty$)

4. Then we test $name_\infty$ to see if it is the least fixed-point.

B = (expression involving B) $\Longrightarrow$ $name_\infty$: B

We illustrate recursive construction on the constructor for pow , which is $1, 2\times pow$.

0. Construct the sequence

$$\begin{aligned}
pow_0 &= null \\
pow_1 &= 1, 2\times pow_0 \\
&= 1, 2\times null \\
&= 1, null \\
&= 1 \\
pow_2 &= 1, 2\times pow_1 \\
&= 1, 2\times 1 \\
&= 1, 2 \\
pow_3 &= 1, 2\times pow_2 \\
&= 1, 2\times(1, 2) \\
&= 1, 2, 4
\end{aligned}$$

The first bunch pow_0 tells us all the elements of the bunch pow that we know without looking at its constructor. In general, pow_n represents our knowledge of pow after n uses of its constructor.

1. Perhaps now we can guess the general member of this sequence

$$pow_n = 2^{0,..n}$$

We could prove this by *nat* induction, but it is not really necessary. The proof would only tell us about pow_n for n: *nat* and we want pow_∞. Besides, we will test our final result.

2. Now that we can express pow_n, we can define pow_∞ as

$$pow_\infty = 2^{0,..\infty} = 2^{nat}$$

and we have found a likely candidate for the least fixed-point of the *pow* constructor.

3. We must test pow_∞ to see if it is a fixed-point.

$$\begin{aligned}
& 2^{nat} = 1, 2{\times}2^{nat} \\
= \;& 2^{nat} = 2^0, 2^1{\times}2^{nat} \\
= \;& 2^{nat} = 2^0, 2^{1+nat} \\
= \;& 2^{nat} = 2^{0,\ 1+nat} \\
\Leftarrow \;& nat = 0, nat{+}1 && \text{nat fixed-point construction} \\
= \;& \top
\end{aligned}$$

4. We must test pow_∞ to see if it is the least fixed-point.

$$\begin{aligned}
& 2^{nat}: B \\
= \;& \forall n: nat\cdot 2^n: B && \text{use the predicate form of nat induction} \\
\Leftarrow \;& 2^0: B \ \wedge\ \forall n: nat\cdot 2^n: B \ \Rightarrow\ 2^{n+1}: B && \text{change variable} \\
= \;& 1: B \ \wedge\ \forall m: 2^{nat}\cdot m: B \ \Rightarrow\ 2{\times}m: B && \text{increase domain} \\
\Leftarrow \;& 1: B \ \wedge\ \forall m: nat\cdot m: B \ \Rightarrow\ 2{\times}m: B && \text{Domain Change Law} \\
= \;& 1: B \ \wedge\ \forall m: nat`B\cdot 2{\times}m: B && \text{increase domain} \\
\Leftarrow \;& 1: B \ \wedge\ \forall m: B\cdot 2{\times}m: B \\
= \;& 1: B \ \wedge\ 2{\times}B: B \\
\Leftarrow \;& B = 1, 2{\times}B
\end{aligned}$$

Since 2^{nat} is the least fixed-point of the *pow* constructor, we conclude $pow = 2^{nat}$.

In step 0, we start with $name_0 = null$, which is usually the best starting bunch for finding a smallest solution. But occasionally that starting bunch fails and some other starting bunch succeeds in producing a solution to the given fixed-point equation.

In step 2, from $name_n$ we obtain a candidate $name_\infty$ for a fixed-point of a constructor by replacing n with ∞. This substitution is simple to perform, and the resulting candidate is usually satisfactory. But the result is sensitive to the way $name_n$ is expressed. From two expressions for $name_n$ that are equal for all finite n, we may obtain expressions for $name_\infty$ that are unequal. Another candidate, one that is not sensitive to the way $name_n$ is expressed, is

$$\S x\cdot LIM\ n\cdot x: name_n$$

But this bunch is sensitive to the choice of domain of x (the domain of n has to be *nat*). Finding a limit is harder than making a substitution, and the result is still not guaranteed to produce a fixed-point. We could define a property, called "continuity", which, together with monotonicity, is sufficient to guarantee that the limit is the least fixed-point, but we leave the subject to other books.

End of Recursive Data Construction

Whenever we add axioms, we must be careful to remain consistent with the theory we already have. A badly chosen axiom can cause inconsistency. Here is an example. Suppose we make

$$bad \;=\; \S n\colon nat\cdot \neg\, n\colon bad$$

an axiom. Thus bad is defined as the bunch of all naturals that are not in bad . From this axiom we find

$$\begin{aligned} & 0\colon bad \\ = \;& 0\colon \S n\colon nat\cdot \neg\, n\colon bad \\ = \;& \neg\, 0\colon bad \end{aligned}$$

is a theorem. From the Completion Rule we find that $0\colon bad \;=\; \neg\, 0\colon bad$ is also an antitheorem. To avoid the inconsistency, we must withdraw this axiom.

Sometimes recursive construction does not produce any answer. For example, the fixed-point equation of the previous paragraph results in the sequence of bunches

$$\begin{aligned} bad_0 &= null \\ bad_1 &= nat \\ bad_2 &= null \end{aligned}$$

and so on, alternating between $null$ and nat . We cannot say what bad_∞ is because we cannot say whether ∞ is even or odd. Even the Limit Axiom tells us nothing. We should not blame recursive construction for failing to find a fixed-point when there is none. However, it sometimes fails to find a fixed-point when there is one (see Exercise 314).

End of Recursive Data Definition

6.1 Recursive Program Definition

Programs, and more generally, specifications, can be defined by axioms just as data can. For our first example, let x and y be integer variables. The name zap is introduced, and the fixed-point equation

$$zap \;=\; \textbf{if } x{=}0 \textbf{ then } y{:=}\,0 \textbf{ else } (x{:=}\,x{-}1.\;\; t{:=}\,t{+}1.\;\; zap)$$

is given as an axiom. The right side of the equation is the constructor. Here are six solutions to this equation.

(a) $x{\ge}0 \;\Rightarrow\; x'{=}y'{=}0 \land t' = t{+}x$

(b) $\textbf{if } x{\ge}0 \textbf{ then } x'{=}y'{=}0 \land t' = t{+}x \textbf{ else } t'{=}\infty$

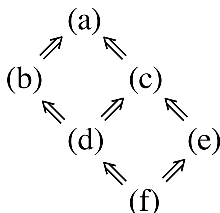
(c) $x'{=}y'{=}0 \land (x{\ge}0 \Rightarrow t' = t{+}x)$

(d) $x'{=}y'{=}0 \land \textbf{if } x{\ge}0 \textbf{ then } t' = t{+}x \textbf{ else } t'{=}\infty$

(e) $x'{=}y'{=}0 \land t' = t{+}x$

(f) $x{\ge}0 \land x'{=}y'{=}0 \land t' = t{+}x$

Solution (a) is the weakest and solution (f) is the strongest, although the solutions are not totally ordered. We can express their order by the following picture.



Solutions (e) and (f) are so strong that they are unimplementable. Solution (d) is implementable, and since it is also deterministic, it is a strongest implementable solution.

From the fixed-point equation defining zap , we cannot say that zap is equal to a particular one of the solutions. But we can say this: it refines the weakest solution

$$x\ge 0 \Rightarrow x'=y'=0 \land t' = t+x \quad \Leftarrow \quad zap$$

so we can use it to solve problems. And it is refined by its constructor

$$zap \quad \Leftarrow \quad \mathbf{if}\ x=0\ \mathbf{then}\ y:= 0\ \mathbf{else}\ (x:= x-1.\ \ t:= t+1.\ \ zap)$$

so we can execute it. For all practical purposes, that is all we need.

6.1.0 Recursive Program Construction

Recursive program construction is similar to recursive data construction, and serves a similar purpose. We illustrate the procedure using the example zap . We start with zap_0 describing the computation as well as we can without looking at the definition of zap . Of course, if we don't look at the definition, we have no idea what computation zap is describing, so let us start with a specification that is satisfied by every computation.

$$zap_0 = \top$$

We obtain the next description of zap by substituting zap_0 for zap in the constructor, and so on.

$$\begin{aligned} zap_1 &= \mathbf{if}\ x=0\ \mathbf{then}\ y:= 0\ \mathbf{else}\ (x:= x-1.\ \ t:= t+1.\ \ zap_0) \\ &= x=0 \ \Rightarrow\ x'=y'=0 \land t'=t \\ zap_2 &= \mathbf{if}\ x=0\ \mathbf{then}\ y:= 0\ \mathbf{else}\ (x:= x-1.\ \ t:= t+1.\ \ zap_1) \\ &= 0\le x<2 \ \Rightarrow\ x'=y'=0 \land t' = t+x \end{aligned}$$

In general, zap_n describes the computation as well as possible after n uses of the constructor. We can now guess (and prove using nat induction if we want)

$$zap_n = 0\le x<n \ \Rightarrow\ x'=y'=0 \land t' = t+x$$

The next step is to replace n with ∞ .

$$zap_\infty = 0\le x<\infty \ \Rightarrow\ x'=y'=0 \land t' = t+x$$

Finally, we must test the result to see if it satisfies the axiom.

$$\begin{aligned} & \text{(right side of equation with } zap_\infty \text{ for } zap\text{)} \\ =\ & \mathbf{if}\ x=0\ \mathbf{then}\ y:= 0\ \mathbf{else}\ (x:= x-1.\ \ t:= t+1.\ \ 0\le x \ \Rightarrow\ x'=y'=0 \land t' = t+x) \\ =\ & \mathbf{if}\ x=0\ \mathbf{then}\ x'=y'=0 \land t'=t\ \mathbf{else}\ 0\le x-1 \ \Rightarrow\ x'=y'=0 \land t' = t+x \\ =\ & 0\le x \ \Rightarrow\ x'=y'=0 \land t' = t+x \\ =\ & \text{(left side of equation with } zap_\infty \text{ for } zap\text{)} \end{aligned}$$

It satisfies the fixed-point equation, and in fact it is the weakest fixed-point.

If we are not considering time, then $\top$ is all we can say about an unknown computation, and we start our recursive construction there. With time, we can say more than just $\top$; we can say that time does not decrease. Starting with $t' \ge t$ we can construct a stronger fixed-point.

$$\begin{aligned} zap_0 &= t' \ge t \\ zap_1 &= \mathbf{if}\ x=0\ \mathbf{then}\ y:= 0\ \mathbf{else}\ (x:= x-1.\ \ t:= t+1.\ \ zap_0) \\ &= \mathbf{if}\ x=0\ \mathbf{then}\ x'=y'=0 \land t'=t\ \mathbf{else}\ t' \ge t+1 \\ zap_2 &= \mathbf{if}\ x=0\ \mathbf{then}\ y:= 0\ \mathbf{else}\ (x:= x-1.\ \ t:= t+1.\ \ zap_1) \\ &= \mathbf{if}\ x=0\ \mathbf{then}\ x'=y'=0 \land t'=t\ \mathbf{else}\ \mathbf{if}\ x=1\ \mathbf{then}\ x'=y'=0 \land t'=t+1\ \mathbf{else}\ t' \ge t+2 \\ &= \mathbf{if}\ 0\le x<2\ \mathbf{then}\ x'=y'=0 \land t' = t+x\ \mathbf{else}\ t' \ge t+2 \end{aligned}$$

In general, zap_n describes what we know up to time n . We can now guess (and prove using nat induction if we want)

$$zap_n = \mathbf{if}\ 0\le x<n\ \mathbf{then}\ x'=y'=0 \land t'=t+x\ \mathbf{else}\ t' \ge t+n$$

We replace n with ∞

$$zap_\infty \;=\; \textbf{if } 0\le x \textbf{ then } x'=y'=0 \land t'=t+x \textbf{ else } t'=\infty$$

and test the result

$$\begin{aligned}
& \text{(right side of equation with } zap_\infty \text{ for } zap\text{)}\\
=\;& \textbf{if } x=0 \textbf{ then } y{:=}\ 0 \textbf{ else } (x{:=}\ x-1.\ \ t{:=}\ t+1.\ \ \textbf{if } 0\le x \textbf{ then } x'=y'=0 \land t'=t+x \textbf{ else } t'=\infty)\\
=\;& \textbf{if } x=0 \textbf{ then } x'=y'=0 \land t'=t \textbf{ else if } 0\le x-1 \textbf{ then } x'=y'=0 \land t'=t+x \textbf{ else } t'=\infty\\
=\;& \textbf{if } 0\le x \textbf{ then } x'=y'=0 \land t' = t+x \textbf{ else } t'=\infty\\
=\;& \text{(left side of equation with } zap_\infty \text{ for } zap\text{)}
\end{aligned}$$

Beginning our recursive construction with $t' \ge t$, we have constructed a stronger but still implementable fixed-point. In this example, if we begin our recursive construction with $\bot$ we obtain the strongest fixed-point, which is unimplementable.

We obtained a candidate zap_∞ for a fixed-point by replacing n with ∞. An alternative candidate is $LIM\ n\cdot zap_n$. In this example, the two candidates are equal, but in other examples the two ways of forming a candidate may give different results.

End of Recursive Program Construction

6.1.1 Loop Definition

Loops can be defined by construction and induction. The axioms for the **while**-loop are

$$\begin{aligned}
& t'\ge t \;\Leftarrow\; \textbf{while } b \textbf{ do } P\\
& \textbf{if } b \textbf{ then } (P.\ \ t{:=}\ t+1.\ \ \textbf{while } b \textbf{ do } P) \textbf{ else } ok \;\Leftarrow\; \textbf{while } b \textbf{ do } P\\
& \qquad \forall\sigma, \sigma'\cdot (t'\ge t \land (\textbf{if } b \textbf{ then } (P.\ \ t{:=}\ t+1.\ \ W) \textbf{ else } ok) \;\Leftarrow\; W)\\
& \Rightarrow\; \forall\sigma, \sigma'\cdot (\textbf{while } b \textbf{ do } P \;\Leftarrow\; W)
\end{aligned}$$

Recursive timing has been included, but this can be changed to any other timing policy. These three axioms are closely analogous to the axioms

$$\begin{aligned}
& 0\text{: } nat\\
& nat+1\text{: } nat\\
& 0, B+1\text{: } B \;\Rightarrow\; nat\text{: } B
\end{aligned}$$

that define nat. The first **while**-loop axiom is a base case saying that at least time does not decrease. The second construction axiom takes a single step, saying that **while** b **do** P refines (implements) its first unrolling; then by Stepwise Refinement we can prove it refines any of its unrollings. The last axiom, induction, says that it is the weakest specification that satisfies the first two axioms.

From these axioms we can prove theorems called fixed-point construction and fixed-point induction. For the **while**-loop they are

$$\begin{aligned}
& \textbf{while } b \textbf{ do } P \;=\; t'\ge t \land (\textbf{if } b \textbf{ then } (P.\ \ t{:=}\ t+1.\ \ \textbf{while } b \textbf{ do } P) \textbf{ else } ok)\\
& \qquad \forall\sigma, \sigma'\cdot (W \;=\; t'\ge t \land (\textbf{if } b \textbf{ then } (P.\ \ t{:=}\ t+1.\ \ W) \textbf{ else } ok))\\
& \Rightarrow\; \forall\sigma, \sigma'\cdot (\textbf{while } b \textbf{ do } P \;\Leftarrow\; W)
\end{aligned}$$

This account differs from that presented in Chapter 5; we have gained some theorems, and also lost some theorems. For example, from this definition, we cannot prove

$$x'\ge x \;\Leftarrow\; \textbf{while } b \textbf{ do } x'\ge x$$

which was easily proved according to Chapter 5.

End of Loop Definition

End of Recursive Program Definition

End of Recursive Definition

7 Theory Design and Implementation

Programmers use the formalisms, abstractions, theories, and structures that have been created for them by the designers and implementers of their programming languages. With every program they write, with every name they introduce, programmers create new formalisms, abstractions, theories, and structures. To make their creations as elegant and useful as possible, programmers should be aware of their role as theory designers and implementers, as well as theory users.

The stack, the queue, and the tree are standard data structures used frequently in programming. It is not the purpose of the present chapter to show their usefulness in applications; we leave that to books devoted to data structures. They are presented here as case studies in theory design and implementation. Each of these data structures contains items of some sort. For example, we can have stacks of integers, stacks of lists of booleans, even stacks of stacks. In this chapter, X is the bunch (or type) of items in a data structure.

7.0 Data Theories

7.0.0 Data-Stack Theory

The stack is a useful data structure for the implementation of programming languages. Its distinguishing feature is that, at any time, the item to be inspected or deleted next is the newest remaining item. It is the structure with the motto: the last one in is the first one out.

We introduce the syntax *stack* , *empty* , *push* , *pop* , and *top* . Informally, they mean the following.

stack	a bunch consisting of all stacks of items of type X
empty	a stack containing no items (an element of bunch *stack*)
push	a function that, given a stack and an item, gives back the stack containing the same items plus the one new item
pop	a function that, given a stack, gives back the stack minus the newest remaining item
top	a function that, given a stack, gives back the newest remaining item

Here are the first four axioms.

$empty: stack$
$push: stack \to X \to stack$
$pop: stack \to stack$
$top: stack \to X$

We want *empty* and *push* to be *stack* constructors. We want a stack obtained by *pop* to be one that was constructed from *empty* and *push* , so we do not need *pop* to be a constructor. A construction axiom can be written in either of the following two ways:

$empty, push\ stack\ X: stack$
$P\ empty \land \forall s: stack \cdot \forall x: X \cdot Ps \Rightarrow P(push\ s\ x) \quad \Leftarrow \quad \forall s: stack \cdot Ps$

where *push* is allowed to distribute over bunch union, and $P: stack \to bool$. To exclude anything else from being a stack requires an induction axiom, which can be written in many ways; here are two:

$empty, push\ B\ X: B \quad \Rightarrow \quad stack: B$
$P\ empty \land \forall s: stack \cdot \forall x: X \cdot Ps \Rightarrow P(push\ s\ x) \quad \Longrightarrow \quad \forall s: stack \cdot Ps$

According to the axioms we have so far, it is possible that all stacks are equal. To say that the constructors always construct different stacks requires two more axioms. Let s and t be elements

of *stack* , and let x and y be elements of X ; then

$push\ s\ x \ne empty$

$push\ s\ x = push\ t\ y \;=\; s{=}t \wedge x{=}y$

And finally, two axioms are needed to say that stacks behave in “last in, first out” fashion.

$pop\ (push\ s\ x) = s$

$top\ (push\ s\ x) = x$

And that completes the data-stack axioms.

End of Data-Stack Theory

Data-stack theory allows us to declare as many stack variables as we want and to use them in expressions according to the axioms. We can declare variables a and b of type *stack* , and then write the assignments $a{:=}\ empty$ and $b{:=}\ push\ a\ 2$.

7.0.1 Data-Stack Implementation

If you need a stack and stacks are not provided in your programming language, you will have to build your stack using the data structures that are provided. Suppose that lists and functions are implemented. Then we can implement a stack of integers by the following definitions.

$stack \;=\; [*int]$

$empty \;=\; [nil]$

$push \;=\; \langle s: stack\rightarrow\langle x: int\rightarrow s+[x]\rangle\rangle$

$pop \;=\; \langle s: stack\rightarrow \mathbf{if}\ s{=}empty\ \mathbf{then}\ empty\ \mathbf{else}\ s\ [0;..\#s-1]\rangle$

$top \;=\; \langle s: stack\rightarrow \mathbf{if}\ s{=}empty\ \mathbf{then}\ 0\ \mathbf{else}\ s\ (\#s-1)\rangle$

To prove that a theory is implemented, we prove

(the axioms of the theory) ⇐ (the definitions of the implementation)

In other words, the definitions must satisfy the axioms. According to a distributive law, this can be done one axiom at a time. For example, the last axiom becomes

$top\ (push\ s\ x) = x$ — replace *push*

$= top\ (\langle s: stack\rightarrow\langle x: int\rightarrow s+[x]\rangle\rangle\ s\ x) = x$ — apply function

$= top\ (s+[x]) = x$ — replace *top*

$= \langle s: stack\rightarrow \mathbf{if}\ s{=}empty\ \mathbf{then}\ 0\ \mathbf{else}\ s\ (\#s-1)\rangle\ (s+[x]) = x$ — apply function and replace *empty*

$= (\mathbf{if}\ s+[x]=[nil]\ \mathbf{then}\ 0\ \mathbf{else}\ (s+[x])\ (\#(s+[x])-1)) = x$ — simplify the **if** and the index

$= (s+[x])\ (\#s) = x$ — index the list

$= x = x$ — reflexive law

$= \top$

End of Data-Stack Implementation

Is stack theory consistent? Since we implemented it using list theory, we know that if list theory is consistent, so is stack theory. Is stack theory complete? To show that a boolean expression is unclassified, we must implement stacks twice, making the expression a theorem in one implementation, and an antitheorem in the other. The expressions

$pop\ empty = empty$

$top\ empty = 0$

are theorems in our implementation, but we can alter the implementation as follows

$pop \;=\; \langle s: stack\rightarrow \mathbf{if}\ s{=}empty\ \mathbf{then}\ push\ empty\ 0\ \mathbf{else}\ s\ [0;..\#s-1]\rangle$

$top \;=\; \langle s: stack\rightarrow \mathbf{if}\ s{=}empty\ \mathbf{then}\ 1\ \mathbf{else}\ s\ (\#s-1)\rangle$

to make them antitheorems. So stack theory is incomplete.

Stack theory specifies the properties of stacks. A person who implements stacks must ensure that all these properties are provided. A person who uses stacks must ensure that only these properties are relied upon. This point deserves emphasis: a theory is a contract between two parties, an implementer and a user (they may be one person with two hats, or two corporations). It makes clear what each party's obligations are to the other, and what each can expect from the other. If something goes wrong, it makes clear who is at fault. A theory makes it possible for each side to modify their part of a program without knowing how the other part is written. This is an essential principle in the construction of large-scale software. In our small example, the stack user must not use $pop\ empty = empty$ even though the stack implementer has provided it; if the user wants it, it should be added to the theory.

7.0.2 Simple Data-Stack Theory

In the data-stack theory just presented, we have axioms $empty: stack$ and $pop: stack \rightarrow stack$; from them we can prove $pop\ empty: stack$. In other words, popping the empty stack gives a stack, though we do not know which one. An implementer is obliged to give a stack for $pop\ empty$, though it does not matter which one. If we never want to pop an empty stack, then the theory is too strong. We should weaken the axiom $pop: stack \rightarrow stack$ and remove the implementer's obligation to provide something that is not wanted. The weaker axiom

$$s \neq empty \Rightarrow pop\ s: stack$$

says that popping a nonempty stack yields a stack, but it is implied by the remaining axioms and so is unnecessary. Similarly from $empty: stack$ and $top: stack \rightarrow X$ we can prove $top\ empty: X$; deleting the axiom $top: stack \rightarrow X$ removes an implementer's obligation to provide an unwanted result for $top\ empty$.

We may decide that we have no need to prove anything about all stacks, and can do without $stack$ induction. After a little thought, we may realize that we never need an empty stack, nor to test if a stack is empty. We can always work on top of a given (possibly non-empty) stack, and in most uses we are required to do so, leaving the stack as we found it. We can delete the axiom $empty: stack$ and all mention of $empty$. We must replace this axiom with the weaker axiom $stack \neq null$ so that we can still declare variables of type $stack$. If we want to test whether a stack is empty, we can begin by pushing some special value, one that will not be pushed again, onto the stack; the empty test is then a test whether the top is the special value.

For most purposes, it is sufficient to be able to push items onto a stack, pop items off, and look at the top item. The theory we need is considerably simpler than the one presented previously. Our simpler data-stack theory introduces the names $stack$, $push$, pop , and top with the following four axioms: Let s be an element of $stack$ and let x be an element of X ; then

$$stack \neq null$$
$$push\ s\ x: stack$$
$$pop\ (push\ s\ x) = s$$
$$top\ (push\ s\ x) = x$$

End of Simple Data-Stack Theory

For the purpose of studying stacks, as a mathematical activity, we want the strongest axioms possible so that we can prove as much as possible. As an engineering activity, theory design is the art of excluding all unwanted implementations while allowing all the others. It is counter-productive to design a stronger theory than necessary; it makes implementation harder, and it makes theory extension harder.

7.0.3 Data-Queue Theory

The queue data structure, also known as a buffer, is useful in simulations and scheduling. Its distinguishing feature is that, at any time, the item to be inspected or deleted next is the oldest remaining item. It is the structure with the motto: the first one in is the first one out.

We introduce the syntax *queue* , *emptyq* , *join* , *leave* , and *front* with the following informal meaning:

queue a bunch consisting of all queues of items of type X
emptyq a queue containing no items (an element of bunch *queue*)
join a function that, given a queue and an item, gives back the queue containing the same items plus the one new item
leave a function that, given a queue, gives back the queue minus the oldest remaining item
front a function that, given a queue, gives back the oldest remaining item

The same kinds of considerations that went into the design of stack theory also guide the design of queue theory. Let q and r be elements of *queue* , and let x and y be elements of X . We certainly want the construction axioms

$emptyq: queue$
$join\ q\ x: queue$

If we want to prove things about the domain of *join* , then we must replace the second construction axiom by the stronger axiom

$join: queue \to X \to queue$

To say that the constructors construct distinct queues, with no repetitions, we need

$join\ q\ x \neq emptyq$
$join\ q\ x = join\ r\ y \;\; = \;\; q{=}r \land x{=}y$

We want a queue obtained by *leave* to be one that was constructed from *emptyq* and *join* , so we do not need

$leave\ q: queue$

for construction, and we don't want to oblige an implementer to provide a representation for *leave emptyq* , so perhaps we will omit that one. We do want to say

$q{\neq}emptyq \;\; \Rightarrow \;\; leave\ q: queue$

And similarly, we want

$q{\neq}emptyq \;\; \Rightarrow \;\; front\ q: X$

If we want to prove something about all queues, we need *queue* induction:

$emptyq, join\ B\ X: B \;\; \Rightarrow \;\; queue: B$

And finally, we need to give queues their "first in, first out" character:

$leave\ (join\ emptyq\ x) \;=\; emptyq$
$q{\neq}emptyq \;\; \Rightarrow \;\; leave\ (join\ q\ x) = join\ (leave\ q)\ x$
$front\ (join\ emptyq\ x) \;=\; x$
$q{\neq}emptyq \;\; \Rightarrow \;\; front\ (join\ q\ x) = front\ \ q$

If we have decided to keep the *queue* induction axiom, we can throw away the two earlier axioms

$q{\neq}emptyq \;\; \Rightarrow \;\; leave\ q: queue$
$q{\neq}emptyq \;\; \Rightarrow \;\; front\ q: X$

since they can now be proven.

End of Data-Queue Theory

After data-stack implementation, data-queue implementation raises no new issues, so we leave it as Exercise 340.

7.0.4 Data-Tree Theory

We introduce the syntax

tree	a bunch consisting of all finite binary trees of items of type X
emptree	a tree containing no items (an element of bunch *tree*)
graft	a function that, given two trees and an item, gives back the tree with the item at the root and the two given trees as left and right subtree
left	a function that, given a tree, gives back its left subtree
right	a function that, given a tree, gives back its right subtree
root	a function that, given a tree, gives back its root item

For the purpose of studying trees, we want a strong theory. Let t , u , v , and w be elements of *tree* , and let x and y be elements of X .

$emptree\colon tree$
$graft\colon tree \to X \to tree \to tree$
$emptree, graft\ B\ X\ B\colon B \ \Rightarrow\ tree\colon B$
$graft\ t\ x\ u \neq emptree$
$graft\ t\ x\ u = graft\ v\ y\ w \ =\ t=v \wedge x=y \wedge u=w$
$left\ (graft\ t\ x\ u) = t$
$root\ (graft\ t\ x\ u) = x$
$right\ (graft\ t\ x\ u) = u$

where, in the construction axiom, *graft* is allowed to distribute over bunch union.

For most programming purposes, the following simpler, weaker theory is sufficient.

$tree \neq null$
$graft\ t\ x\ u\colon tree$
$left\ (graft\ t\ x\ u) = t$
$root\ (graft\ t\ x\ u) = x$
$right\ (graft\ t\ x\ u) = u$

As with stacks, we don't really need to be given an empty tree. As long as we are given some tree, we can build a tree with a distinguished root that serves the same purpose. And we probably don't need *tree* induction.

End of Data-Tree Theory

7.0.5 Data-Tree Implementation

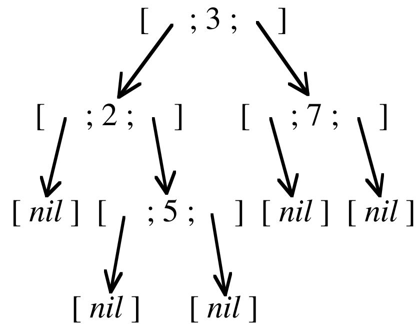
Suppose lists and recursive data definition are implemented. Then we can implement a tree of integers by the following definitions:

$tree\ =\ emptree, graft\ tree\ int\ tree$
$emptree\ =\ [nil]$
$graft\ =\ \langle t\colon tree \to \langle x\colon int \to \langle u\colon tree \to [t; x; u]\rangle\rangle\rangle$
$left\ =\ \langle t\colon tree \to t\ 0\rangle$
$right\ =\ \langle t\colon tree \to t\ 2\rangle$
$root\ =\ \langle t\colon tree \to t\ 1\rangle$

The procedure *graft* makes a list of three items; two of those items are lists themselves. A reasonable implementation strategy for lists is to allocate a small space, one capable of holding an integer or data address, for each item. If an item is an integer, it is put in its place; if an item is a list, it is put somewhere else and a pointer to it (data address) is put in its place. In this implementation of lists, pointers are provided automatically when needed. For example, the tree

$[[[nil]; 2; [[nil]; 5; [nil]]]; 3; [[nil]; 7; [nil]]]$

looks like



Here is another implementation of data-trees.

$tree = emptree, graft\ tree\ int\ tree$
$emptree = 0$
$graft = \langle t: tree \rightarrow \langle x: int \rightarrow \langle u: tree \rightarrow (\text{"left"} \rightarrow t \mid \text{"root"} \rightarrow x \mid \text{"right"} \rightarrow u)\rangle\rangle\rangle$
$left = \langle t: tree \rightarrow t\ \text{"left"}\rangle$
$right = \langle t: tree \rightarrow t\ \text{"right"}\rangle$
$root = \langle t: tree \rightarrow t\ \text{"root"}\rangle$

With this implementation, a tree value looks like this:

```
  "left" →   ("left" → 0
             | "root" → 2
             | "right" →  ("left" → 0
                          | "root" → 5
                          | "right" → 0 ) )
| "root" → 3
| "right" →  ("left" → 0
             | "root" → 7
             | "right" → 0 )
```

If the implementation you have available does not include recursive data definition, you will have to build the pointer structure yourself. For example, in C you can code the implementation of binary trees as follows:

```
struct tree { struct tree *left;  int root;  struct tree *right; };
struct tree *emptree = NULL;
struct tree *graft (struct tree *t, int x, struct tree *u)
    {   struct tree *g;  g = malloc (sizeof(struct tree));
        (*g).left = t;  (*g).root = x;  (*g).right = u;
        return g;
    }
struct tree *left (struct tree *t) { return (*t).left; }
int root (struct tree *t) { return (*t).root; }
struct tree *right (struct tree *t) { return (*t).right; }
```

As you can see, the C code is clumsy. It is not a good idea to apply Program Theory directly to the C code. The use of pointers (data addresses) when recursive data definition is unimplemented is just like the use of **go to** (program addresses) when recursive program definition is unimplemented or implemented badly.

End of Data-Tree Implementation

End of Data Theories

A data theory creates a new type, or value space, or perhaps an extension of an old type. A program theory creates new programs, or rather, new specifications that become programs when the theory is implemented. These two styles of theory correspond to two styles of programming: functional and imperative.

7.1 Program Theories

In program theories, the state is divided into two kinds of variables: the user's variables and the implementer's variables. A user of the theory enjoys full access to the user's variables, but cannot directly access (see or change) the implementer's variables. A user gets access to the implementer's variables only through the theory. On the other side, an implementer of the theory enjoys full access to the implementer's variables, but cannot directly access (see or change) the user's variables. An implementer gets access to the user's variables only through the theory. Some programming languages have a "module" or "object" construct exactly for this purpose. In other languages we just forbid the use of the wrong variables on each side of the boundary.

If we need only one stack or one queue or one tree, we can obtain an economy of expression and of execution by leaving it implicit. There is no need to say which stack to push onto if there is only one, and similarly for the other operations and data structures. Each of the program theories we present will provide only one of its type of data structure to the user, but they can be generalized by adding an extra parameter to each operation.

7.1.0 Program-Stack Theory

The simplest version of program-stack theory introduces three names: *push* (a procedure with parameter of type X), *pop* (a program), and *top* (of type X). In this theory, *push* 3 is a program (assuming $3: X$); it changes the state. Following this program, before any other pushes and pops, *print top* will print 3 . The following two axioms are sufficient.

$$top'=x \;\Leftarrow\; push\ x$$
$$ok \;\Leftarrow\; push\ x.\ pop$$

where $x: X$.

The second axiom says that a pop undoes a push. In fact, it says that any natural number of pushes are undone by the same number of pops.

	ok	use second axiom
$\Leftarrow$	$push\ x.\ pop$	ok is identity for dependent composition
$=$	$push\ x.\ ok.\ pop$	Refinement by Steps reusing the axiom
$\Leftarrow$	$push\ x.\ push\ y.\ pop.\ pop$	

We can prove things like

$$top'=x \;\Leftarrow\; push\ x.\ push\ y.\ push\ z.\ pop.\ pop$$

which say that when we push something onto the stack, we find it there later at the appropriate time. That is all we really want.

End of Program-Stack Theory

7.1.1 Program-Stack Implementation

To implement program-stack theory, we introduce an implementer's variable $s: [*X]$ and define

$$push \;=\; \langle x: X \rightarrow s:= s^+[x]\rangle$$
$$pop \;=\; s:= s\,[0;..\#s-1]$$
$$top \;=\; s\,(\#s-1)$$

And, of course, we must show that these definitions satisfy the axioms. We'll do the first axiom, and leave the other as Exercise 342.

$(top'=x \Leftarrow push\ x)$ use definition of *push* and *top*

$= (s'(\#s'-1)=x \Leftarrow s:= s+[x])$ List Theory

$= \top$

End of Program-Stack Implementation

7.1.2 Fancy Program-Stack Theory

The program-stack theory just presented corresponds to the simpler data-stack theory presented earlier. A slightly fancier program-stack theory introduces two more names: *mkempty* (a program to make the stack empty) and *isempty* (a condition to say whether the stack is empty). Letting $x: X$, the axioms are

$top'=x \land \neg isempty' \Leftarrow push\ x$

$ok \Leftarrow push\ x.\ pop$

$isempty' \Leftarrow mkempty$

End of Fancy Program-Stack Theory

Once we implement program-stack theory using lists, we know that program-stack theory is consistent if list theory is consistent. Program-stack theory, like data-stack theory, is incomplete. Incompleteness is a freedom for the implementer, who can trade economy against robustness. If we care how this trade will be made, we should strengthen the theory. For example, we could add the axiom

$print$ "error" $\Leftarrow mkempty.\ pop$

7.1.3 Weak Program-Stack Theory

The program-stack theory we presented first can be weakened and still retain its stack character. We must keep the axiom

$top'=x \Leftarrow push\ x$

but we do not need the composition $push\ x.\ pop$ to leave all variables unchanged. We do require that any natural number of pushes followed by the same number of pops gives back the original top. The axioms are

$top'=top \Leftarrow balance$

$balance \Leftarrow ok$

$balance \Leftarrow push\ x.\ balance.\ pop$

where *balance* is a specification that helps in writing the axioms, but is not an addition to the theory, and does not need to be implemented. To prove an implementation is correct, we must propose a definition for *balance* that uses the implementer's variables, but it doesn't have to be a program. This weaker theory allows an implementation in which popping does not restore the implementer's variable s to its pre-pushed value, but instead marks the last item as “garbage”.

A weak theory can be extended in ways that are excluded by a strong theory. For example, we can add the names *count* (of type *nat*) and *start* (a program), with the axioms

$count' = 0 \Leftarrow start$

$count' = count+1 \Leftarrow push\ x$

$count' = count+1 \Leftarrow pop$

so that *count* counts the number of pushes and pops since the last use of *start*.

End of Weak Program-Stack Theory

7.1.4 Program-Queue Theory

Program-queue theory introduces five names: *mkemptyq* (a program to make the queue empty), *isemptyq* (a condition to say whether the queue is empty), *join* (a procedure with parameter of type X), *leave* (a program), and *front* (of type X). The axioms are

$isemptyq' \Leftarrow mkemptyq$

$isemptyq \Rightarrow front'=x \land \neg isemptyq' \Leftarrow join\ x$

$\neg isemptyq \Rightarrow front'=front \land \neg isemptyq' \Leftarrow join\ x$

$isemptyq \Rightarrow (join\ x.\ leave = mkemptyq)$

$\neg isemptyq \Rightarrow (join\ x.\ leave = leave.\ join\ x)$

End of Program-Queue Theory

7.1.5 Program-Tree Theory

As usual, there is more than one way to do it. Imagine a tree that is infinite in all directions; there are no leaves and no root. You are standing at one node in the tree facing one of the three directions *up* (towards the parent of this node), *left* (towards the left child of this node), or *right* (towards the right child of this node). Variable *node* (of type X) tells the value of the item where you are, and it can be assigned a new value. Variable *aim* tells what direction you are facing, and it can be assigned a new direction. Program *go* moves you to the next node in the direction you are facing, and turns you facing back the way you came. For example, we might begin with

$aim := up.\ go$

and then look at *aim* to see where we came from. For later use, we might then assign

$node := 3$

The axioms use an auxiliary specification that helps in writing the axioms, but is not an addition to the theory, and does not need to be implemented: *work* means "Do anything, wander around changing the values of nodes if you like, but do not *go* from this node (your location at the start of *work*) in this direction (the value of variable *aim* at the start of *work*). End where you started, facing the way you were facing at the start.". Here are the axioms.

$(aim=up) = (aim' \neq up) \Leftarrow go$

$node'=node \land aim'=aim \Leftarrow go.\ work.\ go$

$work \Leftarrow ok$

$work \Leftarrow node := x$

$work \Leftarrow a=aim \neq b \land (aim := b.\ go.\ work.\ go.\ aim := a)$

$work \Leftarrow work.\ work$

Here is another way to define program-trees. Let T (for tree) and p (for pointer) be implementer's variables. The axioms are

$tree = [tree; X; tree]$

$T: tree$

$p: *(0, 1, 2)$

$node = T@(p; 1)$

$change = \langle x: X \rightarrow T := (p; 1) \rightarrow x \mid T \rangle$

$goUp = p := p_{0;..\leftrightarrow p-1}$

$goLeft = p := p;0$

$goRight = p := p;2$

If strings and the @ operator are implemented, then this theory is already an implementation. If not, it is still a theory, and should be compared to the previous theory for clarity.

End of Program-Tree Theory

End of Program Theories

7.2 Data Transformation

A program is a specification of computer behavior. Sometimes (but not always) a program is the clearest kind of specification. Sometimes it is the easiest kind of specification to write. If we write a specification as a program, there is no work to implement it. Even though a specification may already be a program, we can, if we like, implement it differently. In some programming languages, implementer's variables are distinguished by being placed inside a “module” or “object”, so that changing them is not visible outside the object or module. Perhaps the implementer's variables were chosen to make the specification as clear as possible, but other implementer's variables might be more storage-efficient, or provide faster access on average. Since a theory user has no access to the implementer's variables except through the theory, an implementer is free to change them in any way that provides the same theory to the user. Here's one way.

We can replace the implementer's variables v by new implementer's variables w using a data transformer, which is a boolean expression D relating v and w such that

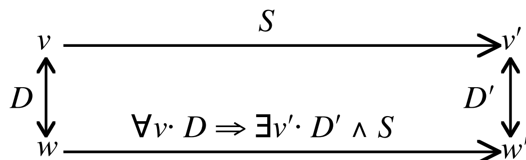
$$\forall w\cdot \exists v\cdot D$$

Here, v and w represent any number of variables. Let D' be the same as D but with primes on all the variables. Then each specification S in the theory is transformed to

$$\forall v\cdot D \Rightarrow \exists v'\cdot D' \land S$$

Specification S talks about its nonlocal variables v (and the user's variables), and the transformed specification talks about its nonlocal variables w (and the user's variables).

Data transformation is invisible to the user. The user imagines that the implementer's variables are initially in state v, and then, according to specification S, they are finally in state v'. Actually, the implementer's variables will initially be in state w related to v by D; the user will be able to suppose they are in a state v because $\forall w\cdot \exists v\cdot D$. The implementer's variables will change state from w to w' according to the transformed specification $\forall v\cdot D \Rightarrow \exists v'\cdot D' \land S$. This says that whatever related initial state v the user was imagining, there is a related final state v' for the user to imagine as the result of S, and so the fiction is maintained. Here is a picture of it.



Implementability of S in its variables (v, v') becomes, via the transformer (D, D'), the new specification in the new variables (w, w').

Our first example is Exercise 363(a). The user's variable is u: *bool* and the implementer's variable is v: *nat*. The theory provides three operations, specified by

$$zero \;=\; v:= 0$$
$$increase \;=\; v:= v+1$$
$$inquire \;=\; u:= even\ v$$

Since the only question asked of the implementer's variable is whether it is even, we decide to replace it by a new implementer's variable w: *bool* according to the data transformer $w = even\ v$. The first operation *zero* becomes

$\forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \land\ (v{:=}\ 0)$

The assignment refers to a state consisting of u and v.

$=\quad \forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \land\ u'{=}u\ \land\ v'{=}0$ — One-Point law

$=\quad \forall v\cdot\ w = even\ v\ \Rightarrow\ w' = even\ 0\ \land\ u'{=}u$ — change of variable law, simplify

$=\quad \forall r{:}\ even\ nat\cdot\ w{=}r\ \Rightarrow\ w'{=}\top\ \land\ u'{=}u$ — One-Point law

$=\quad w'{=}\top\ \land\ u'{=}u$ — The state now consists of u and w.

$=\quad w{:=}\ \top$

Operation *increase* becomes

$\forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \land\ (v{:=}\ v{+}1)$

$=\quad \forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \land\ u'{=}u\ \land\ v'{=}v{+}1$ — One-Point law

$=\quad \forall v\cdot\ w = even\ v\ \Rightarrow\ w' = even\ (v{+}1)\ \land\ u'{=}u$ — change of variable law, simplify

$=\quad \forall r{:}\ even\ nat\cdot\ w{=}r\ \Rightarrow\ w' = \neg r\ \land\ u'{=}u$ — One-Point law

$=\quad w' = \neg w\ \land\ u'{=}u$

$=\quad w{:=}\ \neg w$

Operation *inquire* becomes

$\forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \land\ (u{:=}\ even\ v)$

$=\quad \forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \land\ u' = even\ v\ \land\ v'{=}v$ — One-Point law

$=\quad \forall v\cdot\ w = even\ v\ \Rightarrow\ w' = even\ v\ \land\ u' = even\ v$ — change of variable law

$=\quad \forall r{:}\ even\ nat\cdot\ w{=}r\ \Rightarrow\ w'{=}r\ \land\ u'{=}r$ — One-Point law

$=\quad w'{=}w\ \land\ u'{=}w$

$=\quad u{:=}\ w$

In the previous example, we replaced a bigger state space by a smaller state space. Just to show that it works both ways, here is Exercise 364(a). The user's variable is $u{:}\ bool$ and the implementer's variable is $v{:}\ bool$. The theory provides three operations, specified by

$set\ =\ v{:=}\ \top$

$flip\ =\ v{:=}\ \neg v$

$ask\ =\ u{:=}\ v$

We decide to replace the implementer's variable by a new implementer's variable $w{:}\ nat$ (perhaps for easier access on some computers) according to the data transformer $v = even\ w$. The first operation *set* becomes

$\forall v\cdot\ v = even\ w\ \Rightarrow\ \exists v'\cdot\ v' = even\ w'\ \land\ (v{:=}\ \top)$ — One-Point law twice

$=\quad even\ w'\ \land\ u'{=}u$

$\Leftarrow\quad w{:=}\ 0$

Operation *flip* becomes

$\forall v\cdot\ v = even\ w\ \Rightarrow\ \exists v'\cdot\ v' = even\ w'\ \land\ (v{:=}\ \neg v)$ — One-Point law twice

$=\quad even\ w' \ne even\ w\ \land\ u'{=}u$

$\Leftarrow\quad w{:=}\ w{+}1$

Operation *ask* becomes

$\forall v\cdot\ v = even\ w\ \Rightarrow\ \exists v'\cdot\ v' = even\ w'\ \land\ (u{:=}\ v)$ — One-Point law twice

$=\quad even\ w' = even\ w = u'$

$\Leftarrow\quad u{:=}\ even\ w$

A data transformation does not have to replace all the implementer's variables, and the number of variables being replaced does not have to equal the number of variables replacing them. A data transformation can be done by steps, as a sequence of smaller transformations. A data transformation can be done by parts, as a conjunction of smaller transformations. The next few subsections are examples to illustrate these points.

7.2.0 Security Switch

Exercise 367 is to design a security switch. It has three boolean user's variables a , b , and c . The users assign values to a and b as input to the switch. The switch's output is assigned to c . The output changes when both inputs have changed. More precisely, the output changes when both inputs differ from what they were the previous time the output changed. The idea is that one user might flip their input indicating a desire for the output to change, but the output does not change until the other user flips their input indicating agreement that the output should change. If the first user changes back before the second user changes, the output does not change.

We can implement the switch with two boolean implementer's variables:

A records the state of input a at last output change
B records the state of input b at last output change

There are two operations:

$a:= \neg a.\ \ \textbf{if } a\ne A \wedge b\ne B \textbf{ then } (c:= \neg c.\ \ A:= a.\ \ B:= b) \textbf{ else } ok$

$b:= \neg b.\ \ \textbf{if } a\ne A \wedge b\ne B \textbf{ then } (c:= \neg c.\ \ A:= a.\ \ B:= b) \textbf{ else } ok$

In each operation, a user flips their input variable, and the switch checks if this input assignment makes both inputs differ from what they were at last output change; if so, the output is changed, and the current input values are recorded. This implementation is a direct formalization of the problem, but it can be simplified by data transformation.

We replace implementer's variables A and B by nothing according to the transformer

$A=B=c$

To check that this is a transformer, we check

$\exists A, B\cdot\ A=B=c$ — generalization, using c for both A and B
$\Leftarrow\ \ \top$

There are no new variables, so there was no universal quantification. The transformation does not affect the assignments to a and b , so we have only one transformation to make.

$$\forall A, B\cdot\ \ A=B=c \ \Rightarrow\ \exists A', B'\cdot\ \ A'=B'=c' \ \wedge\ \textbf{if } a\ne A \wedge b\ne B \textbf{ then } (c:= \neg c.\ \ A:= a.\ \ B:= b) \textbf{ else } ok$$

expand assignments and ok

$$= \ \ \forall A, B\cdot\ \ A=B=c \ \Rightarrow\ \exists A', B'\cdot\ \ A'=B'=c' \ \wedge\ \textbf{if } a\ne A \wedge b\ne B \textbf{ then } (a'=a \wedge b'=b \wedge c'=\neg c \wedge A'=a \wedge B'=b) \textbf{ else } (a'=a \wedge b'=b \wedge c'=c \wedge A'=A \wedge B'=B)$$

one-point for A' and B'

$$= \ \ \forall A, B\cdot\ A=B=c \ \Rightarrow\ \textbf{if } a\ne A \wedge b\ne B \textbf{ then } (a'=a \wedge b'=b \wedge c'=\neg c \wedge c'=a \wedge c'=b) \textbf{ else } (a'=a \wedge b'=b \wedge c'=c \wedge c'=A \wedge c'=B)$$

one-point for A and B

$$= \ \ \textbf{if } a\ne c \wedge b\ne c \textbf{ then } (a'=a \wedge b'=b \wedge c'=\neg c \wedge c'=a \wedge c'=b) \textbf{ else } (a'=a \wedge b'=b \wedge c'=c \wedge c'=c \wedge c'=c)$$

use **if**-part as context to change **then**-part

$$= \ \ \textbf{if } a\ne c \wedge b\ne c \textbf{ then } (a'=a \wedge b'=b \wedge c'=\neg c \wedge c'=\neg c \wedge c'=\neg c) \textbf{ else } (a'=a \wedge b'=b \wedge c'=c \wedge c'=c \wedge c'=c)$$

$$= \ \ \textbf{if } a\ne c \wedge b\ne c \textbf{ then } c:= \neg c \textbf{ else } ok$$

$$= \ \ c:= (a\ne c \wedge b\ne c) \ne c$$

Output c becomes the majority value of a , b , and c . (As a circuit, that's three “exclusive or” gates and one “and” gate.)

End of Security Switch

7.2.1 Take a Number

The next example is Exercise 370 (take a number): Maintain a list of natural numbers standing for those that are "in use". The three operations are:

- make the list empty (for initialization)
- assign to variable n a number that is not in use, and add this number to the list (now it is in use)
- given a number n that is in use, remove it from the list (now it is no longer in use, and it can be reused later)

The user's variable is $n: nat$. Although the exercise talks about a list, we see from the operations that the items are always distinct, their order is irrelevant, and there is no nesting structure; that suggests using a bunch variable. But we will need to quantify over this variable, so we need it to be an element. We therefore use a set variable $s \subseteq \{nat\}$ as our implementer's variable. The three operations are

$$start \;=\; s'=\{null\}$$
$$take \;=\; \neg\, n' \in s \;\wedge\; s' = s \cup \{n'\}$$
$$give \;=\; n \in s \;\Rightarrow\; \neg\, n \in s' \;\wedge\; s' \cup \{n\} = s$$

Here is a data transformation that replaces set s with natural m according to the transformer

$$s \subseteq \{0,..m\}$$

Instead of maintaining the exact set of numbers that are in use, we will maintain a possibly larger set. We will still never give out a number that is in use. We transform $start$ as follows.

$$\forall s\cdot\; s \subseteq \{0,..m\} \;\Rightarrow\; \exists s'\cdot\; s' \subseteq \{0,..m'\} \wedge s'=\{null\} \qquad \text{one-point and identity}$$
$$=\quad \top$$
$$\Leftarrow\quad ok$$

The transformed specification is just $\top$, which is most efficiently refined as ok . Since s is only a subset of $\{0,..m\}$, not necessarily equal to $\{0,..m\}$, it does not matter what m is; we may as well leave it alone. Operation $take$ is transformed as follows.

$$\forall s\cdot\; s \subseteq \{0,..m\} \;\Rightarrow\; \exists s'\cdot\; s' \subseteq \{0,..m'\} \;\wedge\; \neg\, n' \in s \;\wedge\; s' = s \cup \{n'\} \qquad \text{several omitted steps}$$
$$=\quad m \le n' < m'$$
$$\Leftarrow\quad n:= m.\;\; m:= m+1$$

Operation $give$ is transformed as follows.

$$\forall s\cdot\; s \subseteq \{0,..m\} \;\Rightarrow\; \exists s'\cdot\; s' \subseteq \{0,..m'\} \;\wedge\; (n \in s \;\Rightarrow\; \neg\, n \in s' \;\wedge\; s' \cup \{n\} = s) \qquad \text{several omitted steps}$$
$$=\quad (n+1 = m \Rightarrow n \le m') \wedge (n+1 < m \Rightarrow m \le m')$$
$$\Leftarrow\quad ok$$

Thanks to the data transformation, we have an extremely efficient solution to the problem. One might argue that we have not solved the problem at all, because we do not maintain a list of numbers that are "in use". But who can tell? The only use made of the list is to obtain a number that is not currently in use, and that service is provided.

Our implementation of the "take a number" problem corresponds to the "take a number" machines that are common at busy service centers. Now suppose we want to provide two "take a number" machines that can operate independently. We might try replacing s with two variables $i, j: nat$ according to the transformer $s \subseteq \{0,..max\; i\; j\}$. Operation $take$ becomes

$$\forall s\cdot\; s \subseteq \{0,..max\; i\; j\} \;\Rightarrow\; \exists s'\cdot\; s' \subseteq \{0,..max\; i'\; j'\} \;\wedge\; \neg\, n' \in s \;\wedge\; s' = s \cup \{n'\} \qquad \text{several omitted steps}$$
$$=\quad max\; i\; j \le n' < max\; i'\; j'$$
$$\Leftarrow\quad n:= max\; i\; j.\;\; \mathbf{if}\; i \ge j\; \mathbf{then}\; i:= i+1\; \mathbf{else}\; j:= j+1$$

From the program on the last line we see that this data transformation does not provide the independent operation of two machines as we were hoping. Perhaps a different data transformation will work better. Let's put the even numbers on one machine and the odd numbers on the other. The new variables are i: $2\times nat$ and j: $2\times nat+1$. The transformer is

$$\forall k\text{: } \sim s\cdot\ even\ k \land k<i\ \lor\ odd\ k \land k<j$$

Now *take* becomes

$$\forall s\cdot\quad (\forall k\text{: } \sim s\cdot\ even\ k \land k<i\ \lor\ odd\ k \land k<j)$$
$$\Rightarrow\ \exists s'\cdot\ (\forall k\text{: } \sim s'\cdot\ even\ k \land k<i'\ \lor\ odd\ k \land k<j')\ \land\ \neg\ n'\in s\ \land\ s' = s\cup\{n'\}$$

several omitted steps

$$=\quad even\ n'\ \land\ i\le n'<i'\ \lor\ odd\ n'\ \land\ j\le n'<j'$$
$$\Leftarrow\quad (n\text{:= } i.\ i\text{:= } i+2)\ \lor\ (n\text{:= } j.\ j\text{:= } j+2)$$

Now we have a “distributed” solution to the problem: we can take a number from either machine without disturbing the other. The price of the distribution is that we have lost all fairness between the two machines; a recently arrived customer using one machine may be served before an earlier customer using the other machine.

End of Take a Number

7.2.2 Parsing

Exercise 362 (parsing): Define E as a bunch of strings of lists of characters satisfying

$$E\ =\ [\text{"x"}],\ \ [\text{"if"}];E;[\text{"then"}];E;[\text{"else"}];E$$

Given a string of lists of characters, write a program to determine if the string is in the bunch E .

For the problem to be nontrivial, we assume that recursive data definition and bunch inclusion are not implemented. The solution will have to be a search, so we need a variable to represent the bunch of strings still in contention, beginning with all the strings in E , eliminating strings as we go, and ending either when the given string is found or when none of the remaining strings is the given string.

Let the given string be s (a constant). Our first decision is to parse from left to right, so we introduce natural variable n , increasing from 0 to at most $\leftrightarrow s$, indicating how much of s we have parsed. Let A be a variable whose value is a bunch of strings of lists of characters. Bunch A will consist of all strings in E that might possibly be s according to what we have seen of s . We can express the result as the final value of boolean variable q .

To reduce the number of cases that we have to consider, we will use two sentinels. We assume that s ends with the sentinel ["eos"] (end of string); this is an item that cannot appear anywhere except at the end of s (some programming languages provide this sentinel automatically). And when we initialize variable A , we will add the sentinel ["eog"] (end of grammar) to the end of every string, and assume that ["eog"] cannot appear anywhere except at the end of strings in A . The problem and its refinement are as follows:

$$q' = (s_{0;..\leftrightarrow s-1} : E)\ \ \Leftarrow\ \ A\text{:= } E;[\text{"eog"}].\ \ n\text{:= }0.\ \ P$$

where $P\ =\ n\le\leftrightarrow s\ \land\ A_{0;..n} = s_{0;..n}\ \Rightarrow\ q' = (s_{0;..\leftrightarrow s-1};[\text{"eog"}] : A)$. In words, the new problem P says that if the strings in A look like s up to index n , then the question is whether s is in A (with a suitable adjustment of sentinels). The proof of this refinement uses the fact that E is a nonempty bunch, but we will not need the fact that E is a bunch of nonempty strings. Here is the refinement of the remaining problem.

$P \Leftarrow$ **if** s_n: A_n **then** $(A{:=} (\S a{:}\ A\cdot\ a_n = s_n).\ \ n{:=}\ n+1.\ \ P)$
else $q{:=}$ ["eog"]: $A_n\ \wedge\ s_n$=["eos"]

From P we know that all strings in A are identical to s up to index n . If there are strings in A that agree with s at index n , then we reduce bunch A to just those strings, and move along one index. If not, then either we have run out of candidates and we should assign $\bot$ to q , or we have come to the end of s and also to the end of one of the candidates and we should assign $\top$ to q . We omit the proofs of these refinements in order to pursue our current topic, data transformation.

We now replace variable A with variable b whose value is a single string of lists of characters. We represent bunch E with ["E"] , which we assume cannot be in the given string s . (In parsing theory "E" is called a “nonterminal”.) For example, the string

["if"]; ["x"]; ["then"]; ["E"]; ["else"]; ["E"]

represents the bunch of strings

["if"]; ["x"]; ["then"]; E; ["else"]; E

The data transformer is, informally,

A = (b with all occurrences of item ["E"] replaced by bunch E)

Let Q be the result of transforming P . The result of the transformation is as follows.

$q' = (s_{0;..\leftrightarrow s-1} : E)\ \ \Leftarrow\ \ b{:=}$ ["E"];["eog"]. $n{:=}\ 0.\ \ Q$

$Q \Leftarrow$ **if** $s_n{=}b_n$ **then** $(n{:=}\ n+1.\ \ Q)$
else if b_n=["E"] $\wedge$ s_n=["x"] **then** ($b{:=}\ b_{0;..n}$;["x"];$b_{n+1;..\leftrightarrow b}$. $n{:=}\ n+1.\ \ Q$)
else if b_n=["E"] $\wedge$ s_n=["if"]
then ($b{:=}\ b_{0;..n}$;["if"];["E"];["then"];["E"];["else"];["E"];$b_{n+1;..\leftrightarrow b}$. $n{:=}\ n+1.\ \ Q$)
else $q{:=}\ b_n$=["eog"] $\wedge$ s_n=["eos"]

We can make a minor improvement by changing the representation of E from ["E"] to ["x"] ; then one of the cases disappears, and we get

$q' = (s_{0;..\leftrightarrow s-1} : E)\ \ \Leftarrow\ \ b{:=}$ ["x"];["eog"]. $n{:=}\ 0.\ \ Q$

$Q \Leftarrow$ **if** $s_n{=}b_n$ **then** $(n{:=}\ n+1.\ \ Q)$
else if b_n=["x"] $\wedge$ s_n=["if"]
then ($b{:=}\ b_{0;..n}$;["if"];["x"];["then"];["x"];["else"];["x"];$b_{n+1;..\leftrightarrow b}$. $n{:=}\ n+1.\ \ Q$)
else $q{:=}\ b_n$=["eog"] $\wedge$ s_n=["eos"]

Our next improvement is to notice that we don't need the initial portion of b , which is identical to the initial portion of s . So we transform again, replacing b with c using the transformer

$b\ =\ s_{0;..n};c$

Let R be the result of transforming Q . The result of the transformation is as follows.

$q' = (s_{0;..\leftrightarrow s-1} : E)\ \ \Leftarrow\ \ c{:=}$ ["x"];["eog"]. $n{:=}\ 0.\ \ R$

$R \Leftarrow$ **if** $s_n{=}c_0$ **then** $(c{:=}\ c_{1;..\leftrightarrow c}.\ \ n{:=}\ n+1.\ \ R)$
else if c_0=["x"] $\wedge$ s_n=["if"] **then** ($c{:=}$ ["x"];["then"];["x"];["else"];c. $n{:=}\ n+1.\ \ R$)
else $q{:=}\ c_0$=["eog"] $\wedge$ s_n=["eos"]

Variable c behaves as a stack, so we could replace it by stack operations.

End of Parsing

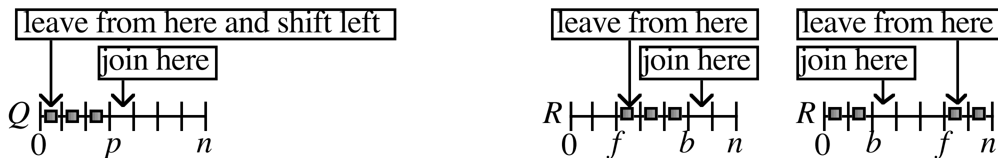
7.2.3 Limited Queue

The next example, Exercise 371, transforms a limited queue to achieve a time bound that is not met by the original implementation. A limited queue is a queue with a limited number of places for items. Let the limit be positive natural n, and let Q: $[n*X]$ and p: *nat* be implementer's variables. Then the original implementation is as follows.

$mkemptyq = p := 0$
$isemptyq = p=0$
$isfullq = p=n$
$join\ x = Qp := x.\ p := p+1$
$leave = \textbf{for}\ i := 1;..p\ \textbf{do}\ Q(i-1) := Qi.\ p := p-1$
$front = Q0$

A user of this theory would be well advised to precede any use of *join* with the test $\neg isfullq$, and any use of *leave* or *front* with the test $\neg isemptyq$.

A new item joins the back of the queue at position p taking zero time (measured recursively) to do so. The front item is always found instantly at position 0. Unfortunately, removing the front item from the queue takes time $p-1$ to shift all remaining items down one index. We want to transform the queue so that all operations are instant. Variables Q and p will be replaced by R: $[n*X]$ and f, b: $0,..n$ with f and b indicating the current front and back.



The idea is that b and f move cyclically around the list; when f is to the left of b the queue items are between them; when b is to the left of f the queue items are in the outside portions. Here is the data transformer D.

$$\begin{aligned} & 0 \le p = b-f < n \ \land\ Q[0;..p] = R[f;..b] \\ \lor\ \ & 0 < p = n-f+b \le n \ \land\ Q[0;..p] = R[(f;..n); (0;..b)] \end{aligned}$$

Now we transform. First *mkemptyq*.

$$\begin{aligned} & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land p'=0 \land Q'=Q && \text{several omitted steps} \\ = \ \ & f'=b' \\ \Leftarrow \ \ & f := 0.\ b := 0 \end{aligned}$$

Next we transform *isemptyq*. Although *isemptyq* happens to be boolean and can be interpreted as an unimplementable specification, its purpose (like *front*, which isn't boolean) is to tell the user about the state of the queue. We don't transform arbitrary expressions; we transform implementable specifications (usually programs). So we suppose c is a user's variable, and transform $c := isemptyq$.

$$\begin{aligned} & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land c'=(p=0) \land p'=p \land Q'=Q && \text{several omitted steps} \\ = \ \ & \quad f<b \ \land\ f'<b' \ \land\ b-f = b'-f' \ \land\ R[f;..b] = R'[f';..b'] \ \land\ \neg c' \\ & \lor\ \ f<b \ \land\ f'>b' \ \land\ b-f = n+b'-f' \ \land\ R[f;..b] = R'[(f';..n); (0;..b')] \ \land\ \neg c' \\ & \lor\ \ f>b \ \land\ f'<b' \ \land\ n+b-f = b'-f' \ \land\ R[(f;..n); (0;..b)] = R'[f';..b'] \ \land\ \neg c' \\ & \lor\ \ f>b \ \land\ f'>b' \ \land\ b-f = b'-f' \ \land\ R[(f;..n); (0;..b)]=R'[(f';..n); (0;..b')] \ \land\ \neg c' \end{aligned}$$

Initially R might be in the "inside" or "outside" configuration, and finally R' might be either way, so that gives us four disjuncts. Very suspiciously, we have $\neg c'$ in every case. That's because $f=b$ is missing! So the transformed operation is unimplementable. That's the transformer's way of telling us that the new variables do not hold enough information to answer whether the queue is empty. The problem occurs when $f=b$ because that could be either an empty queue or a full queue. A solution is to add a new variable m: *bool* to say whether we have the "inside" mode or "outside" mode. We revise the transformer D as follows:

$$m \land 0 \le p = b-f < n \land Q[0;..p] = R[f;..b]$$
$$\lor \quad \neg m \land 0 < p = n-f+b \le n \land Q[0;..p] = R[(f;..n); (0;..b)]$$

Now we have to retransform *mkemptyq* .

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land p'=0 \land Q'=Q \qquad \text{several omitted steps}$$
$$= \quad m' \land f'=b'$$
$$\Leftarrow \quad m:= \top.\ f:= 0.\ b:= 0$$

Next we transform $c:= isemptyq$.

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land c'=(p=0) \land p'=p \land Q'=Q \qquad \text{several omitted steps}$$
$$= \quad m \land f<b \land m' \land f'<b' \land b-f = b'-f' \land R[f;..b] = R'[f';..b'] \land \neg c'$$
$$\lor \quad m \land f<b \land \neg m' \land f'>b' \land b-f = n+b'-f' \land R[f;..b] = R'[(f';..n); (0;..b')] \land \neg c'$$
$$\lor \quad \neg m \land f>b \land m' \land f'<b' \land n+b-f = b'-f' \land R[(f;..n); (0;..b)] = R'[f';..b'] \land \neg c'$$
$$\lor \quad \neg m \land f>b \land \neg m' \land f'>b' \land b-f = b'-f' \land R[(f;..n); (0;..b)] = R'[(f';..n); (0;..b')] \land \neg c'$$
$$\lor \quad m \land f=b \land m' \land f'=b' \land c'$$
$$\lor \quad \neg m \land f=b \land \neg m' \land f'=b' \land R[(f;..n); (0;..b)]=R'[(f';..n); (0;..b')] \land \neg c'$$
$$\Leftarrow \quad c' = (m \land f=b) \land f'=f \land b'=b \land R'=R$$
$$= \quad c:= m \land f=b$$

The transformed operation offered us the opportunity to rotate the queue within R , but we declined to do so. For other data structures, it is sometimes a good strategy to reorganize the data structure during an operation, and data transformation always tells us what reorganizations are possible. Each of the remaining transformations offers the same opportunity, but there is no reason to rotate the queue, and we decline each time.

Next we transform $c:= isfullq$, *join* x , and *leave* .

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land c'=(p=n) \land p'=p \land Q'=Q \qquad \text{several omitted steps}$$
$$\Leftarrow \quad c:= \neg m \land f=b$$

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land Q'=Q[0;..p]+[x]+Q[p+1;..n] \land p'=p+1 \qquad \text{several omitted steps}$$
$$\Leftarrow \quad Rb:= x.\ \textbf{if}\ b+1=n\ \textbf{then}\ (b:= 0.\ m:= \bot)\ \textbf{else}\ b:= b+1$$

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land Q'=Q[(1;..p); (p-1;..n)] \land p'=p-1 \qquad \text{several omitted steps}$$
$$\Leftarrow \quad \textbf{if}\ f+1=n\ \textbf{then}\ (f:= 0.\ m:= \top)\ \textbf{else}\ f:= f+1$$

Last we transform $x:= front$ where x is a user's variable of the same type as the items.

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land x'=Q0 \land p'=p \land Q'=Q \qquad \text{several omitted steps}$$
$$\Leftarrow \quad x:= R f$$

End of Limited Queue

7.2.4 Soundness and Completeness

optional

Data transformation is sound in the sense that a user cannot tell that a transformation has been made; that was the criterion of its design. But it is possible to find two specifications of identical behavior (from a user's point of view) for which there is no data transformer to transform one into the other. In that sense, data transformation is incomplete.

Exercise 374 illustrates the problem. The user's variable is i and the implementer's variable is j, both of type nat. The operations are:

$initialize = i' = 0 \le j' < 3$

$step = \textbf{if } j>0 \textbf{ then } (i:= i+1.\ j:= j-1) \textbf{ else } ok$

The user can look at i but not at j. The user can $initialize$, which starts i at 0 and starts j at any of 3 values. The user can then repeatedly $step$ and observe that i increases 0 or 1 or 2 times and then stops increasing, which effectively tells the user what value j started with.

If this were a practical problem, we would notice that $initialize$ can be refined, resolving the nondeterminism. For example,

$initialize \Leftarrow i:= 0.\ j:= 0$

We could then transform $initialize$ and $step$ to get rid of j, replacing it with nothing. The transformer is $j=0$. It transforms the implementation of $initialize$ as follows:

$\forall j\cdot j=0 \Rightarrow \exists j'\cdot j'=0 \wedge i'=j'=0$

$= \quad i:= 0$

And it transforms $step$ as follows:

$\forall j\cdot j=0 \Rightarrow \exists j'\cdot j'=0 \ \wedge \ \textbf{if } j>0 \textbf{ then } (i:= i+1.\ j:= j-1) \textbf{ else } ok$

$= \quad ok$

If this were a practical problem, we would be done. But the theoretical problem is to replace j with boolean variable b without resolving the nondeterminism, so that

$initialize$ is transformed to $i'=0$

$step$ is transformed to $\textbf{if } b \wedge i<2 \textbf{ then } i' = i+1 \textbf{ else } ok$

Now the transformed $initialize$ starts b either at $\top$, meaning that i will be increased, or at $\bot$, meaning that i will not be increased. Each use of the transformed $step$ tests b to see if we might increase i, and checks $i<2$ to ensure that i will remain below 3. If i is increased, b is again assigned either of its two values. The user will see i start at 0 and increase 0 or 1 or 2 times and then stop increasing, exactly as in the original specification. The nondeterminism is maintained. But there is no transformer in variables i, j, and b to do the job.

End of Soundness and Completeness

End of Data Transformation

End of Theory Design and Implementation

8 Concurrency

Concurrency, also known as parallelism, means two or more activities occurring at the same time. In some other books, the words "concurrency" and "parallelism" are used to mean that the activities occur in an unspecified sequence, or that they are composed of smaller activities that occur in an interleaved sequence. But in this book they mean that there is more than one activity at a time.

8.0 Independent Composition

We define the independent composition of specifications P and Q so that $P||Q$ (pronounced " P parallel Q ") is satisfied by a computer that behaves according to P and, at the same time, in parallel, according to Q. The operands of $||$ are called processes.

When we defined the dependent composition of P and Q, we required that P and Q have exactly the same state variables, so that we could identify the final state of P with the initial state of Q. For independent composition $P||Q$, we require that P and Q have completely different state variables, and the state variables of the composition $P||Q$ are those of both P and Q. If we ignore time and space, independent composition is conjunction.

$$P||Q \;=\; P\wedge Q$$

When we decide to create an independent composition, we decide how to partition the variables. Given specification S, if we choose to refine it as $S \Leftarrow P||Q$, we have to decide which variables of S belong to P, and which to Q. For example, in variables x, y, and z, the specification

$$x' = x+1 \;\wedge\; y' = y+2 \;\wedge\; z'=z$$

can be refined by the independent composition

$$x:= x+1 \;||\; y:= y+2$$

if we partition the variables. Clearly x has to belong to the left process for the assignment to x to make sense, and similarly y has to belong to the right process. As for z, it doesn't matter which process we give it to; either way

$$x:= x+1 \;||\; y:= y+2 \;\;=\;\; x' = x+1 \;\wedge\; y' = y+2 \;\wedge\; z'=z$$

The person who introduces the independent composition is responsible for deciding how to partition the variables. If we are presented with an independent composition, and the person who wrote it failed to record the partitioning, we have to determine a partitioning that makes sense. Here's a way that usually works: If either x' or $x:=$ appears in a process specification, then x belongs to that process. If neither x' nor $x:=$ appears at all, then x can be placed on either side of the partition. This way of partitioning does not work when x' or $x:=$ appears in both process specifications.

In the next example

$$x:= y \;||\; y:= x$$

again x belongs to the left process, y to the right process, and z to either process. In the left process, y appears, but neither y' nor $y:=$ appears, so y is a state constant, not a state variable, in the left process. Similarly x is a state constant in the right process. And the result is

$$x:= y \;||\; y:= x \;\;=\;\; x'=y \wedge y'=x \wedge z'=z$$

Variables x and y swap values, apparently without a temporary variable. In fact, an implementation of a process will have to make a private copy of the initial value of a variable belonging to the other process if the other process contains an assignment to that variable.

In boolean variable b and integer variable x ,

$$b{:=}\ x{=}x \ \|\ x{:=}\ x{+}1 \qquad \text{replace } x{=}x \text{ by } \top$$
$$=\quad b{:=}\ \top \ \|\ x{:=}\ x{+}1$$

On the first line, it may seem possible for the process on the right side to increase x between the two evaluations of x in the left process, resulting in the assignment of $\bot$ to b . And that would be a mathematical disaster; we could not even be sure $x{=}x$. According to the last line, this does not happen; both occurrences of x in the left process refer to the initial value of variable x . We can use the reflexive and transparent axioms of equality, and replace $x{=}x$ by $\top$.

In a dependent composition as defined in Chapter 4, the intermediate values of variables are local to the dependent composition; they are hidden by the quantifier $\exists x'', y'', \cdots$. If one process is a dependent composition, the other cannot see its intermediate values. For example,

$$(x{:=}\ x{+}1.\ x{:=}\ x{-}1)\ \|\ y{:=}\ x$$
$$=\quad ok \,\|\, y{:=}\ x$$
$$=\quad y{:=}\ x$$

On the first line, it may seem possible for the process on the right side to evaluate x between the two assignments to x in the left process. According to the last line, this does not happen; the occurrence of x in the right process refers to the initial value of variable x . In the next chapter we introduce interactive variables and communication channels between processes so they can see the intermediate values of each other's variables, but in this chapter processes are not able to interact.

In the previous example, we replaced $(x{:=}\ x{+}1.\, x{:=}\ x{-}1)$ by ok . And of course we can make the reverse replacement whenever x is one of the state variables. Although x is one of the variables of the composition

$$ok \,\|\, x{:=}\ 3$$

it is not one of the variables of the left process ok due to the assignment in the right process. So we cannot equate that composition to

$$(x{:=}\ x{+}1.\ x{:=}\ x{-}1)\ \|\ x{:=}\ 3$$

Sometimes the need for shared memory arises from poor program structure. For example, suppose we decide to have two processes, as follows.

$$(x{:=}\ x{+}y.\ x{:=}\ x{\times}y)$$
$$\|\quad (y{:=}\ x{-}y.\ y{:=}\ x/y)$$

The first modifies x twice, and the second modifies y twice. But suppose we want the second assignment in each process to use the values of x and y after the first assignments of both processes. This may seem to require not only a shared memory, but also synchronization of the two processes at their mid-points, forcing the faster process to wait for the slower one, and then to allow the two processes to continue with the new, updated values of x and y . Actually, it requires neither shared memory nor synchronization devices. It is achieved by writing

$$(x{:=}\ x{+}y\ \|\ y{:=}\ x{-}y).\ (x{:=}\ x{\times}y\ \|\ y{:=}\ x/y)$$

So far, independent composition is just conjunction, and there is no need to introduce a second symbol $\|$ for conjunction. But now we consider time. The time variable is not subject to partitioning; it belongs to both processes. In $P\|Q$, both P and Q begin execution at time t , but their executions may finish at different times. Execution of the composition $P\|Q$ finishes when both P and Q are finished. With time, independent composition is defined as

$$P\|Q \;=\; \exists t_P, t_Q\cdot\ \langle t'{\rightarrow}P\rangle t_P \ \wedge\ \langle t'{\rightarrow}Q\rangle t_Q \ \wedge\ t' = max\ t_P\ t_Q$$
$$=\; \exists t_P, t_Q\cdot\quad (\text{substitute } t_P \text{ for } t' \text{ in } P)$$
$$\wedge\ (\text{substitute } t_Q \text{ for } t' \text{ in } Q)$$
$$\wedge\ t' = max\ t_P\ t_Q$$

8.0.0 Laws of Independent Composition

Let x and y be different state variables, let e , f , and b be expressions of the prestate, and let P , Q , R , and S be specifications. Then

$(x{:=}\, e \parallel y{:=}f).\ P \;=\;$ (for x substitute e and independently for y substitute f in P) — independent substitution

$P \parallel Q \;=\; Q \parallel P$ — symmetry

$P \parallel (Q \parallel R) \;=\; (P \parallel Q) \parallel R$ — associativity

$P \parallel ok \;=\; ok \parallel P \;=\; P$ — identity

$P \parallel Q{\vee}R \;=\; (P \parallel Q) \vee (P \parallel R)$ — distributivity

$P \parallel$ **if** b **then** Q **else** $R \;=\;$ **if** b **then** $(P \parallel Q)$ **else** $(P \parallel R)$ — distributivity

if b **then** $(P\|Q)$ **else** $(R\|S) \;=\;$ **if** b **then** P **else** $R \parallel$ **if** b **then** Q **else** S — distributivity

The Associative Law says we can compose any number of processes without worrying how they are grouped. As an example of the Substitution Law,

$$(x{:=}\, x{+}y \parallel y{:=}\, x{\times}y).\ z' = x{-}y \;=\; z' = (x{+}y) - (x{\times}y)$$

Note that each substitution replaces all and only the original occurrences of its variable. This law generalizes the earlier Substitution Law from one variable to two, and it can be generalized further to any number of variables.

Refinement by Steps works for independent composition:

If $A \Leftarrow B\|C$ and $B \Leftarrow D$ and $C \Leftarrow E$ are theorems, then $A \Leftarrow D\|E$ is a theorem.

So does Refinement by Parts:

If $A \Leftarrow B\|C$ and $D \Leftarrow E\|F$ are theorems, then $A{\wedge}D \Leftarrow B{\wedge}E \parallel C{\wedge}F$ is a theorem.

End of Laws of Independent Composition

8.0.1 List Concurrency

We have defined independent composition by partitioning the variables. For finer-grained concurrency, we can extend this same idea to the individual items within list variables. In Chapter 5 we defined assignment to a list item as

$$Li{:=}\, e \;=\; L'i{=}e \;\wedge\; (\forall j\cdot\ j{\ne}i \Rightarrow L'j{=}Lj) \;\wedge\; x'{=}x \;\wedge\; y'{=}y \;\wedge\; ...$$

which says not only that the assigned item has the right final value, but also that all other items and all other variables do not change value. For independent composition, we must specify the final values of only the items and variables in one side of the partition.

As a good example of list concurrency, we do Exercise 140: find the maximum item in a list. The maximum of a list is easily expressed with the MAX quantifier, but we will assume MAX is not implemented. The easiest and simplest solution is probably functional, with parallelism coming from the fact that the arguments of a function (operands of an operator) can always be evaluated in parallel. To use our parallel operator, we present an imperative solution. Let L be the list whose maximum item is sought. If L is an empty list, its maximum is $-\infty$; assume that L is nonempty. Assume further that L is a variable whose value is not wanted after we know its maximum (we'll remove this assumption later). Our specification will be $L'\ 0 = MAX\ L$; at the end, item 0 of list L will be the maximum of all original items. The first step is to generalize from the maximum of a nonempty list to the maximum of a nonempty segment of a list. So define

$$findmax \;=\; \langle i, j \rightarrow i{<}j \Rightarrow L'\ i = MAX\ L\ [i;..j]\rangle$$

Our specification is $findmax\ 0\ (\#L)$. We refine as follows.

$$findmax\ i\ j \ \Leftarrow\ \textbf{if}\ j{-}i = 1\ \textbf{then}\ ok\ \textbf{else}\ (\ (findmax\ i\ (div\ (i{+}j)\ 2) \ \|\ findmax\ (div\ (i{+}j)\ 2)\ j).\ L\ i := max\ (L\ i)\ (L\ (div\ (i{+}j)\ 2))\ )$$

If $j{-}i = 1$ the segment contains one item; to place the maximum item (the only item) at index i requires no change. In the other case, the segment contains more than one item; we divide the segment into two halves, placing the maximum of each half at the beginning of the half. In the parallel composition, the two processes $findmax\ i\ (div\ (i{+}j)\ 2)$ and $findmax\ (div\ (i{+}j)\ 2)\ j$ change disjoint segments of the list. We finish by placing the maximum of the two maximums at the start of the whole segment. The recursive execution time is $ceil\ (log\ (j{-}i))$, exactly the same as for binary search, which this program closely resembles.

If list L must remain constant, we can use a new list M of the same type as L to collect our partial results. We redefine

$$findmax\ =\ \langle i, j\rightarrow i{<}j \Rightarrow M'\ i = MAX\ L\ [i;..j]\rangle$$

and in the program we change ok to $M\ i := L\ i$ and we change the final assignment to

$$M\ i := max\ (M\ i)\ (M\ (div\ (i{+}j)\ 2))$$

End of List Concurrency

End of Independent Composition

8.1 Sequential to Parallel Transformation

The goal of this section is to transform programs without concurrency into programs with concurrency. A simple example illustrates the idea. Ignoring time,

$$x:= y.\ x:= x{+}1.\ z:= y$$
$$=\quad x:= y.\ (x:= x{+}1\ \|\ z:= y)$$
$$=\quad (x:= y.\ x:= x{+}1)\ \|\ z:= y$$

Execution of the program on the first line can be depicted as follows.

start ⟶ $x:= y$ ⟶ $x:= x{+}1$ ⟶ $z:= y$ ⟶ finish

The first two assignments cannot be executed concurrently, but the last two can, so we transform the program. Execution can now be depicted as

start ⟶ $x:= y$ ⟶ ($x:= x{+}1$ and $z:= y$) ⟶ finish

Now we have the first and last assignments next to each other, in sequence; they too can be executed concurrently. Execution can be

start ⟶ ($x:= y$ ⟶ $x:= x{+}1$ and $z:= y$) ⟶ finish

Whenever two programs occur in sequence, and neither assigns to any variable assigned in the other, and no variable assigned in the first appears in the second, they can be placed in parallel; a copy must be made of the initial value of any variable appearing in the first and assigned in the second. Whenever two programs occur in sequence, and neither assigns to any variable appearing

in the other, they can be placed in parallel without any copying of initial values. This transformation does not change the result of a computation, but it may decrease the time, and that is the reason for doing it.

Program transformation to obtain concurrency can often be performed automatically by the implementation. Sometimes it can only be performed by the implementation because the result is not expressible as a source program.

8.1.0 Buffer

Consider two programs, *produce* and *consume*, whose only common variable is b. *produce* assigns to b and *consume* uses the value of b.

$$produce \;=\; \cdots\cdots\cdots b{:=}\, e\cdots\cdots\cdots$$
$$consume \;=\; \cdots\cdots\cdots x{:=}\, b\cdots\cdots\cdots$$

These two programs are executed alternately, repeatedly, forever.

$$control \;=\; produce.\ consume.\ control$$

Using P for *produce* and C for *consume*, execution looks like this:

$$P \longrightarrow C \longrightarrow P \longrightarrow C \longrightarrow P \longrightarrow C \longrightarrow P \longrightarrow C \longrightarrow$$

Many programs have producer and consumer components somewhere in them. Variable b is called a buffer; it may be a large data structure. The idea is that *produce* and *consume* are time-consuming, and we can save time if we put them in parallel. As they are, we cannot put them in parallel because the first assigns to b and the second uses b. So we unroll the loop once.

$$control \;=\; produce.\ newcontrol$$
$$newcontrol \;=\; consume.\ produce.\ newcontrol$$

and *newcontrol* can be transformed to

$$newcontrol \;=\; (consume \,\|\, produce).\ newcontrol$$

In this transformed program, the implementation of *consume* will have to capture a copy of the initial value of b. Or, we could do this capture at source level by splitting b into two variables, p and c, as follows.

$$produce \;=\; \cdots\cdots\cdots p{:=}\, e\cdots\cdots\cdots$$
$$consume \;=\; \cdots\cdots\cdots x{:=}\, c\cdots\cdots\cdots$$
$$control \;=\; produce.\ newcontrol$$
$$newcontrol \;=\; c{:=}\, p.\ (consume \,\|\, produce).\ newcontrol$$

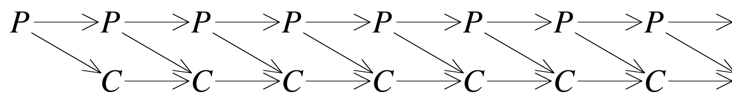
Using B for the assignment $c{:=}\, p$, execution is

P ↘ B ↗ P ↘ B ↗ P ↘ B ↗ P ↘ B ↗ P ↘ B ↗ P ↘ B ↗ P ↘ B ↗
B ↘ C ↗ B ↘ C ↗ B ↘ C ↗ B ↘ C ↗ B ↘ C ↗ B ↘ C ↗ B ↘ C ↗

If one of *produce* or *consume* consistently takes longer than the other, this is the best that can be done. If their execution times vary so that in some cycles *produce* takes longer while in others *consume* takes longer, we can improve by splitting the buffer into an infinite list. We need natural variable w to indicate how much *produce* has written into the buffer, and natural variable r to indicate how much *consume* has read from the buffer. We initialize both w and r to 0. Then

$$produce \;=\; \cdots\cdots\cdots bw{:=}\, e.\ w{:=}\, w{+}1\cdots\cdots\cdots$$
$$consume \;=\; \cdots\cdots\cdots x{:=}\, br.\ r{:=}\, r{+}1\cdots\cdots\cdots$$
$$control \;=\; produce.\ consume.\ control$$

If $w \neq r$ then *produce* and *consume* can be executed in parallel, as follows.



When the execution of *produce* is fast, it can get arbitrarily far ahead of the execution of *consume*. When the execution of *consume* is fast, it can catch up to *produce* but not pass it; the sequence is retained when $w=r$. The opportunity for parallel execution can be found automatically by the programming language implementation (compiler), or it can be told to the implementation in some suitable notation. But, in this example, the resulting execution pattern is not expressible as a source program without additional interactive constructs (Chapter 9).

If the buffer is a finite list of length n, we can use it in a cyclic fashion with this modification:

$produce \;=\; \cdots\cdots\; bw:= e.\;\; w:= mod\;(w+1)\;n \cdots\cdots$
$consume \;=\; \cdots\cdots\; x:= br.\;\; r:= mod\;(r+1)\;n \cdots\cdots$
$control \;=\; produce.\;\; consume.\;\; control$

As before, *consume* cannot overtake *produce* because $w=r$ when the buffer is empty. But now *produce* cannot get more than n executions ahead of *consume* because $w=r$ also when the buffer is full.

End of Buffer

Programs are sometimes easier to develop and prove when they do not include any mention of concurrency. The burden of finding concurrency can be placed upon a clever implementation. Synchronization is what remains of sequential execution after all opportunities for concurrency have been found.

8.1.1 Insertion Sort

Exercise 169 asks for a program to sort a list in time bounded by the square of the length of the list. Here is a solution. Let the list be L, and define

$sort \;=\; \langle n \rightarrow \forall i, j: 0,..n \cdot i \le j \Rightarrow Li \le Lj \rangle$

so that $sort\; n$ says that L is sorted up to index n. The specification is

$(\, L' \text{ is a permutation of } L\,) \wedge sort'\;(\#L) \wedge t' \le t + (\#L)^2$

We leave the first conjunct informal, and ensure that it is satisfied by making all changes to L using

$swap\; i\; j \;=\; Li:= Lj \,||\, Lj:= Li$

We ignore the last conjunct; program transformation will give us a linear time solution. The second conjunct is equal to $sort\; 0 \Rightarrow sort'\;(\#L)$ since $sort\; 0$ is a theorem.

$sort\; 0 \Rightarrow sort'\;(\#L) \;\Leftarrow\; \textbf{for}\; n:= 0;..\#L\; \textbf{do}\; sort\; n \Rightarrow sort'\;(n+1)$

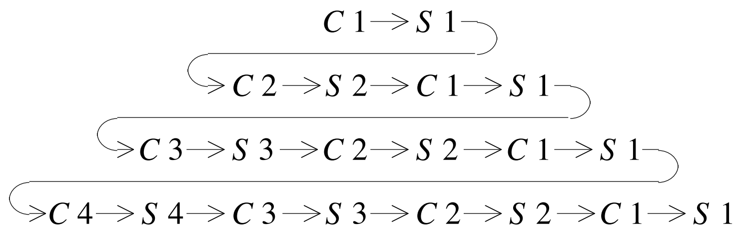
To solve $sort\; n \Rightarrow sort'\;(n+1)$, it may help to refer to an example list.

$$\underset{0}{[}\;\; L0 \;\; \underset{1}{;} \;\; L1 \;\; \underset{2}{;} \;\; L2 \;\; \underset{3}{;} \;\; L3 \;\; \underset{4}{;} \;\; L4 \;\; \underset{5}{]}$$

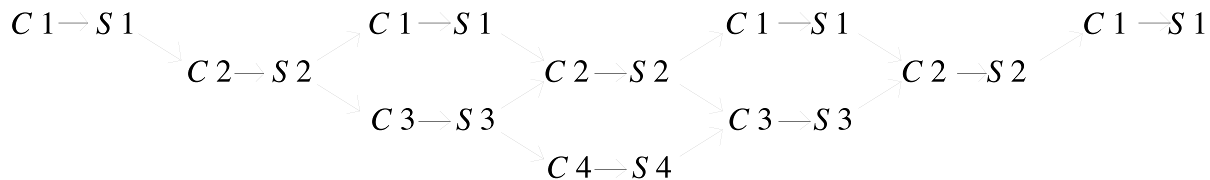
$sort\; n \Rightarrow sort'\;(n+1) \;\Leftarrow\;$ **if** $n=0$ **then** ok
else if $L\,(n-1) \le L\; n$ **then** ok
else $(swap\;(n-1)\;n.\;\; sort\;(n-1) \Rightarrow sort'\; n)$

If we consider $sort\ n \Rightarrow sort'\ (n+1)$ to be a procedure with parameter n we are finished; the final specification $sort\ (n-1) \Rightarrow sort'\ n$ calls the same procedure with argument $n-1$. Or, we could let n be a variable instead of a **for**-loop index, and decrease it by 1 just before the final call. We leave this detail, and move on to the possibilities for parallel execution.

Let $C\ n$ stand for the comparison $L\ (n-1) \le L\ n$ and let $S\ n$ stand for $swap\ (n-1)\ n$. For $\#L=5$, the worst case sequential execution is shown in the following picture.



If i and j differ by more than 1 , then $S\ i$ and $S\ j$ can be executed concurrently. Under the same condition, $S\ i$ can be executed and $C\ j$ can be evaluated concurrently. And of course, any two expressions such as $C\ i$ and $C\ j$ can always be evaluated concurrently. Execution becomes



For the ease of writing a quadratic-time sequential sort, given a clever implementation, we obtain a linear-time parallel sort.

End of Insertion Sort

8.1.2 Dining Philosophers

Exercise 384: Five philosophers are sitting around a round table. At the center of the table is an infinite bowl of noodles. Between each pair of neighboring philosophers is a chopstick. Whenever a philosopher gets hungry, the hungry philosopher reaches for the chopstick on the left and the chopstick on the right, because it takes two chopsticks to eat. If either chopstick is unavailable because the neighboring philosopher is using it, then this hungry philosopher will have to wait until it is available again. When both chopsticks are available, the philosopher eats for a while, then puts down the chopsticks, and goes back to thinking, until the philosopher gets hungry again. The problem is to write a program whose execution simulates the life of these philosophers. It may happen that all five philosophers get hungry at the same time, they each pick up their left chopstick, they then notice that their right chopstick isn't there, and they each decide to wait for their right chopstick while holding on to their left chopstick. That's a deadlock, and the program must be written so that doesn't happen. If we write the program so that only one philosopher gets hungry at a time, there won't be any deadlock, but there won't be much concurrency either.

This problem is a standard one, used in many textbooks, to illustrate the problems of concurrency in programming. There is often one more criterion: each philosopher eats infinitely many times. But we won't bother with that. We'll start with the one-at-a-time version in which there is no concurrency and no deadlock. Number the philosophers from 0 through 4 going round the

table. Likewise number the chopsticks so that the two chopsticks for philosopher i are numbered i and $i+1$ (all additions in this exercise are modulo 5).

$$life \;=\; (P\,0 \lor P\,1 \lor P\,2 \lor P\,3 \lor P\,4).\; life$$
$$P\,i \;=\; up\,i.\; up(i+1).\; eat\,i.\; down\,i.\; down(i+1)$$
$$up\,i \;=\; chopstick\,i:=\top$$
$$down\,i \;=\; chopstick\,i:=\bot$$
$$eat\,i \;=\; \cdots\cdots chopstick\,i\cdots\cdots chopstick(i+1)\cdots\cdots$$

These definitions say that life is a completely arbitrary sequence of $P\,i$ actions (choose any one, then repeat), where a $P\,i$ action says that philosopher i picks up the left chopstick, then picks up the right chopstick, then eats, then puts down the left chopstick, then puts down the right chopstick. For these definitions to become a program, we need to decide how to make the choice among the $P\,i$ each iteration; or perhaps we can leave it to the implementation to make the choice (this is where the criterion that each philosopher eats infinitely often would be met). It is unclear how to define $eat\,i$, except that it uses two chopsticks. (If this program were intended to accomplish some purpose, we could eliminate variable $chopstick$, replacing both occurrences in $eat\,i$ by $\top$. But the program is intended to describe an activity, and eating makes use of two chopsticks.)

Now we transform to get concurrency.

If $i\ne j$, $(up\,i.\, up\,j)$ becomes $(up\,i \,\|\, up\,j)$.
If $i\ne j$, $(up\,i.\, down\,j)$ becomes $(up\,i \,\|\, down\,j)$.
If $i\ne j$, $(down\,i.\, up\,j)$ becomes $(down\,i \,\|\, up\,j)$.
If $i\ne j$, $(down\,i.\, down\,j)$ becomes $(down\,i \,\|\, down\,j)$.
If $i\ne j \land i+1\ne j$, $(eat\,i.\, up\,j)$ becomes $(eat\,i \,\|\, up\,j)$.
If $i\ne j \land i\ne j+1$, $(up\,i.\, eat\,j)$ becomes $(up\,i \,\|\, eat\,j)$.
If $i\ne j \land i+1\ne j$, $(eat\,i.\, down\,j)$ becomes $(eat\,i \,\|\, down\,j)$.
If $i\ne j \land i\ne j+1$, $(down\,i.\, eat\,j)$ becomes $(down\,i \,\|\, eat\,j)$.
If $i\ne j \land i+1\ne j \land i\ne j+1$, $(eat\,i.\, eat\,j)$ becomes $(eat\,i \,\|\, eat\,j)$.

Different chopsticks can be picked up or put down at the same time. Eating can be in parallel with picking up or putting down a chopstick, as long as it isn't one of the chopsticks being used for the eating. And finally, two philosophers can eat at the same time as long as they are not neighbors. All these transformations are immediately seen from the definitions of up , $down$, eat , and independent composition. They are not all immediately applicable to the original program, but whenever a transformation is made, it may enable further transformations.

Before any transformation, there is no possibility of deadlock. No transformation introduces the possibility. The result is the maximum concurrency that does not lead to deadlock. A clever implementation can take the initial program (without concurrency) and make the transformations.

A mistake often made in solving the problem of the dining philosophers is to start with too much concurrency.

$$life \;=\; P\,0 \,\|\, P\,1 \,\|\, P\,2 \,\|\, P\,3 \,\|\, P\,4$$
$$P\,i \;=\; (up\,i \,\|\, up(i+1)).\; eat\,i.\; (down\,i \,\|\, down(i+1)).\; P\,i$$

Clearly $P\,0$ cannot be placed in parallel with $P\,1$ because they both assign and use $chopstick\,1$. Those who start this way must then try to correct the error by adding mutual exclusion devices and deadlock avoidance devices, and that is what makes the problem hard. It is better not to make the error; then the mutual exclusion devices and deadlock avoidance devices are not needed.

End of Dining Philosophers

End of Sequential to Parallel Transformation

End of Concurrency

9 Interaction

We have been describing computation according to the initial values and final values of state variables. A state variable declaration

$$\mathbf{var}\ x{:}\ T\cdot S \ = \ \exists x, x'{:}\ T\cdot S$$

says that a state variable is really two mathematical variables, one for the initial value and one for the final value. Within the scope of the declaration, x and x' are available for use in specification S. There are intermediate values whenever there is a dependent (sequential) composition, but these intermediate values are local to the definition of dependent composition.

$$P.\,Q \ = \ \exists x'', y'', ...\cdot \ \langle x', y', ... \rightarrow P\rangle\, x''\ y''\ ... \ \land \ \langle x, y, ... \rightarrow Q\rangle\, x''\ y''\ ...$$

Consider $(P.\,Q) \parallel R$. The intermediate values between P and Q are hidden in the dependent composition, and are not visible to R, so they cannot be used for process interaction.

A variable whose value is visible only initially and finally is called a boundary variable, and a variable whose value is visible all the time is called an interactive variable. So far our variables have all been boundary variables. Now we introduce interactive variables whose intermediate values are visible to parallel processes. These variables can be used to describe and reason about interactions between people and computers, and between processes, during the course of a computation.

9.0 Interactive Variables

Let the notation **ivar** $x{:}\ T\cdot S$ declare x to be an interactive variable of type T and scope S. It is defined as follows.

$$\mathbf{ivar}\ x{:}\ T\cdot S \ = \ \exists x{:}\ time \rightarrow T\cdot S$$

where $time$ is the domain of time, usually either the extended integers or the extended reals. An interactive variable is a function of time. The value of variable x at time t is $x\,t$.

Suppose a and b are boundary variables, x and y are interactive variables, and t is time. For independent composition we partition all the state variables, boundary and interactive. Suppose a and x belong to P, and b and y belong to Q.

$$\begin{aligned} P\|Q \ = \ \exists t_P, t_Q\cdot \ & \langle t' \rightarrow P\rangle\, t_P \ \land \ (\forall t''\cdot\ t_P \le t'' \le t' \Rightarrow x t'' = x(t_P)) \\ \land\ & \langle t' \rightarrow Q\rangle\, t_Q \ \land \ (\forall t''\cdot\ t_Q \le t'' \le t' \Rightarrow y t'' = y(t_Q)) \\ \land\ & t' = max\ t_P\ t_Q \end{aligned}$$

The new part says that when the shorter process is finished, its interactive variables remain unchanged while the longer process is finishing.

Using the same processes and variables as in the previous paragraph, the assignment $x{:=}\ a+b+x+y$ in process P assigns to variable x the sum of four values. Since a and x are variables of process P, their values are the latest ones assigned to them by process P, or their initial values if process P has not assigned to them. Since b is a boundary variable of process Q, its value, as seen in P, is its initial value, regardless of whether Q has assigned to it. Since y is an interactive variable of process Q, its value, as seen in P, is the latest one assigned to it by process Q, or its initial value if Q has not assigned to it, or unknown if Q is in the middle of assigning to it. Since x is an interactive variable, its new value can be seen in all parallel processes. The expression $a+b+x+y$ is an abuse of notation, since a and b are numbers and x and y are functions from time to numbers; the value being assigned is actually $a+b+xt+yt$, but we omit the argument t when the context makes it clear. We will similarly write x' to mean xt', and x'' to mean xt''.

The definition of ok says that the boundary variables and time are unchanged. So in process P of the previous two paragraphs,

$$ok \;=\; a'=a \;\land\; t'=t$$

There is no need to say $x'=x$, which means $xt'=xt$, since $t'=t$. We do not mention b and y because they are not variables of process P .

Assignment to an interactive variable cannot be instantaneous because it is time that distinguishes its values. In a process where the boundary variables are a and b , and the interactive variables are x and y ,

$$x{:=}\,e \;=\; a'=a \;\land\; b'=b \;\land\; x'=e \;\land\; (\forall t''\cdot\ t\le t''\le t' \Rightarrow y''=y)$$
$$\land\;\; t' = t+(\text{the time required to evaluate and store } e\,)$$

interactive variable y remains unchanged throughout the duration of the assignment to x . Nothing is said about the value of x during the assignment.

Assignment to a boundary variable can be instantaneous if we wish. If we choose to account for its time, we must say that all interactive variables remain unchanged during the assignment.

Dependent composition hides the intermediate values of the boundary and time variables, leaving the intermediate values of the interactive variables visible. In boundary variables a and b , and interactive variables x and y , and time t , we define

$$P.\,Q \;=\; \exists a'',b'',t''\cdot\ \langle a',b',t'\to P\rangle\, a''\ b''\ t'' \;\land\; \langle a,b,t\to Q\rangle\, a''\ b''\ t''$$

Most of the specification laws and refinement laws survive the addition of interactive variables, but sadly, the Substitution Law no longer works.

If processes P and Q are in parallel, they have different variables. Suppose again that boundary variable a and interactive variable x are the variables of process P , and that boundary variable b and interactive variable y are the variables of process Q . In specification P , the inputs are a , b , xt , and yt'' for $t\le t''<t'$. In specification P , the outputs are a' , and xt'' for $t<t''\le t'$. Specification P is implementable when

$$\forall a,b,X,y,t\cdot\ \exists a',x,t'\cdot\ P \;\land\; t\le t' \;\land\; \forall t''\cdot\ t<t''\le t' \;\lor\; x\,t''=X\,t''$$

As before, P must be satisfiable with nondecreasing time; the new part says that P must not constrain its interactive variables outside the interval from t to t' . We do not need to know the context of a process specification to check its implementability; variables b and y appear only in the outside universal quantification.

Exercise 385 is an example in the same variables a , b , x , y , and t . Suppose that time is an extended integer, and that each assignment takes time 1 .

$(x{:=}\,2.\ x{:=}\,x+y.\ x{:=}\,x+y) \,\|\, (y{:=}\,3.\ y{:=}\,x+y)$ Clearly, x is a variable in the left process and y is a variable in the right process. Let's put a in the left process and b in the right process.

$=\;\; (a'=a \land xt'=2 \land t'=t+1.\ a'=a \land xt'= xt+yt \land t'=t+1.\ a'=a \land xt'= xt+yt \land t'=t+1)$
$\;\;\;\| \; (b'=b \land yt'=3 \land t'=t+1.\ b'=b \land yt'= xt+yt \land t'=t+1)$

$=\;\; (a'=a \land x(t+1)=2 \land x(t+2)= x(t+1)+y(t+1) \land x(t+3)= x(t+2)+y(t+2) \land t'=t+3)$
$\;\;\;\| \; (b'=b \land y(t+1)=3 \land y(t+2)= x(t+1)+y(t+1) \land t'=t+2)$

$=\;\; a'=a \land x(t+1)=2 \land x(t+2)= x(t+1)+y(t+1) \land x(t+3)= x(t+2)+y(t+2)$
$\;\;\;\land\; b'=b \land y(t+1)=3 \land y(t+2)= x(t+1)+y(t+1) \land y(t+3)=y(t+2) \land t'=t+3$

$=\;\; a'=a \land x(t+1)=2 \land x(t+2)=5 \land x(t+3)=10$
$\;\;\;\land\; b'=b \land y(t+1)=3 \land y(t+2)=y(t+3)=5 \land t'=t+3$

The example gives the appearance of lock-step synchrony only because we took each assignment time to be 1. More realistically, different assignments take different times, perhaps specified nondeterministically with lower and upper bounds. Whatever timing policy we decide on, whether deterministic or nondeterministic, whether discrete or continuous, the definitions and theory remain unchanged. Of course, complicated timing leads quickly to very complicated expressions that describe all possible interactions. If we want to know only something, not everything, about the possible behaviors, we can proceed by implications instead of equations, weakening for the purpose of simplifying. Programming goes the other way: we start with a specification of desired behavior, and strengthen as necessary to obtain a program.

9.0.0 Thermostat

Exercise 388: specify a thermostat for a gas burner. The thermostat operates in parallel with other processes

$$thermometer \,||\, control \,||\, thermostat \,||\, burner$$

The thermometer and the control are typically located together, but they are logically distinct. The inputs to the thermostat are:

- real *temperature* , which comes from the thermometer and indicates the actual temperature.
- real *desired* , which comes from the control and indicates the desired temperature.
- boolean *flame* , which comes from a flame sensor in the burner and indicates whether there is a flame.

These three variables must be interactive variables because their values may be changed at any time by another process and the thermostat must react to their current values. These three variables do not belong to the thermostat, and cannot be assigned values by the thermostat. The outputs of the thermostat are:

- boolean *gas* ; assigning it $\top$ turns the gas on and $\bot$ turns the gas off.
- boolean *spark* ; assigning it $\top$ causes sparks for the purpose of igniting the gas.

Variables *gas* and *spark* belong to the thermostat process. They must also be interactive variables; the burner needs their current values.

Heat is wanted when the actual temperature falls ε below the desired temperature, and not wanted when the actual temperature rises ε above the desired temperature, where ε is small enough to be unnoticeable, but large enough to prevent rapid oscillation. To obtain heat, the spark should be applied to the gas for at least 1 second to give it a chance to ignite and to allow the flame to become stable. But a safety regulation states that the gas must not remain on and unlit for more than 3 seconds. Another regulation says that when the gas is shut off, it must not be turned on again for at least 20 seconds to allow any accumulated gas to clear. And finally, the gas burner must respond to its inputs within 1 second.

Here is a specification:

$$thermostat \;=\; (gas{:=}\,\bot \,||\, spark{:=}\,\bot).\;\; GasIsOff$$

$$\begin{aligned} GasIsOff \;=\; &\textbf{if } temperature < desired - \varepsilon \\ &\textbf{then } ((gas{:=}\,\top \,||\, spark{:=}\,\top \,||\, t{+}1 \le t' \le t{+}3).\;\; spark{:=}\,\bot.\;\; GasIsOn) \\ &\textbf{else } (((\textbf{frame } gas, spark\cdot\; ok) \,||\, t < t' \le t{+}1).\;\; GasIsOff) \end{aligned}$$

$$\begin{aligned} GasIsOn \;=\; &\textbf{if } temperature < desired + \varepsilon \;\wedge\; flame \\ &\textbf{then } (((\textbf{frame } gas, spark\cdot\; ok) \,||\, t < t' \le t{+}1).\;\; GasIsOn) \\ &\textbf{else } ((gas{:=}\,\bot \,||\, (\textbf{frame } spark\cdot\; ok) \,||\, t{+}20 \le t' \le t{+}21).\;\; GasIsOff) \end{aligned}$$

We are using the time variable to represent real time in seconds. The specification $t+1 \le t' \le t+3$ represents the passage of at least 1 second but not more than 3 seconds. The specification $t+20 \le t' \le t+21$ is similar. A specification that a computation be slow enough is always easy to satisfy. A specification that it be fast enough requires us to build fast enough hardware; in this case it is easy since instruction times are microseconds and the time bounds are seconds.

One can always argue about whether a formal specification captures the intent of an informal specification. For example, if the gas is off, and heat becomes wanted, and the ignition sequence begins, and then heat is no longer wanted, this last input may not be noticed for up to 3 seconds. It may be argued that this is not responding to an input within 1 second, or it may be argued that the entire ignition sequence is the response to the first input, and until its completion no response to further inputs is required. At least the formal specification is unambiguous.

End of Thermostat

9.0.1 Space

The main purpose of interactive variables is to provide a means for processes to interact. In this subsection, we show another use. We make the space variable s into an interactive variable in order to look at the space occupied during the course of a computation. As an example, Exercise 389 is contrived to be as simple as possible while including time and space calculations in an infinite computation.

Suppose *alloc* allocates 1 unit of memory space and takes time 1 to do so. Then the following computation slowly allocates memory.

$$GrowSlow \Leftarrow \textbf{if } t=2\times x \textbf{ then } (alloc \parallel x:= t) \textbf{ else } t:= t+1.\ \ GrowSlow$$

If the time is equal to $2\times x$, then one space is allocated, and in parallel x becomes the time stamp of the allocation; otherwise the clock ticks. The process is repeated forever. Prove that if the space is initially less than the logarithm of the time, and x is suitably initialized, then at all times the space is less than the logarithm of the time.

It is not clear what initialization is suitable for x, so leaving that aside for a moment, we define *GrowSlow* to be the desired specification.

$$GrowSlow = s < log\ t \ \Rightarrow\ (\forall t''\cdot\ t''\ge t \ \Rightarrow\ s'' < log\ t'')$$

where s is an interactive variable, so s is really $s\ t$ and s'' is really $s\ t''$. We are just interested in the space calculation and not in actually allocating space, so we can take *alloc* to be $s:= s+1$. There is no need for x to be interactive, so let's make it a boundary variable. To make the proof easier, we let all variables be extended naturals, although the result we are proving holds also for real time.

Now we have to prove the refinement, and to do that it helps to break it into pieces. The body of the loop can be written as a disjunction.

$$\begin{aligned} & \textbf{if } t=2\times x \textbf{ then } (s:= s+1 \parallel x:= t) \textbf{ else } t:= t+1 \\ = \quad & t=2\times x \land s'=s+1 \land x'=t \land t'=t+1 \ \lor\ t\ne 2\times x \land s'=s \land x'=x \land t'=t+1 \end{aligned}$$

Now the refinement has the form

$$\begin{aligned} & (A\Rightarrow B \ \Leftarrow\ C\lor D.\ A\Rightarrow B) && \text{. distributes over } \lor \\ = \quad & (A\Rightarrow B \ \Leftarrow\ (C.\ A\Rightarrow B) \lor (D.\ A\Rightarrow B)) && \text{antidistributive law} \\ = \quad & (A\Rightarrow B \ \Leftarrow\ (C.\ A\Rightarrow B)) \land (A\Rightarrow B \ \Leftarrow\ (D.\ A\Rightarrow B)) && \text{portation twice} \\ = \quad & (B \ \Leftarrow\ A \land (C.\ A\Rightarrow B)) \land (B \ \Leftarrow\ A \land (D.\ A\Rightarrow B)) \end{aligned}$$

So we can break the proof into two cases:

$$B \Leftarrow A \wedge (C.\ A \Rightarrow B)$$
$$B \Leftarrow A \wedge (D.\ A \Rightarrow B)$$

starting each time with the right side (antecedent) and working toward the left side (consequent). First case:

$$\begin{array}{ll}
& s < log\ t \ \wedge\ (\ t=2\times x \wedge s'=s+1 \wedge x'=t \wedge t'=t+1. \\
& \qquad s < log\ t \ \Rightarrow\ \forall t''\cdot\ t''\ge t \ \Rightarrow\ s'' < log\ t'') \\
& \qquad\qquad \text{remove dependent composition, remembering that } s \text{ is interactive} \\
= & s < log\ t \ \wedge\ (\exists x'', t'''\cdot\ \ t=2\times x \wedge s'''=s+1 \wedge x''=t \wedge t'''=t+1 \\
& \qquad \wedge\ (s''' < log\ t''' \ \Rightarrow\ \forall t''\cdot\ t''\ge t''' \ \Rightarrow\ s'' < log\ t'')) \\
& \qquad\qquad \text{Use } s'''=s+1 \text{ and drop it. Use one-point to eliminate } \exists x'', t''' . \\
\Rightarrow & s < log\ t \ \wedge\ t=2\times x \ \wedge\ (s+1 < log(t+1) \ \Rightarrow\ \forall t''\cdot\ t''\ge t+1 \Rightarrow\ s'' < log\ t'')
\end{array}$$

The next step should be discharge. We need

$$\begin{array}{ll}
& s < log\ t \ \wedge\ t=2\times x \ \Rightarrow\ s+1 < log(t+1) \\
= & 2^s < t = 2\times x \ \Rightarrow\ 2^{s+1} < t+1 \\
= & 2^s < t = 2\times x \ \Rightarrow\ 2^{s+1} \le t \\
= & 2^s < t = 2\times x \ \Rightarrow\ 2^{s+1} \le 2\times x \\
= & 2^s < t = 2\times x \ \Rightarrow\ 2^s \le x \\
\Leftarrow & 2^s \le x
\end{array}$$

This is the missing initialization of x . So we go back and redefine $GrowSlow$.

$$GrowSlow \ = \ s < log\ t \ \wedge\ x\ge 2^s \ \Rightarrow\ (\forall t''\cdot\ t''\ge t \ \Rightarrow\ s'' < log\ t'')$$

Now we redo the proof. First case:

$$\begin{array}{ll}
& s < log\ t \ \wedge\ x\ge 2^s \ \wedge\ (\ t=2\times x \wedge s'=s+1 \wedge x'=t \wedge t'=t+1. \\
& \qquad s < log\ t \ \wedge\ x\ge 2^s \ \Rightarrow\ \forall t''\cdot\ t''\ge t \ \Rightarrow\ s'' < log\ t'') \\
& \qquad\qquad \text{remove dependent composition, remembering that } s \text{ is interactive} \\
= & \quad s < log\ t \ \wedge\ x\ge 2^s \\
& \wedge\ (\exists x'', t'''\cdot\ \ t=2\times x \wedge s'''=s+1 \wedge x''=t \wedge t'''=t+1 \\
& \qquad \wedge\ (s''' < log\ t''' \ \wedge\ x''\ge 2^{s'''} \ \Rightarrow\ \forall t''\cdot\ t''\ge t''' \ \Rightarrow\ s'' < log\ t'')) \\
& \qquad\qquad \text{Use } s'''=s+1 \text{ and drop it. Use one-point to eliminate } \exists x'', t''' . \\
\Rightarrow & \quad s < log\ t \ \wedge\ x\ge 2^s \ \wedge\ t=2\times x \\
& \wedge\ (s+1 < log(t+1) \ \wedge\ t\ge 2^{s+1} \ \Rightarrow\ \forall t''\cdot\ t''\ge t+1 \Rightarrow\ s'' < log\ t'') \\
& \qquad\qquad \text{discharge, as calculated earlier} \\
= & s < log\ t \ \wedge\ x\ge 2^s \ \wedge\ t=2\times x \ \wedge\ \forall t''\cdot\ t''\ge t+1 \Rightarrow\ s'' < log\ t'' \\
& \quad \text{when } t''=t \text{, then } s''=s \text{ and since } s < log\ t \text{, the domain of } t'' \text{ can be increased} \\
\Rightarrow & \forall t''\cdot\ t''\ge t \ \Rightarrow\ s'' < log\ t''
\end{array}$$

The second case is easier than the first.

$$\begin{array}{ll}
& s < log\ t \ \wedge\ x\ge 2^s \ \wedge\ (\ t\ne 2\times x \wedge s'=s \wedge x'=x \wedge t'=t+1. \\
& \qquad s < log\ t \ \wedge\ x\ge 2^s \ \Rightarrow\ \forall t''\cdot\ t''\ge t \ \Rightarrow\ s'' < log\ t'') \\
& \qquad\qquad \text{remove dependent composition, remembering that } s \text{ is interactive} \\
= & \quad s < log\ t \ \wedge\ x\ge 2^s \\
& \wedge\ (\exists x'', t'''\cdot\ \ t\ne 2\times x \wedge s'''=s \wedge x''=x \wedge t'''=t+1 \\
& \qquad \wedge\ (s''' < log\ t''' \ \wedge\ x''\ge 2^{s'''} \ \Rightarrow\ \forall t''\cdot\ t''\ge t''' \ \Rightarrow\ s'' < log\ t'')) \\
& \qquad\qquad \text{Use } s'''=s \text{ and drop it. Use one-point to eliminate } \exists x'', t''' . \\
\Rightarrow & \quad s < log\ t \ \wedge\ x\ge 2^s \ \wedge\ t\ne 2\times x \\
& \wedge\ (s < log\ t \ \wedge\ x\ge 2^s \ \Rightarrow\ \forall t''\cdot\ t''\ge t+1 \Rightarrow\ s'' < log\ t'') \qquad \text{discharge} \\
= & s < log\ t \ \wedge\ x\ge 2^s \ \wedge\ t\ne 2\times x \ \wedge\ \forall t''\cdot\ t''\ge t+1 \Rightarrow\ s'' < log\ t'' \\
& \quad \text{when } t''=t \text{, then } s''=s \text{ and since } s < log\ t \text{, the domain of } t'' \text{ can be increased} \\
\Rightarrow & \forall t''\cdot\ t''\ge t \ \Rightarrow\ s'' < log\ t''
\end{array}$$

End of Space

End of Interactive Variables

A shared variable is a variable that can be written and read by any process. Shared variables are popular for process interaction, but they present enormous problems for people who wish to reason about their programs, and for those who must build the hardware and software to implement them. For their trouble, there is no benefit. Interactive variables are not fully shared; all processes can read an interactive variable, but only one process can write it. Interactive variables are easier to reason about and implement than fully shared variables. Even boundary variables are shared a little: their initial values are visible to all processes. They are easiest to reason about and implement, but they provide the least interaction.

Although interactive variables are tamer than shared variables, there are still two problems with them. The first is that they provide too much information. Usually, a process does not need the values of all interactive variables at all times; each process needs only something about the values (an expression in interactive variables), and only at certain times. The other problem is that processes may be executed on different processors, and the rates of execution may not be identical. This makes it hard to know exactly when to read the value of an interactive variable; it certainly should not be read while its owner process is in the middle of writing to it.

We now turn to a form of communication between processes that does not have these problems: it provides just the right information, and mediates the timing between the processes. And, paradoxically, it provides the means for fully sharing variables safely.

9.1 Communication

This section introduces named communication channels through which a computation communicates with its environment, which may be people or other computations running in parallel. For each channel, only one process (person or computation) writes to it, but all processes can read all the messages, each at its own speed. For two-way communication, use two channels. We start the section by considering only one reading process, which may be the same process that writes, or may be a different process. We consider multiple reading processes later when we come to Subsection 9.1.9 on broadcast.

Communication on channel c is described by two infinite strings $\mathbb{M}c$ and $\mathbb{T}c$ called the message script and the time script, and two extended natural variables rc and wc called the read cursor and the write cursor. The message script is the string of all messages, past, present, and future, that pass along the channel. The time script is the corresponding string of times that the messages were or are or will be sent. The scripts are state constants, not state variables. The read cursor is a state variable saying how many messages have been read, or input, on the channel. The write cursor is a state variable saying how many messages have been written, or output, on the channel. If there is only one channel, or if the channel is known from context, we may leave out the channel name, abbreviating the names of the scripts and cursors to $\mathbb{M}$, $\mathbb{T}$, w, and r.

During execution, the read and write cursors increase as inputs and outputs occur; more and more of the script items are seen, but the scripts do not vary. At any time, the future messages and the times they are sent on a channel may be unknown, but they can be referred to as items in the scripts. For example, after 2 more reads the next input on channel c will be $\mathbb{M}c_{rc+2}$, and after 5 more writes the next output will be $\mathbb{M}c_{wc+5}$ and it will occur at time $\mathbb{T}c_{wc+5}$. Omitting the channel name from the script and cursor names, after 2 more reads the next input will be $\mathbb{M}_{r+2}$, and after 5 more writes the next output will be $\mathbb{M}_{w+5}$ at time $\mathbb{T}_{w+5}$.

$$\begin{array}{lllllllllllll} \mathbb{M} & = & 6 & ;\ 4 & ;\ 7 & ;\ 1 & ;\ 0 & ;\ 3 & ;\ 8 & ;\ 9 & ;\ 2 & ;\ 5 & ;\ ... \\ \mathbb{T} & = & 3 & ;\ 5 & ;\ 5 & ;\ 20 & ;\ 25 & ;\ 28 & ;\ 31 & ;\ 31 & ;\ 45 & ;\ 48 & ;\ ... \\ & & & & & \uparrow & & \uparrow \\ & & & & & r & & w \end{array}$$

The scripts and the cursors are not programming notations, but they allow us to specify any desired communications. Here is an example specification. It says that if the next input on channel c is even, then the next output on channel d will be $\top$, and otherwise it will be $\bot$. Formally, we may write

$$\textbf{if } even\ (\mathbb{M}c_{rc})\ \textbf{then } \mathbb{M}d_{wd} = \top\ \textbf{else } \mathbb{M}d_{wd} = \bot$$

or, more briefly,

$$\mathbb{M}d_{wd}\ =\ even\ (\mathbb{M}c_{rc})$$

If there are only a finite number of communications on a channel, then after the last message, the time script items are all ∞, and the message script items are of no interest.

9.1.0 Implementability

Consider computations involving two memory variables x and y, a time variable t, and communications on a single channel. The state of a computation consists of the values of the memory variables, the time variable, and the cursor variables. During a computation, the memory variables can change value in any direction, but time and the cursors can only increase. Once an input has been read, it cannot be unread; once an output has been written, it cannot be unwritten. Every computation satisfies

$$t'{\ge}t\ \ \land\ \ r'{\ge}r\ \ \land\ \ w'{\ge}w$$

An implementable specification can say what the scripts are in the segment written by a computation, that is the segment $\mathbb{M}_{w;..w'}$ and $\mathbb{T}_{w;..w'}$ between the initial and final values of the write cursor, but it cannot specify the scripts outside this segment. Furthermore, the time script must be monotonic, and all its values in this segment must be in the range from t to t'.

A specification S (in initial state σ, final state σ', message script $\mathbb{M}$, and time script $\mathbb{T}$) is implementable if and only if

$$\begin{array}{ll} \forall\sigma, \mathbb{M}'', \mathbb{T}''\cdot\ \exists\sigma', \mathbb{M}, \mathbb{T}\cdot & S\ \land\ t' \ge t\ \land\ r' \ge r\ \land\ w' \ge w \\ & \land\ \mathbb{M}_{(0;..w);\ (w';..\infty)} = \mathbb{M}''_{(0;..w);\ (w';..\infty)} \\ & \land\ \mathbb{T}_{(0;..w);\ (w';..\infty)} = \mathbb{T}''_{(0;..w);\ (w';..\infty)} \\ & \land\ \forall i, j\colon w,..w'\cdot\ i{\le}j \Rightarrow t \le \mathbb{T}_i \le \mathbb{T}_j \le t' \end{array}$$

If we have many channels, we need similar conjuncts for each. If we have no channels, implementability reduces to the definition given in Chapter 4.

To implement communication channels, it is not necessary to build two infinite strings. At any given time, only those messages that have been written and not yet read need to be stored. The time script is only for specification and proof, and does not need to be stored at all.

End of Implementability

9.1.1 Input and Output

Here are five programming notations for communication. Let c be a channel. The notation $c!\ e$ describes a computation that writes the output message e on channel c. The notation $c!$ describes a computation that sends a signal on channel c (no message; the act of signalling is the only information). The notation $c?$ describes a computation that reads one input on channel c. We use the channel name c to denote the message that was last previously read on the channel. And $\surd c$ is a boolean expression meaning "there is unread input available on channel c". Here are the formal definitions.

$$c!\ e \;=\; \mathbb{M}_w = e \;\land\; \mathbb{T}_w = t \;\land\; (w{:=}\ w+1) \qquad \text{“} c \text{ output } e \text{”}$$
$$c! \;=\; \mathbb{T}_w = t \;\land\; (w{:=}\ w+1) \qquad \text{“} c \text{ signal”}$$
$$c? \;=\; r{:=}\ r+1 \qquad \text{“} c \text{ input”}$$
$$c \;=\; \mathbb{M}_{r-1}$$
$$\surd c \;=\; \mathbb{T}_r \le t \qquad \text{“check } c \text{”}$$

Suppose the input channel from a keyboard is named key, and the output channel to a screen is named $screen$. Then execution of the program

if $\surd key$
then (key?. **if** key="y" **then** $screen$! "If you wish." **else** $screen$! "Not if you don't want.")
else $screen$! "Well?"

tests if a character of input is available, and if so, reads it and prints some output, which depends on the character read, and if not, prints other output.

Let us refine the specification $\mathbb{M}d_{wd} = even\ (\mathbb{M}c_{rc})$ given earlier.

$$\mathbb{M}d_{wd} = even\ (\mathbb{M}c_{rc}) \;\Leftarrow\; c?.\ d!\ even\ c$$

To prove the refinement, we can rewrite the solution as follows:

$$c?.\ d!\ even\ c$$
$$=\; rc{:=}\ rc+1.\ \mathbb{M}d_{wd} = even\ (\mathbb{M}c_{rc-1}) \;\land\; \mathbb{T}d_{wd} = t \;\land\; (wd{:=}\ wd+1)$$
$$=\; \mathbb{M}d_{wd} = even\ (\mathbb{M}c_{rc}) \;\land\; \mathbb{T}d_{wd} = t \;\land\; rc'=rc+1 \;\land\; wc'=wc \;\land\; rd'=rd \;\land\; wd'=wd+1$$

which implies the problem.

A problem specification should be written as clearly, as understandably, as possible. A programmer refines the problem specification to obtain a solution program, which a computer can execute. In our example, the solution seems more understandable than the problem! Whenever that is the case, we should consider using the program as the problem specification, and then there is no need for refinement.

Our next problem is to read numbers from channel c, and write their doubles on channel d. Ignoring time, the specification can be written

$$S \;=\; \forall n{:}\ nat\cdot\ \mathbb{M}d_{wd+n} = 2 \times \mathbb{M}c_{rc+n}$$

We cannot assume that the input and output are the first input and output ever on channels c and d. We can only ask that from now on, starting at the initial read cursor rc and initial write cursor wd, the outputs will be double the inputs. This specification can be refined as follows.

$$S \;\Leftarrow\; c?.\ d!\ 2{\times}c.\ S$$

The proof is:

$$c?.\ d!\ 2{\times}c.\ S$$
$$=\; rc{:=}\ rc+1.\ \mathbb{M}d_{wd} = 2 \times \mathbb{M}c_{rc-1} \;\land\; (wd{:=}\ wd+1).\ S$$
$$=\; \mathbb{M}d_{wd} = 2 \times \mathbb{M}c_{rc} \;\land\; \forall n{:}\ nat\cdot\ \mathbb{M}d_{wd+1+n} = 2 \times \mathbb{M}c_{rc+1+n}$$
$$=\; \forall n{:}\ nat\cdot\ \mathbb{M}d_{wd+n} = 2 \times \mathbb{M}c_{rc+n}$$
$$=\; S$$

End of Input and Output

9.1.2 Communication Timing

In the real time measure, we need to know how long output takes, how long communication transit takes, and how long input takes, and we place time increments appropriately. To be independent of these implementation details, we can use the transit time measure, in which we suppose that the acts of input and output take no time at all, and that communication transit takes 1 time unit.

The message to be read next on channel c is $\mathbb{M}c_{rc}$. This message was or is or will be sent at time $\mathbb{T}c_{rc}$. Its arrival time, according to the transit time measure, is $\mathbb{T}c_{rc}+1$. So input becomes

$$t:= max\ t\ (\mathbb{T}c_{rc}+1).\ \ c?$$

If the input has already arrived, $\mathbb{T}c_{rc}+1 \le t$, and no time is spent waiting for input; otherwise execution of $c?$ is delayed until the input arrives. And the input check $\surd c$ becomes

$$\surd c\ \ =\ \ \mathbb{T}c_{rc}+1 \le t$$

In some applications (called "batch processing"), all inputs are available at the start of execution; for these applications, we may as well leave out the time assignments for input, and we have no need for the input check. In other applications (called "process control"), inputs are provided at regular intervals by a physical sampling device; the time script (but not the message script) is known in advance. In still other applications (called "interactive computing"), a human provides inputs at irregular intervals, and we have no way of saying what the time script is. In this case, we have to leave out the waiting times, and just attach a note to our calculation saying that execution time will be increased by any time spent waiting for input.

Exercise 407(a): Let W be "wait for input on channel c and then read it". Formally,

$$W\ \ =\ \ t:= max\ t\ (\mathbb{T}_r+1).\ \ c?$$

Prove $W \ \ \Leftarrow\ \ \mathbf{if}\ \surd c\ \mathbf{then}\ c?\ \mathbf{else}\ (t:= t+1.\ \ W)$ assuming time is an extended integer. The significance of this exercise is that input is often implemented in just this way, with a test to see if input is available, and a loop if it is not. Proof:

$$\begin{aligned}
&\mathbf{if}\ \surd c\ \mathbf{then}\ c?\ \mathbf{else}\ (t:= t+1.\ \ W) && \text{replace } \surd c \text{ and } W\\
=\ \ &\mathbf{if}\ \mathbb{T}_r+1 \le t\ \mathbf{then}\ c?\ \mathbf{else}\ (t:= t+1.\ \ t:= max\ t\ (\mathbb{T}_r+1).\ \ c?)\\
=\ \ &\mathbf{if}\ \mathbb{T}_r+1 \le t\ \mathbf{then}\ (t:= t.\ \ c?)\ \mathbf{else}\ (t:= max\ (t+1)\ (\mathbb{T}_r+1).\ \ c?)\\
& && \text{If } \mathbb{T}_r+1 \le t \text{, then } t = max\ t\ (\mathbb{T}_r+1)\ .\\
& && \text{If } \mathbb{T}_r+1 > t \text{ then } max\ (t+1)\ (\mathbb{T}_r+1)\ =\ \mathbb{T}_r+1\ =\ max\ t\ (\mathbb{T}_r+1)\ .\\
=\ \ &\mathbf{if}\ \mathbb{T}_r+1 \le t\ \mathbf{then}\ (t:= max\ t\ (\mathbb{T}_r+1).\ \ c?)\ \mathbf{else}\ (t:= max\ t\ (\mathbb{T}_r+1).\ \ c?)\\
=\ \ &W
\end{aligned}$$

End of Communication Timing

9.1.3 Recursive Communication

optional; requires Chapter 6

Define dbl by the fixed-point construction (including recursive time but ignoring input waits)

$$dbl\ \ =\ \ c?.\ \ d!\ 2\times c.\ \ t:= t+1.\ \ dbl$$

Regarding dbl as the unknown, this equation has several solutions. The weakest is

$$\forall n\colon nat\cdot\ \mathbb{M}d_{wd+n} = 2\times\mathbb{M}c_{rc+n}\ \ \wedge\ \ \mathbb{T}d_{wd+n} = t+n$$

A strongest implementable solution is

$$\begin{aligned}
&(\forall n\colon nat\cdot\ \mathbb{M}d_{wd+n} = 2\times\mathbb{M}c_{rc+n}\ \ \wedge\ \ \mathbb{T}d_{wd+n} = t+n)\\
\wedge\ \ &rc'=wd'=t'=\infty\ \ \wedge\ \ wc'=wc\ \ \wedge\ \ rd'=rd
\end{aligned}$$

The strongest solution is $\bot$. If this fixed-point construction is all we know about dbl , then we cannot say that it is equal to a particular one of the solutions. But we can say this: it refines the weakest solution

$$\forall n\colon nat\cdot\ \mathbb{M}d_{wd+n} = 2\times\mathbb{M}c_{rc+n}\ \ \wedge\ \ \mathbb{T}d_{wd+n} = t+n\ \ \ \Leftarrow\ \ \ dbl$$

and it is refined by the right side of the fixed-point construction

$$dbl \Leftarrow c?.\ d!\ 2{\times}c.\ t:= t+1.\ dbl$$

Thus we can use it to solve problems, and we can execute it.

If we begin recursive construction with

$$dbl_0 = \top$$

we find

$$\begin{aligned} dbl_1 &= c?.\ d!\ 2{\times}c.\ t:= t+1.\ dbl_0 \\ &= rc:= rc+1.\ \mathbb{M}d_{wd} = 2 \times \mathbb{M}c_{rc-1} \ \land\ \mathbb{T}d_{wd} = t \ \land (wd:= wd+1).\ t:= t+1.\ \top \\ &= \mathbb{M}d_{wd} = 2 \times \mathbb{M}c_{rc} \ \land\ \mathbb{T}d_{wd} = t \\ dbl_2 &= c?.\ d!\ 2{\times}c.\ t:= t+1.\ dbl_1 \\ &= rc:= rc+1.\ \mathbb{M}d_{wd} = 2 \times \mathbb{M}c_{rc-1} \ \land\ \mathbb{T}d_{wd} = t \ \land (wd:= wd+1). \\ &\quad\ t:= t+1.\ \mathbb{M}d_{wd} = 2 \times \mathbb{M}c_{rc} \ \land\ \mathbb{T}d_{wd} = t \\ &= \mathbb{M}d_{wd} = 2 \times \mathbb{M}c_{rc} \ \land\ \mathbb{T}d_{wd} = t \ \land\ \mathbb{M}d_{wd+1} = 2{\times}\mathbb{M}c_{rc+1} \ \land\ \mathbb{T}d_{wd+1} = t+1 \end{aligned}$$

and so on. The result of the construction

$$dbl_\infty = \forall n: nat\cdot \mathbb{M}d_{wd+n} = 2 \times \mathbb{M}c_{rc+n} \ \land\ \mathbb{T}d_{wd+n} = t+n$$

is the weakest solution of the dbl fixed-point construction. If we begin recursive construction with $t'{\ge}t \ \land\ rc'{\ge}rc \ \land\ wc'{\ge}wc \ \land\ rd'{\ge}rd \ \land\ wd'{\ge}wd$ we get a strongest implementable solution.

End of Recursive Communication

9.1.4 Merge

Merging means reading repeatedly from two or more input channels and writing those inputs onto another channel. The output is an interleaving of the messages from the input channels. The output must be all and only the messages read from the inputs, and it must preserve the order in which they were read on each channel. Infinite merging can be specified formally as follows. Let the input channels be c and d, and the output channel be e. Then

$$merge = (c?.\ e!\ c) \lor (d?.\ e!\ d).\ merge$$

This specification does not state any criterion for choosing between the input channels at each step. To write a merge program, we must decide on a criterion for choosing. We might choose between the input channels based on the value of the inputs or on their arrival times.

Exercise 411(a) (time merge) asks us to choose the first available input at each step. If input is already available on both channels c and d, take either one; if input is available on just one channel, take that one; if input is available on neither channel, wait for the first one and take it (in case of a tie, take either one). Here is the specification.

$$\begin{aligned} timemerge = &\quad (\surd c \ \lor\ \mathbb{T}c_{rc} \le \mathbb{T}d_{rd}) \ \land\ (c?.\ e!\ c) \\ &\lor\ (\surd d \ \lor\ \mathbb{T}c_{rc} \ge \mathbb{T}d_{rd}) \ \land\ (d?.\ e!\ d). \\ &\ timemerge \end{aligned}$$

To account for the time spent waiting for input, we should insert $t:= max\ t\ (\mathbb{T}_r + 1)$ just before each input operation, and for recursive time we should insert $t:= t+1$ before the recursive call.

In Subsection 9.1.2 on Communication Timing we proved that waiting for input can be implemented recursively. Using the same reasoning, we implement $timemerge$ as follows.

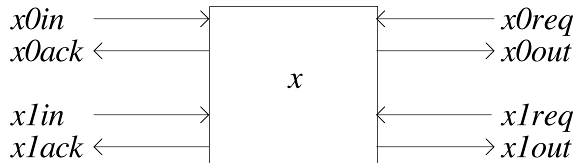
$$\begin{aligned} timemerge \Leftarrow\ & \textbf{if}\ \surd c\ \textbf{then}\ (c?.\ e!\ c)\ \textbf{else}\ ok. \\ & \textbf{if}\ \surd d\ \textbf{then}\ (d?.\ e!\ d)\ \textbf{else}\ ok. \\ & t:= t+1.\ timemerge \end{aligned}$$

assuming time is an extended integer.

End of Merge

9.1.5 Monitor

To obtain the effect of a fully shared variable, we create a process called a monitor that resolves conflicting uses of the variable. A monitor for variable x receives on channels $x0in$, $x1in$, ... data from other processes to be written to the variable, whereupon it sends an acknowledgement back to the writing process on one of the channels $x0ack$, $x1ack$, ... It receives on channels $x0req$, $x1req$, ... requests from other processes to read the variable, whereupon it sends the value of the variable back to the requesting process on one of the channels $x0out$, $x1out$, ...



A monitor for variable x with two writing processes and two reading processes can be defined as follows. Let m be the minimum of the times $\mathbb{T}x0in_{rx0in}$, $\mathbb{T}x1in_{rx1in}$, $\mathbb{T}x0req_{rx0req}$, and $\mathbb{T}x1req_{rx1req}$ of the next input on each of the input channels. Then

$$\begin{aligned} monitor \;=\; & \quad (\surd x0in \;\vee\; \mathbb{T}x0in_{rx0in} = m) \;\wedge\; (x0in?.\;\; x{:=}\, x0in.\;\; x0ack!) \\ & \vee\;\; (\surd x1in \;\vee\; \mathbb{T}x1in_{rx1in} = m) \;\wedge\; (x1in?.\;\; x{:=}\, x1in.\;\; x1ack!) \\ & \vee\;\; (\surd x0req \;\vee\; \mathbb{T}x0req_{rx0req} = m) \;\wedge\; (x0req?.\;\; x0out!\, x) \\ & \vee\;\; (\surd x1req \;\vee\; \mathbb{T}x1req_{rx1req} = m) \;\wedge\; (x1req?.\;\; x1out!\, x). \\ & monitor \end{aligned}$$

Just like $timemerge$, a monitor takes the first available input and responds to it. A monitor for several variables, for several writing processes, and for several reading processes, is similar. When more than one input is available, an implementation must make a choice. Here's one way to implement a monitor, assuming time is an extended integer:

$$\begin{aligned} monitor \;\Leftarrow\; & \textbf{if } \surd x0in \textbf{ then } (x0in?.\;\; x{:=}\, x0in.\;\; x0ack!) \textbf{ else } ok. \\ & \textbf{if } \surd x1in \textbf{ then } (x1in?.\;\; x{:=}\, x1in.\;\; x1ack!) \textbf{ else } ok. \\ & \textbf{if } \surd x0req \textbf{ then } (x0req?.\;\; x0out!\, x) \textbf{ else } ok. \\ & \textbf{if } \surd x1req \textbf{ then } (x1req?.\;\; x1out!\, x) \textbf{ else } ok. \\ & t{:=}\, t{+}1.\;\; monitor \end{aligned}$$

We earlier solved Exercise 388 to specify a thermostat for a gas burner using interactive variables gas , $temperature$, $desired$, $flame$, and $spark$, as follows.

$$thermostat \;=\; (gas{:=}\, \bot \,\|\, spark{:=}\, \bot).\;\; GasIsOff$$

$$\begin{aligned} GasIsOff \;=\; & \textbf{if } temperature < desired - \varepsilon \\ & \textbf{then } ((gas{:=}\, \top \,\|\, spark{:=}\, \top \,\|\, t{+}1 \le t' \le t{+}3).\;\; spark{:=}\, \bot.\;\; GasIsOn) \\ & \textbf{else } (((\textbf{frame } gas, spark\cdot\, ok) \,\|\, t < t' \le t{+}1).\;\; GasIsOff) \end{aligned}$$

$$\begin{aligned} GasIsOn \;=\; & \textbf{if } temperature < desired + \varepsilon \;\wedge\; flame \\ & \textbf{then } (((\textbf{frame } gas, spark\cdot\, ok) \,\|\, t < t' \le t{+}1).\;\; GasIsOn) \\ & \textbf{else } ((gas{:=}\, \bot \,\|\, (\textbf{frame } spark\cdot\, ok) \,\|\, t{+}20 \le t' \le t{+}21).\;\; GasIsOff) \end{aligned}$$

If we use communication channels instead of interactive variables, we have to build a monitor for these variables, and rewrite our thermostat specification. Here is the result.

$thermostat$ = $((gasin!\ \bot.\ gasack?) \| (sparkin!\ \bot.\ sparkack?)).\ GasIsOff$

$GasIsOff$ = $((temperaturereq!.\ temperature?) \| (desiredreq!.\ desired?)).$
if $temperature < desired - \varepsilon$
then $(\ ((gasin!\ \top.\ gasack?) \| (sparkin!\ \top.\ sparkack?) \| t{+}1 \le t' \le t{+}3).$
$sparkin!\ \bot.\ sparkack?.\ GasIsOn\ )$
else $(t < t' \le t{+}1.\ GasIsOff)$

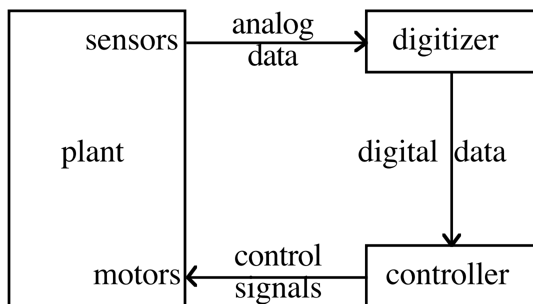
$GasIsOn$ = $(\ (temperaturereq!.\ temperature?) \| (desiredreq!.\ desired?)$
$\| (flamereq!.\ flame?)\ ).$
if $temperature < desired + \varepsilon\ \wedge\ flame$
then $(t < t' \le t{+}1.\ GasIsOn)$
else $(((gasin!\ \bot.\ gasack?) \| t{+}20 \le t' \le t{+}21).\ GasIsOff)$

End of Monitor

The calculation of space requirements when there is concurrency may sometimes require a monitor for the space variable, so that any process can request an update, and the updates can be communicated to all processes. The monitor for the space variable is also the arbiter between competing space allocation requests.

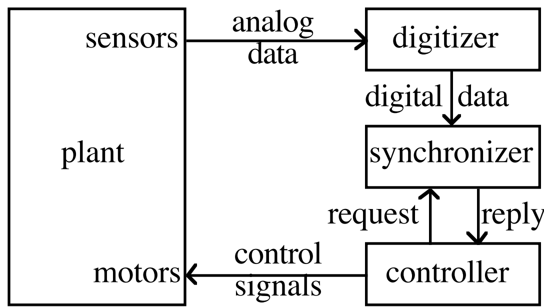
9.1.6 Reaction Controller

Many kinds of reactions are controlled by a feedback loop, as shown in the following picture.



The "plant" could be a chemical reactor, or a nuclear reactor, or even just an assembly plant. The sensors detect concentrations or temperatures or positions in the form of analog data, and feed them to a digitizer. The digitizer converts these data to digital form suitable for the controller. The controller computes what should happen next to control the plant; perhaps some rods should be pushed in farther, or some valves should be opened, or a robot arm should move in some direction. The controller sends signals to the plant to cause the appropriate change.

Here's the problem. The sensors send their data continuously to the digitizer. The digitizer is fast and uniform, sending digital data rapidly to the controller. The time required by the controller to compute its output signals varies according to the input messages; sometimes the computation is trivial and it can keep up with the input; sometimes the computation is more complex and it falls behind. When several inputs have piled up, the controller should not continue to read them and compute outputs in the hope of catching up. Instead, we want all but the latest input to be discarded. It is not essential that control signals be produced as rapidly as digital data. But it is essential that each control signal be based on the latest available data. How can we achieve this? The solution is to place a synchronizer between the digitizer and controller, as in the following picture.



The synchronizer's job is as simple and uniform as the digitizer's; it can easily keep up. It repeatedly reads the data from the digitizer, always keeping only the latest. Whenever the controller requests some data, the synchronizer sends the latest. This is exactly the function of a monitor, and we could implement the synchronizer that way. But a synchronizer is simpler than a monitor in two respects: first, there is only one writing process and one reading process; second, the writing process is uniformly faster than the reading process. Here is its definition.

synchronizer $=$ *digitaldata*?.
if √*request* **then** (*request*? || *reply*! *digitaldata*) **else** *ok*.
synchronizer

If we were using interactive variables instead of channels, there would be no problem of reading old data; reading an interactive variable always reads its latest value, even if the variable is written more often than it is read. But there would be the problem of how to make sure that the interactive variable is not read while it is being written.

End of Reaction Controller

9.1.7 Channel Declaration

The next input on a channel is not necessarily the one that was last previously written on that channel. In one variable x and one channel c (ignoring time),

$c!\ 2.\ c?.\ x:= c$

$= \quad \mathbb{M}_w = 2 \ \land\ w' = w+1 \ \land\ r' = r+1 \ \land\ x' = \mathbb{M}_r$

We do not know that initially $w=r$, so we cannot conclude that finally $x'=2$. That's because there may have been a previous write that hasn't been read yet. For example,

$c!\ 1.\ c!\ 2.\ c?.\ x:= c$

The next input on a channel is always the first one on that channel that has not yet been read. The same is true in a parallel composition.

$c!\ 2\ \|\ (c?.\ x:= c)$

$= \quad \mathbb{M}_w = 2 \ \land\ w' = w+1 \ \land\ r' = r+1 \ \land\ x' = \mathbb{M}_r$

Again we cannot say $x'=2$ because there may be a previous unread output

$c!\ 1.\ (c!\ 2\ \|\ (c?.\ x:= c)).\ c?$

and the final value of x may be the 1 from the earlier output, with the 2 going to the later input. In order to achieve useful communication between processes, we have to introduce a local channel.

Channel declaration is similar to variable declaration; it defines a new channel within some local portion of a program or specification. A channel declaration applies to what follows it, according to the precedence table on the final page of this book. Here is a syntax and equivalent specification.

chan c: T· P $=$ $\exists \mathbb{M}c$: ∞*T· $\exists \mathbb{T}c$: ∞**xreal*· **var** rc , wc: *xnat* := 0· P

The type T says what communications are possible on this new channel. The declaration introduces two scripts, which are infinite strings; they are not state variables, but state constants of

unknown value (mathematical variables). We have let time be extended real, but we could let it be extended integer. The channel declaration also introduces a read cursor rc with initial value 0 to say that initially there has been no input on this channel, and a write cursor wc with initial value 0 to say that initially there has been no output on this channel.

A local channel can be used without concurrency as a queue, or buffer. For example,

chan c: int· c! 3. c! 4. c?. x:= c. c?. x:= x+c

assigns 7 to x. Here is the proof, including time.

$$\textbf{chan } c\text{: } int\cdot\ c!\ 3.\ c!\ 4.\ t\text{:= } max\ t\ (\mathbb{T}_r + 1).\ c?.\ x\text{:= } c.\ t\text{:= } max\ t\ (\mathbb{T}_r + 1).\ c?.\ x\text{:= } x+c$$

$$= \quad \exists\mathbb{M}\text{: } \infty^*int\cdot\ \exists\mathbb{T}\text{: } \infty^*xint\cdot\ \ \textbf{var } r, w\text{: } xnat := 0\cdot$$
$$\mathbb{M}_w = 3\ \wedge\ \mathbb{T}_w = t\ \wedge\ (w\text{:= } w+1).$$
$$\mathbb{M}_w = 4\ \wedge\ \mathbb{T}_w = t\ \wedge\ (w\text{:= } w+1).$$
$$t\text{:= } max\ t\ (\mathbb{T}_r + 1).\ \ r\text{:= } r+1.$$
$$x\text{:= } \mathbb{M}_{r-1}.$$
$$t\text{:= } max\ t\ (\mathbb{T}_r + 1).\ \ r\text{:= } r+1.$$
$$x\text{:= } x + \mathbb{M}_{r-1}$$

now use the Substitution Law several times

$$= \quad \exists\mathbb{M}\text{: } \infty^*int\cdot\ \exists\mathbb{T}\text{: } \infty^*xint\cdot\ \exists r, r', w, w'\text{: } xnat\cdot$$
$$\mathbb{M}_0 = 3\ \wedge\ \mathbb{T}_0 = t\ \wedge\ \mathbb{M}_1 = 4\ \wedge\ \mathbb{T}_1 = t\ \wedge\ r' = 2\ \wedge\ w' = 2\ \wedge\ x' = \mathbb{M}_0 + \mathbb{M}_1$$
$$\wedge\ \ t' = max\ (max\ t\ (\mathbb{T}_0 + 1))\ (\mathbb{T}_1 + 1)\ \wedge\ \text{(other variables unchanged)}$$

$$= \quad x'{=}7\ \wedge\ t' = t{+}1\ \wedge\ \text{(other variables unchanged)}$$

Here are two processes with a communication between them. Ignoring time,

$$\textbf{chan } c\text{: } int\cdot\ c!\ 2\ \|\ (c?.\ x\text{:= } c)$$

Use the definition of local channel declaration, and use the previous result for the independent composition

$$= \quad \exists\mathbb{M}\text{: } \infty^*int\cdot\ \textbf{var } r, w\text{: } xnat := 0\cdot$$
$$\mathbb{M}_w = 2\ \wedge\ w' = w+1\ \wedge\ r'\text{:= } r+1\ \wedge\ x' = \mathbb{M}_r\ \wedge\ \text{(other variables unchanged)}$$

Now apply the initialization r:= 0 and w:= 0 using the Substitution Law

$$= \quad \exists\mathbb{M}\text{: } \infty^*int\cdot\ \textbf{var } r, w\text{: } xnat\cdot$$
$$\mathbb{M}_0 = 2\ \wedge\ w'{=}1\ \wedge\ r'{=}1\ \wedge\ x' = \mathbb{M}_0\ \wedge\ \text{(other variables unchanged)}$$
$$= \quad x'{=}2\ \wedge\ \text{(other variables unchanged)}$$
$$= \quad x\text{:= } 2$$

Replacing 2 by an arbitrary expression, we have a general theorem equating communication on a local channel with assignment. If we had included time, the result would have been

$$x'{=}2\ \wedge\ t' = t{+}1\ \wedge\ \text{(other variables unchanged)}$$
$$= \quad x\text{:= } 2.\ \ t\text{:= } t+1$$

End of Channel Declaration

9.1.8 Deadlock

In the previous subsection we saw that a local channel can be used as a buffer. Let's see what happens if we try to read first and write after. Inserting the input wait into

chan c: int· c?. c! 5

gives us

$\textbf{chan } c: int\cdot\ \ t:= max\ t\ (\mathbb{T}_r + 1).\ \ c?.\ \ c!\ 5$

$= \quad \exists\mathbb{M}: \infty{*}int\cdot\ \exists\mathbb{T}: \infty{*}xint\cdot\ \ \textbf{var } r, w: xnat := 0\cdot$
$t:= max\ t\ (\mathbb{T}_r + 1).\ \ r:= r{+}1.\ \ \mathbb{M}_w = 5\ \wedge\ \mathbb{T}_w = t\ \wedge (w:= w{+}1)$

We'll do this one slowly. First, expand **var** and $w:= w{+}1$, taking r , w , x , and t as the state variables.

$= \quad \exists\mathbb{M}: \infty{*}int\cdot\ \exists\mathbb{T}: \infty{*}xint\cdot\ \exists r, r', w, w': xnat\cdot$
$r:= 0.\ \ w:= 0.\ \ t:= max\ t\ (\mathbb{T}_r + 1).\ \ r:= r{+}1.$
$\mathbb{M}_w = 5\ \wedge\ \mathbb{T}_w = t\ \wedge\ r'{=}r\ \wedge\ w' = w{+}1\ \wedge\ x'{=}x\ \wedge\ t'{=}t$

Now use the Substitution Law four times.

$= \quad \exists\mathbb{M}: \infty{*}int\cdot\ \exists\mathbb{T}: \infty{*}xint\cdot\ \exists r, r', w, w': xnat\cdot$
$\mathbb{M}_0 = 5\ \wedge\ \mathbb{T}_0 = max\ t\ (\mathbb{T}_0 + 1)\ \wedge\ r'{=}1\ \wedge\ w'{=}1 \wedge x'{=}x\ \wedge t' = max\ t\ (\mathbb{T}_0 + 1)$

Look at the conjunct $\mathbb{T}_0 = max\ t\ (\mathbb{T}_0 + 1)$. For any start time $t > -\infty$ it says $\mathbb{T}_0 = \infty$.

$= \quad x'{=}x\ \wedge\ t'{=}\infty$

The theory tells us that execution takes forever because the wait for input is infinite.

The word "deadlock" is usually used to mean that several processes are waiting on each other, as in the dining philosophers example of Chapter 8. But it might also be used to mean that a single sequential computation is waiting on itself, as in the previous paragraph. Here's the more traditional example with two processes.

$\textbf{chan } c, d: int\cdot\ \ (c?.\ \ d!\ 6)\ \|\ (d?.\ \ c!\ 7)$

Inserting the input waits, we get

$\textbf{chan } c, d: int\cdot\ \ (t:= max\ t\ (\mathbb{T}c_{\ rc} + 1).\ \ c?.\ \ d!\ 6)\ \|\ (t:= max\ t\ (\mathbb{T}d_{\ rd} + 1).\ \ d?.\ \ c!\ 7)$

after a little work, we obtain

$= \quad \exists\mathbb{M}c, \mathbb{M}d: \infty{*}int\cdot\ \exists\mathbb{T}c, \mathbb{T}d: \infty{*}xint\cdot\ \exists rc, rc', wc, wc', rd, rd', wd, wd': xnat\cdot$
$\mathbb{M}d_{\ 0} = 6\ \wedge\ \mathbb{T}d_{\ 0} = max\ t\ (\mathbb{T}c_{\ 0} + 1)\ \wedge\ \mathbb{M}c_{\ 0} = 7\ \wedge\ \mathbb{T}c_{\ 0} = max\ t\ (\mathbb{T}d_{\ 0} + 1)$
$\wedge\ \ rc'{=}wc'{=}rd'{=}wd'{=}1 \wedge x'{=}x\ \wedge\ t' = max\ (max\ t\ (\mathbb{T}c_{\ 0} + 1))\ (max\ t\ (\mathbb{T}d_{\ 0} + 1))$

Once again, for start time $t > -\infty$, the conjuncts $\mathbb{T}d_{\ 0} = max\ t\ (\mathbb{T}c_{\ 0} + 1)\ \wedge\ \mathbb{T}c_{\ 0} = max\ t\ (\mathbb{T}d_{\ 0} + 1)$ tell us that $\mathbb{T}d_{\ 0} = \mathbb{T}c_{\ 0} = \infty$.

$= \quad x'{=}x\ \wedge\ t'{=}\infty$

To prove that a computation is free from deadlock, prove that all message times are finite.

End of Deadlock

9.1.9 Broadcast

A channel consists of a message script, a time script, a read cursor, and a write cursor. Whenever a computation splits into parallel processes, the state variables must be partitioned among the processes. The scripts are not state variables; they do not belong to any process. The cursors are state variables, so one of the processes can write to the channel, and one (perhaps the same one, perhaps a different one) can read from the channel. Suppose the structure is

$P.\ (Q\ \|\ R\ \|\ S).\ T$

and suppose Q writes to channel c and R reads from channel c . The messages written by Q follow those written by P , and those written by T follow those written by Q . The messages read by R follow those read by P , and those read by T follow those read by R . There is no problem of two processes attempting to write at the same time, and the timing discipline makes sure that reading a message waits until after it is written.

Although communication on a channel, as defined so far, is one-way from a single writer to a single reader, we can have as many channels as we want. So we can have two-way conversations between

all pairs of processes. But sometimes it is convenient to have a broadcast from one process to more than one of the parallel processes. In the program structure of the previous paragraph, we might want Q to write and both of R and S to read on the same channel. Broadcast is achieved by several read cursors, one for each reading process. Then all reading processes read the same messages, each at its own rate. There is no harm in two processes reading the same message, even at the same time. But there is a problem with broadcast: which of the read cursors becomes the read cursor for T ? All of the read cursors start with the same value, but they may not end with the same value. There is no sensible way to continue reading from that channel. So we allow broadcast on a channel only when the parallel composition is not followed sequentially by a program that reads from that channel.

We next present a broadcast example that combines communicating processes, local channel declaration, and dynamic process generation, in one beautiful little program. It is also a striking example of the importance of good notation and good theory. It has been "solved" before without them, but the "solutions" required many pages, intricate synchronization arguments, lacked proof, and were sometimes wrong.

Exercise 415 is multiplication of power series: Write a program to read from channel a an infinite sequence of coefficients $a_0\ a_1\ a_2\ a_3\ ...$ of a power series $a_0 + a_1{\times}x + a_2{\times}x^2 + a_3{\times}x^3 + ...$ and in parallel to read from channel b an infinite sequence of coefficients $b_0\ b_1\ b_2\ b_3\ ...$ of a power series $b_0 + b_1{\times}x + b_2{\times}x^2 + b_3{\times}x^3 + ...$ and in parallel to write on channel c the infinite sequence of coefficients $c_0\ c_1\ c_2\ c_3\ ...$ of the power series $c_0 + c_1{\times}x + c_2{\times}x^2 + c_3{\times}x^3 + ...$ equal to the product of the two input series. Assume that all inputs are already available; there are no input delays. Produce the outputs one per time unit.

The question provides us with a notation for the coefficients: $a_n = \mathbb{M}a_{\ ra+n}$, $b_n = \mathbb{M}b_{\ rb+n}$, and $c_n = \mathbb{M}c_{\ rc+n}$. Let us use A , B , and C for the power series, so we can express our desired result as

$$\begin{aligned} C &= A{\times}B \\ &= (a_0 + a_1{\times}x + a_2{\times}x^2 + a_3{\times}x^3 + ...) \times (b_0 + b_1{\times}x + b_2{\times}x^2 + b_3{\times}x^3 + ...) \\ &= \quad a_0{\times}b_0 + (a_0{\times}b_1 + a_1{\times}b_0){\times}x + (a_0{\times}b_2 + a_1{\times}b_1 + a_2{\times}b_0){\times}x^2 \\ &\quad + (a_0{\times}b_3 + a_1{\times}b_2 + a_2{\times}b_1 + a_3{\times}b_0){\times}x^3 + ... \end{aligned}$$

from which we see $c_n\ =\ \Sigma i\text{: } 0,..n{+}1\cdot\ a_i{\times}b_{n-i}$. The question relieves us from concern with input times, but we are still concerned with output times. The complete specification is

$$C = A{\times}B\ \land\ \forall n\cdot\ \mathbb{T}c_{\ wc+n} = t{+}n$$

Consider the problem: output coefficient n requires $n{+}1$ multiplications and n additions from $2{\times}(n{+}1)$ input coefficients, and it must be produced 1 time unit after the previous coefficient. To accomplish this requires more and more data storage, and more and more parallelism, as execution progresses.

As usual, let us concentrate on the result first, and leave the time for later. Let

$$A_1 = a_1 + a_2{\times}x + a_3{\times}x^2 + a_4{\times}x^3 + ...$$
$$B_1 = b_1 + b_2{\times}x + b_3{\times}x^2 + b_4{\times}x^3 + ...$$

be the power series from channels a and b beginning with coefficient 1 . Then

$$\begin{aligned} & A{\times}B \\ = \quad & (a_0 + A_1{\times}x) \times (b_0 + B_1{\times}x) \\ = \quad & a_0{\times}b_0 + (a_0{\times}B_1 + A_1{\times}b_0){\times}x + A_1{\times}B_1{\times}x^2 \end{aligned}$$

In place of the problem $A\times B$ we have five new problems. The first is to read one coefficient from each input channel and output their product; that's easy. The next two, $a_0\times B_1$ and $A_1\times b_0$, are multiplying a power series by a constant; that's easier than multiplying two power series, requiring only a loop. The next, $A_1\times B_1$, is exactly the problem we started with, but one coefficient farther along; it can be solved by recursion. Finally, we have to add three power series together. Unfortunately, these three power series are not synchronized properly. We must add the leading coefficients of $a_0\times B_1$ and $A_1\times b_0$ without any coefficient from $A_1\times B_1$, and thereafter add coefficient $n+1$ of $a_0\times B_1$ and $A_1\times b_0$ to coefficient n of $A_1\times B_1$. To synchronize, we move $a_0\times B_1$ and $A_1\times b_0$ one coefficient farther along. Let

$$A_2 = a_2 + a_3\times x + a_4\times x^2 + a_5\times x^3 + ...$$
$$B_2 = b_2 + b_3\times x + b_4\times x^2 + b_5\times x^3 + ...$$

be the power series from channels a and b beginning with coefficient 2 . Continuing the earlier equation for $A\times B$,

$$= \quad a_0\times b_0 + (a_0\times(b_1 + B_2\times x) + (a_1 + A_2\times x)\times b_0)\times x + A_1\times B_1\times x^2$$
$$= \quad a_0\times b_0 + (a_0\times b_1 + a_1\times b_0)\times x + (a_0\times B_2 + A_1\times B_1 + A_2\times b_0)\times x^2$$

From this expansion of the desired product we can almost write a solution directly.

One problem remains. A recursive call will be used to obtain a sequence of coefficients of the product $A_1\times B_1$ in order to produce the coefficients of $A\times B$. But the output channel for $A_1\times B_1$ cannot be channel c , the output channel for the main computation $A\times B$. Instead, a local channel must be used for output from $A_1\times B_1$. We need a channel parameter, for which we invent the notation $\langle ! \ \rangle$. A channel parameter is really four parameters: one for the message script, one for the time script, one for the write cursor, and one for the read cursor. (The cursors are variables, so their parameters are reference parameters; see Subsection 5.5.2.)

Now we are ready. Define P (for product) to be our specification (ignoring time for a moment) parameterized by output channel.

$$P = \langle !c\colon rat \rightarrow C = A\times B\rangle$$

We refine $P\ c$ as follows.

$$P\ c \quad \Leftarrow \quad (a? \,\|\, b?).\ \ c!\ a\times b.$$
$$\mathbf{var}\ a0\colon rat := a\cdot\ \ \mathbf{var}\ b0\colon rat := b\cdot\ \ \mathbf{chan}\ d\colon rat\cdot$$
$$P\ d\ \|\ ((a? \,\|\, b?).\ \ c!\ a0\times b + a\times b0.\ \ C = a0\times B + D + A\times b0)$$

$$C = a0\times B + D + A\times b0 \quad \Leftarrow \quad (a? \,\|\, b? \,\|\, d?).\ \ c!\ a0\times b + d + a\times b0.\ \ C = a0\times B + D + A\times b0$$

That is the whole program: 4 lines! First, an input is read from each of channels a and b and their product is output on channel c ; that takes care of $a_0\times b_0$. We will need these values again, so we declare local variables (really constants) $a0$ and $b0$ to retain their values. Now that we have read one message from each input channel, we call $P\ d$ to provide the coefficients of $A_1\times B_1$ on local channel d , in parallel with the remainder of the program. Both $P\ d$ and its parallel process will be reading from channels a and b using separate read cursors; there is no computation sequentially following them. In parallel with $P\ d$ we read the next inputs a_1 and b_1 and output the coefficient $a_0\times b_1 + a_1\times b_0$. Finally we execute the loop specified as $C = a0\times B + D + A\times b0$, where D is the power series whose coefficients are read from channel d .

The proof is completely straightforward. Here it is in detail. We start with the right side of the first refinement.

$(a? \| b?).\ \ c!\ a{\times}b.$
$\textbf{var}\ a0{:}\ rat := a\cdot\ \ \textbf{var}\ b0{:}\ rat := b\cdot\ \ \textbf{chan}\ d{:}\ rat\cdot$
$P\ d\ \ \|\ \ ((a? \| b?).\ \ c!\ a0{\times}b + a{\times}b0.\ \ C = a0{\times}B + D + A{\times}b0)$

$=$ $(ra{:}= ra+1 \| rb{:}= rb+1).\ \ \mathbb{M}c_{\ wc} = \mathbb{M}a_{\ ra-1} \times \mathbb{M}b_{\ rb-1}\ \ \wedge\ \ (wc{:}= wc+1).$
$\exists a0, a0', b0, b0', \mathbb{M}d, rd, rd', wd, wd'\cdot$
$a0{:}= \mathbb{M}a_{\ ra-1}.\ \ b0{:}= \mathbb{M}b_{\ rb-1}.\ \ rd{:}= 0.\ \ wd{:}= 0.$
$\quad(\forall n\cdot\ \mathbb{M}d_{\ wd+n} = (\Sigma i{:}\ 0,..n+1\cdot\ \mathbb{M}a_{\ ra+i} \times \mathbb{M}b_{\ rb+n-i}))$
$\wedge\ \ ((ra{:}= ra+1 \| rb{:}= rb+1).\ \ \mathbb{M}c_{\ wc} = a0{\times}\mathbb{M}b_{\ rb-1} + \mathbb{M}a_{\ ra-1}{\times}b0 \wedge (wc{:}= wc+1).$
$\quad\forall n\cdot\ \mathbb{M}c_{\ wc+n} = a0 \times \mathbb{M}b_{\ rb+n} + \mathbb{M}d_{\ rd+n} + \mathbb{M}a_{\ ra+n} \times b0)$

Make all substitutions indicated by assignments.

$=$ $\quad\mathbb{M}c_{\ wc} = \mathbb{M}a_{\ ra} \times \mathbb{M}b_{\ rb}$
$\wedge\ \ \exists a0, a0', b0, b0', \mathbb{M}d, rd, rd', wd, w_d'\cdot$
$\quad(\forall n\cdot\ \mathbb{M}d_{\ n} = \Sigma i{:}\ 0,..n+1\cdot\ \mathbb{M}a_{\ ra+1+i} \times \mathbb{M}b_{\ rb+1+n-i})$
$\quad\wedge\ \ \mathbb{M}c_{\ wc+1} = \mathbb{M}a_{\ ra} \times \mathbb{M}b_{\ rb+1} + \mathbb{M}a_{\ ra+1} \times \mathbb{M}b_{\ rb}$
$\quad\wedge\ \ (\forall n\cdot\ \mathbb{M}c_{\ wc+2+n} = \mathbb{M}a_{\ ra} \times \mathbb{M}b_{\ rb+2+n} + \mathbb{M}d_{\ n} + \mathbb{M}a_{\ ra+2+n} \times \mathbb{M}b_{\ rb})$

Use the first universal quantification to replace $\mathbb{M}_d n$ in the second.
Then throw away the first universal quantification (weakening our expression).
Now all existential quantifications are unused, and can be thrown away.

$\Longrightarrow$ $\quad\mathbb{M}c_{\ wc} = \mathbb{M}a_{\ ra} \times \mathbb{M}b_{\ rb}$
$\wedge\ \ \mathbb{M}c_{\ wc+1} = \mathbb{M}a_{\ ra} \times \mathbb{M}b_{\ rb+1} + \mathbb{M}a_{\ ra+1} \times \mathbb{M}b_{\ rb}$
$\wedge\ \ \forall n\cdot\ \mathbb{M}c_{\ wc+2+n} = \quad \mathbb{M}a_{\ ra} \times \mathbb{M}b_{\ rb+2+n}$
$\qquad\qquad + \ (\Sigma i{:}\ 0,..n+1\cdot\ \mathbb{M}a_{\ ra+1+i} \times \mathbb{M}b_{\ rb+1+n-i})$
$\qquad\qquad + \ \mathbb{M}a_{\ ra+2+n} \times \mathbb{M}b_{\ rb}$

Now put the three conjuncts together.

$=$ $\forall n\cdot\ \mathbb{M}c_{\ wc+n} = \Sigma i{:}\ 0,..n+1\cdot\ \mathbb{M}a_{\ ra+i} \times \mathbb{M}b_{\ rb+n-i}$
$=$ $P\ c$

We still have to prove the loop refinement.

$(a? \| b? \| d?).\ \ c!\ a0{\times}b + d + a{\times}b0.\ \ C = a0{\times}B + D + A{\times}b0$

$=$ $(ra{:}= ra+1 \| rb{:}= rb+1 \| rd{:}= rd+1).$
$\mathbb{M}c_{\ wc} = a0 \times \mathbb{M}b_{\ rb-1} + \mathbb{M}d_{\ rd-1} + \mathbb{M}a_{\ ra-1} \times b0\ \ \wedge\ \ (wc{:}= wc+1).$
$\forall n\cdot\ \mathbb{M}c_{\ wc+n} = a0 \times \mathbb{M}b_{\ rb+n} + \mathbb{M}d_{\ rd+n} + \mathbb{M}a_{\ ra+n} \times b0$

Make all substitutions indicated by assignments.

$=$ $\quad\mathbb{M}c_{\ wc} = a0 \times \mathbb{M}b_{\ rb} + \mathbb{M}d_{\ rd} + \mathbb{M}a_{\ ra} \times b0$
$\wedge\ \ \forall n\cdot\ \mathbb{M}c_{\ wc+1+n} = a0 \times \mathbb{M}b_{\ rb+1+n} + \mathbb{M}d_{\ rd+1+n} + \mathbb{M}a_{\ ra+1+n} \times b0$

Put the two conjuncts together.

$=$ $\forall n\cdot\ \mathbb{M}c_{\ wc+n} = a0 \times \mathbb{M}b_{\ rb+n} + \mathbb{M}d_{\ rd+n} + \mathbb{M}a_{\ ra+n} \times b0$
$=$ $C = a0{\times}B + D + A{\times}b0$

According to the recursive measure of time, we must place a time increment before the recursive call $P\ d$ and before the recursive call $C = a0{\times}B + D + A{\times}b0$. We do not need a time increment before inputs on channels a and b according to information given in the question. We do need a time increment before the input on channel d . Placing only these necessary time increments, output $c_0 = a_0{\times}b_0$ will occur at time $t+0$ as desired, but output $c_1 = a_0{\times}b_1 + a_1{\times}b_0$ will also occur at time $t+0$, which is too soon. In order to make output c_1 occur at time $t+1$ as desired, we must place a time increment between the first two outputs. We can consider this time increment to account for actual computing time, or as a delay (see Section 5.3, “Time and Space Dependence”). Here is the program with time.

$$Q\,c \;\Leftarrow\; (a? \,\|\, b?).\;\; c!\; a{\times}b.$$
$$\textbf{var}\; a0: rat := a\cdot\;\; \textbf{var}\; b0: rat := b\cdot\;\; \textbf{chan}\; d: rat\cdot$$
$$(t{:=}\, t{+}1.\;\; Q\,d) \;\|\; ((a? \,\|\, b?).\;\; t{:=}\, t{+}1.\;\; c!\; a0{\times}b + a{\times}b0.\;\; R)$$

$$R \;\Leftarrow\; (a? \,\|\, b? \,\|\, (t{:=}\, max\; t\; (\mathbb{T}d_{rd} + 1).\;\; d?)).\;\; c!\; a0{\times}b + d + a{\times}b0.\;\; t{:=}\, t{+}1.\;\; R$$

where Q and R are defined, as follows:

$$Q\,c \;=\; \forall n\cdot\; \mathbb{T}c_{wc+n} = t{+}n$$
$$Q\,d \;=\; \forall n\cdot\; \mathbb{T}d_{wd+n} = t{+}n$$
$$R \;=\; (\forall n\cdot\; \mathbb{T}d_{rd+n} = t{+}n) \;\Rightarrow\; (\forall n\cdot\; \mathbb{T}c_{wc+n} = t{+}1{+}n)$$

Within loop R, the assignment $t{:=}\, max\; t\; (\mathbb{T}d_{rd} + 1)$ represents a delay of 1 time unit the first iteration (because $t = \mathbb{T}d_{rd}$), and a delay of 0 time units each subsequent iteration (because $t = \mathbb{T}d_{rd} + 1$). This makes the proof very ugly. To make the proof pretty, we can replace $t{:=}\, max\; t\; (\mathbb{T}d_{rd} + 1)$ by $t{:=}\, max\; (t{+}1)\; (\mathbb{T}d_{rd} + 1)$ and delete $t{:=}\, t{+}1$ just before the call to R. These changes together do not change the timing at all; they just make the proof easier. The assignment $t{:=}\, max\; (t{+}1)\; (\mathbb{T}d_{rd} + 1)$ increases the time by at least 1, so the loop includes a time increase without the $t{:=}\, t{+}1$. The program with time is now

$$Q\,c \;\Leftarrow\; (a? \,\|\, b?).\;\; c!\; a{\times}b.$$
$$\textbf{var}\; a0: rat := a\cdot\;\; \textbf{var}\; b0: rat := b\cdot\;\; \textbf{chan}\; d: rat\cdot$$
$$(t{:=}\, t{+}1.\;\; Q\,d) \;\|\; ((a? \,\|\, b?).\;\; t{:=}\, t{+}1.\;\; c!\; a0{\times}b + a{\times}b0.\;\; R)$$

$$R \;\Leftarrow\; (a? \,\|\, b? \,\|\, (t{:=}\, max\; (t{+}1)\; (\mathbb{T}d_{rd} + 1).\;\; d?)).\;\; c!\; a0{\times}b + d + a{\times}b0.\;\; R$$

Here is the proof of the first of these refinements, beginning with the right side.

$$(a? \,\|\, b?).\;\; c!\; a{\times}b.$$
$$\textbf{var}\; a0: rat := a\cdot\;\; \textbf{var}\; b0: rat := b\cdot\;\; \textbf{chan}\; d: rat\cdot$$
$$(t{:=}\, t{+}1.\;\; Q\,d) \;\|\; ((a? \,\|\, b?).\;\; t{:=}\, t{+}1.\;\; c!\; a0{\times}b + a{\times}b0.\;\; R)$$

We can ignore $a?$ and $b?$ because they have no effect on timing (they are substitutions for variables that do not appear in $Q\,d$ and R). We also ignore what messages are output, looking only at their times. We can therefore also ignore variables $a0$ and $b0$.

$$\Rightarrow\quad \mathbb{T}c_{wc} = t \;\wedge\; (wc{:=}\, wc{+}1).$$
$$\exists \mathbb{T}d, rd, rd', wd, wd'\cdot\;\; rd{:=}\, 0.\;\; wd{:=}\, 0.$$
$$(t{:=}\, t{+}1.\;\; \forall n\cdot\; \mathbb{T}d_{wd+n} = t{+}n)$$
$$\wedge\;\; (\; t{:=}\, t{+}1.\;\; \mathbb{T}c_{wc} = t \;\wedge (wc{:=}\, wc{+}1).$$
$$(\forall n\cdot\; \mathbb{T}d_{rd+n} = t{+}n) \Rightarrow (\forall n\cdot\; \mathbb{T}c_{wc+n} = t{+}1{+}n)\;)$$

Make all substitutions indicated by assignments.

$$=\quad \mathbb{T}c_{wc} = t$$
$$\wedge\;\; \exists \mathbb{T}d, rd, rd', wd, wd'\cdot$$
$$(\forall n\cdot\; \mathbb{T}d_{n} = t{+}1{+}n)$$
$$\wedge\; \mathbb{T}c_{wc+1} = t{+}1$$
$$\wedge\; ((\forall n\cdot\; \mathbb{T}d_{n} = t{+}1{+}n) \Rightarrow (\forall n\cdot\; \mathbb{T}c_{wc+2+n} = t{+}2{+}n))$$

Use the first universal quantification to discharge the antecedent.
Then throw away the first universal quantification (weakening our expression).
Now all existential quantifications are unused, and can be thrown away.

$$\Rightarrow\quad \mathbb{T}c_{wc} = t \;\wedge\; \mathbb{T}c_{wc+1} = t{+}1 \;\wedge\; \forall n\cdot\; \mathbb{T}c_{wc+2+n} = t{+}2{+}n$$

Now put the three conjuncts together.

$$=\quad \forall n\cdot\; \mathbb{T}c_{wc+n} = t{+}n$$
$$=\quad Q\,c$$

We still have to prove the loop refinement.

$(R \Leftarrow (a? \| b? \| (t{:=}\ max\ (t+1)\ (\mathbb{T}d_{rd}+1).\ d?)).\ c!\ a0{\times}b + d + a{\times}b0.\ R)$

Ignore $a?$ and $b?$ and the output message.

$$\Leftarrow \quad ((\forall n\cdot \mathbb{T}d_{rd+n} = t+n) \Rightarrow (\forall n\cdot \mathbb{T}c_{wc+n} = t+1+n))$$
$$\Leftarrow (\ t{:=}\ max\ (t+1)\ (\mathbb{T}d_{rd}+1).\ rd{:=}\ rd+1.\ \mathbb{T}c_{wc} = t\ \wedge (wc{:=}\ wc+1).$$
$$(\forall n\cdot \mathbb{T}d_{rd+n} = t+n) \Rightarrow (\forall n\cdot \mathbb{T}c_{wc+n} = t+1+n)\ )$$

Use the Law of Portation to move the first antecedent to the right side, where it becomes a conjunct.

$$= \quad (\forall n\cdot \mathbb{T}c_{wc+n} = t+1+n)$$
$$\Leftarrow \quad (\forall n\cdot \mathbb{T}d_{rd+n} = t+n)$$
$$\wedge\ (\ t{:=}\ max\ (t+1)\ (\mathbb{T}d_{rd}+1).\ rd{:=}\ rd+1.\ \mathbb{T}c_{wc} = t\ \wedge (wc{:=}\ wc+1).$$
$$(\forall n\cdot \mathbb{T}d_{rd+n} = t+n) \Rightarrow (\forall n\cdot \mathbb{T}c_{wc+n} = t+1+n)\ )$$

Specializing $\forall n\cdot \mathbb{T}d_{rd+n} = t+n$ to the case $n{=}0$, we use $\mathbb{T}d_{rd} = t$ to simplify $max\ (t+1)\ (\mathbb{T}d_{rd}+1)$.

$$= \quad (\forall n\cdot \mathbb{T}c_{wc+n} = t+1+n)$$
$$\Leftarrow \quad (\forall n\cdot \mathbb{T}d_{rd+n} = t+n)$$
$$\wedge\ (\ t{:=}\ t+1.\ rd{:=}\ rd+1.\ \mathbb{T}c_{wc} = t\ \wedge (wc{:=}\ wc+1).$$
$$(\forall n\cdot \mathbb{T}d_{rd+n} = t+n) \Rightarrow (\forall n\cdot \mathbb{T}c_{wc+n} = t+1+n)\ )$$

Make all substitutions indicated by assignments.

$$= \quad (\forall n\cdot \mathbb{T}c_{wc+n} = t+1+n)$$
$$\Leftarrow \quad (\forall n\cdot \mathbb{T}d_{rd+n} = t+n)$$
$$\wedge\ \mathbb{T}c_{wc} = t+1$$
$$\wedge\ ((\forall n\cdot \mathbb{T}d_{rd+1+n} = t+1+n) \Rightarrow (\forall n\cdot \mathbb{T}c_{wc+1+n} = t+2+n))$$

The conjunct $\forall n\cdot \mathbb{T}d_{rd+n} = t+n$ discharges the antecedent $\forall n\cdot \mathbb{T}d_{rd+1+n} = t+1+n$ which can be dropped.

$$\Leftarrow \quad (\forall n\cdot \mathbb{T}c_{wc+n} = t+1+n)$$
$$\Leftarrow \mathbb{T}c_{wc} = t+1\ \wedge\ (\forall n\cdot \mathbb{T}c_{wc+1+n} = t+2+n)$$

$$= \quad \top$$

End of Broadcast

End of Communication

End of Interaction

For many students, the first understanding of programs they are taught is how programs are executed. And for many students, that is the only understanding they are given. With that understanding, the only method available for checking whether a program is correct is to test it by executing it with a variety of inputs to see if the resulting outputs are right. All programs should be tested, but there are two problems with testing.

One problem with testing is: how do you know if the outputs are right? For some programs, such as graphics programs for producing pretty pictures, the only way to know if the output is right is to test the program and judge the result. But in other cases, a program may give answers you do not already know (that may be why you wrote the program), and testing it does not tell you if it is right. In such cases, you should test to see at least if the answers are reasonable.

The other problem with testing is: you cannot try all inputs. Even if all the test cases you try give reasonable answers, there may be errors lurking in untried cases.

If you have read and understood this book to here, you now have an understanding of programs that is completely different from execution. When you prove that a program refines a specification, you are considering all inputs at once, and you are proving that the outputs have the properties stated in the specification. That is far more than can ever be accomplished by testing. But it is also more work than trying some inputs and looking at the outputs. That raises the question: when is the extra assurance of correctness worth the extra work?

If the program you are writing is easy enough that you can probably get it right without any theory, and it does not really matter if there are some errors in it, then the extra assurance of correctness provided by the theory may not be worth the trouble. If you are writing a pacemaker controller for a heart, or the software that controls a subway system, or an air traffic control program, or nuclear power plant software, or any other programs that people's lives will depend on, then the extra assurance is definitely worth the trouble, and you would be negligent if you did not use the theory.

To prove that a program refines a specification after the program is finished is a very difficult task. It is much easier to perform the proof while the program is being written. The information needed to make one step in programming is exactly the same information that is needed to prove that step is correct. The extra work is mainly to write down that information formally. It is also the same information that will be needed later for program modification, so writing it explicitly at each step will save effort later. And if you find, by trying to prove a step, that the step is incorrect, you save the effort of building the rest of your program on a wrong step. As a further bonus, after you become practiced and skillful at using the theory, you find that it helps in the program design; it suggests programming steps. In the end, it may not be any extra effort at all.

In this book we have looked only at small programs. But the theory is not limited to small programs; it is independent of scale, applicable to any size of software. In a large software project, the first design decision might be to divide the task into several pieces that will fit together in some way. This decision can be written as a refinement, specifying exactly what the parts are and how they fit together, and then the refinement can be proven. Using the theory in the early stages is enormously beneficial, because if an early step is wrong, it is enormously costly to correct later.

For a theory of programming to be in widespread use for industrial program design, it must be supported by tools. Ideally, an automated prover checks each refinement, remaining silent if the refinement is correct, complaining whenever there is a mistake, and saying exactly what is wrong. At present there are a few tools that provide some assistance, but they are far from ideal. There is plenty of opportunity for tool builders, and they need a thorough knowledge of a practical theory of programming.

10 Exercises

Exercises marked with √ have been done in previous chapters.

10.0 Preface

0 There are four cards on a table showing symbols D, E, 2, and 3 (one per card). Each card has a letter on one side and a digit on the other. Which card(s) do you need to turn over to determine whether every card with a D on one side has a 3 on the other? Why?

End of Preface

10.1 Basic Theories

1 Simplify each of the following boolean expressions.

(a) $x \land \neg x$
(b) $x \lor \neg x$
(c) $x \Rightarrow \neg x$
(d) $x \Leftarrow \neg x$
(e) $x = \neg x$
(f) $x \ne \neg x$

2 Prove each of the following laws of Boolean Theory using the proof format given in Subsection 1.0.1, and any laws listed in Section 11.4. Do not use the Completion Rule.

(a) $a{\land}b \Rightarrow a{\lor}b$
(b) $(a{\land}b) \lor (b{\land}c) \lor (a{\land}c) \;=\; (a{\lor}b) \land (b{\lor}c) \land (a{\lor}c)$
(c) $\neg a \Rightarrow (a \Rightarrow b)$
(d) $a = (b \Rightarrow a) \;=\; a \lor b$
(e) $a = (a \Rightarrow b) \;=\; a \land b$
(f) $(a{\Rightarrow}c) \land (b{\Rightarrow}\neg c) \Rightarrow \neg(a{\land}b)$
(g) $a \land \neg b \Rightarrow a \lor b$
(h) $(a{\Rightarrow}b) \land (c{\Rightarrow}d) \land (a{\lor}c) \Rightarrow (b{\lor}d)$
(i) $a \land \neg a \Rightarrow b$
(j) $(a{\Rightarrow}b) \lor (b{\Rightarrow}a)$
(k)√ $\neg(a \land \neg(a{\lor}b))$
(l) $(\neg a{\Rightarrow}\neg b) \land (a{\ne}b) \;\lor\; (a{\land}c \Rightarrow b{\land}c)$
(m) $(a{\Rightarrow}\neg a) \Rightarrow \neg a$
(n) $(a{\Rightarrow}b) \land (\neg a{\Rightarrow}b) \;=\; b$
(o) $(a{\Rightarrow}b){\Rightarrow}a \;=\; a$
(p) $a{=}b \lor a{=}c \lor b{=}c$
(q) $a{\land}b \lor a{\land}\neg b \;=\; a$
(r) $a{\Rightarrow}(b{\Rightarrow}a)$
(s) $a \Rightarrow a \land b \;=\; a \Rightarrow b \;=\; a \lor b \Rightarrow b$
(t) **if** a **then** a **else** $\neg a$
(u) **if** $b{\land}c$ **then** P **else** Q $=$ **if** b **then if** c **then** P **else** Q **else** Q
(v) **if** $b{\lor}c$ **then** P **else** Q $=$ **if** b **then** P **else if** c **then** P **else** Q
(w) **if** b **then** P **else if** b **then** Q **else** R $=$ **if** b **then** P **else** R
(x) **if if** b **then** c **else** d **then** P **else** Q
$=$ **if** b **then if** c **then** P **else** Q **else if** d **then** P **else** Q
(y) **if** b **then if** c **then** P **else** R **else if** c **then** Q **else** R
$=$ **if** c **then if** b **then** P **else** Q **else** R

3 (dual) One operator is the dual of another operator if it negates the result when applied to the negated operands. The zero-operand operators $\top$ and $\bot$ are each other's duals. If $op_0\,(\neg a) = \neg(op_1\; a)$ then op_0 and op_1 are duals. If $(\neg a)\; op_0\; (\neg b) = \neg(a\; op_1\; b)$ then op_0 and op_1 are duals. And so on for more operands.

(a) Of the 4 one-operand boolean operators, there is 1 pair of duals, and 2 operators that are their own duals. Find them.

(b) Of the 16 two-operand boolean operators, there are 6 pairs of duals, and 4 operators that are their own duals. Find them.

(c) What is the dual of the three-operand operator **if then else** ? Express it using only the operator **if then else** .

(d) The dual of a boolean expression without variables is formed as follows: replace each operator with its dual, adding parentheses if necessary to maintain the precedence. Explain why the dual of a theorem is an antitheorem, and vice versa.

(e) Let P be a boolean expression without variables. From part (d) we know that every boolean expression without variables of the form

$$(\text{dual of } P) = \neg P$$

is a theorem. Therefore, to find the dual of a boolean expression with variables, we must replace each operator by its dual and negate each variable. For example, if a and b are boolean variables, then the dual of $a \wedge b$ is $\neg a \vee \neg b$. And since

$$(\text{dual of } a \wedge b) = \neg(a \wedge b)$$

we have one of the Duality Laws:

$$\neg a \vee \neg b = \neg(a \wedge b)$$

The other of the Duality Laws is obtained by equating the dual and negation of $a \vee b$. Obtain five laws that do not appear in this book by equating a dual with a negation.

(f) Dual operators have truth tables that are each other's vertical mirror reflections. For example, the truth table for $\wedge$ (below left) is the vertical mirror reflection of the truth table for $\vee$ (below right).

$\wedge$:

operands	result
$\top\top$	$\top$
$\top\bot$	$\bot$
$\bot\top$	$\bot$
$\bot\bot$	$\bot$

$\vee$:

operands	result
$\top\top$	$\top$
$\top\bot$	$\top$
$\bot\top$	$\top$
$\bot\bot$	$\bot$

Design symbols (you may redesign existing symbols where necessary) for the 4 one-operand and 16 two-operand boolean operators according to the following criteria.

(i) Dual operators should have symbols that are vertical mirror reflections (like $\wedge$ and $\vee$). This implies that self-dual operators have vertically symmetric symbols, and all others have vertically asymmetric symbols.

(ii) If $a\; op_0\; b = b\; op_1\; a$ then op_0 and op_1 should have symbols that are horizontal mirror reflections (like $\Rightarrow$ and $\Leftarrow$). This implies that symmetric operators have horizontally symmetric symbols, and all others have horizontally asymmetric symbols.

4 Truth tables and the Evaluation Rule can be replaced by a new proof rule and some new axioms. The new proof rule says: "A boolean expression does not gain, lose, or change classification when a theorem within it is replaced by another theorem. Similarly, a boolean expression does not gain, lose, or change classification when an antitheorem within it is replaced by another antitheorem.". The truth tables become new axioms; for example, one truth table entry becomes the axiom $\top \vee \top$ and another becomes the axiom $\top \vee \bot$. These two axioms can be reduced to one axiom by the introduction of a variable, giving $\top \vee x$. Write the truth tables as axioms and antiaxioms as succinctly as possible.

5 Complete the following laws of Boolean Theory

(a) $\top =$

(b) $\bot =$

(c) $\neg a =$

(d) $a \wedge b =$

(e) $a \vee b =$

(f) $a = b =$

(g) $a \neq b =$

(h) $a \Rightarrow b =$

by adding a right side using only the following symbols (in any quantity)

(i) $\neg\ \wedge\ a\ b\ (\ )$

(ii) $\neg\ \vee\ a\ b\ (\ )$

(iii) $\neg\ \Rightarrow\ a\ b\ (\ )$

(iv) $\neq\ \Rightarrow\ a\ b\ (\ )$

(v) $\neg$ **if then else** a b ( )

6 (BDD) A BDD (Binary Decision Diagram) is a boolean expression that has one of the following 3 forms: $\top$, $\bot$, **if** variable **then** BDD **else** BDD . For example,

if x **then if** a **then** $\top$ **else** $\bot$ **else if** y **then if** b **then** $\top$ **else** $\bot$ **else** $\bot$

is a BDD. An OBDD (Ordered BDD) is a BDD with an ordering on the variables, and in each **if then else** , the variable in the **if**-part must come before any of the variables in its **then**- and **else**-parts ("before" means according to the ordering). For example, using alphabetic ordering for the variables, the previous example is not an OBDD, but

if a **then if** c **then** $\top$ **else** $\bot$ **else if** b **then if** c **then** $\top$ **else** $\bot$ **else** $\bot$

is an OBDD. An LBDD (Labeled BDD) is a set of definitions of the following 3 forms:

label = $\top$
label = $\bot$
label = **if** variable **then** label **else** label

The labels are separate from the variables; each label used in a **then**-part or **else**-part must be defined by one of the definitions; exactly one label must be defined but unused. The following is an LBDD.

true = $\top$
false = $\bot$
alice = **if** b **then** true **else** false
bob = **if** a **then** alice **else** false

An LOBDD is an LBDD that becomes an OBDD when the labels are expanded. The ordering prevents any recursive use of the labels. The previous example is an LOBDD. An RBDD (Reduced BDD) is a BDD such that, in each **if then else** , the **then**- and **else**-parts differ. An ROBDD is both reduced and ordered; an RLBDD is both reduced and labeled; an RLOBDD is reduced, labeled, and ordered. The previous example is an RLOBDD.

(a) Express $\neg a$, $a \wedge b$, $a \vee b$, $a \Rightarrow b$, $a = b$, $a \neq b$, and **if** a **then** b **else** c as BDDs.

(b) How can you conjoin two OBDDs and get an OBDD?

(c) How can you determine if two RLOBDDs are equal?

(d) How can we represent an RLOBDD in order to determine efficiently if an assignment of values to variables satisfies it (solves it, gives it value $\top$)?

7 Express formally and succinctly that exactly one of three statements is true.

8 Design symbols for the 10 two-operand boolean operators that are not presented in Chapter 1, and find laws about these operators.

9 The Case Analysis Laws equate the three-operand operator **if** a **then** b **else** c to expressions using only two-operand and one-operand operators. In each, the variable a appears twice. Find an equal expression using only two-operand and one-operand operators in which the variable a appears only once. Hint: use continuing operators.

10 Consider a fully parenthesized expression containing only the symbols $\top \perp = \neq ( )$ in any quantity and any syntactically acceptable order.

(a) Show that all syntactically acceptable rearrangements are equivalent.

(b) Show that it is equivalent to any expression obtained from it by making an even number of the following substitutions: $\top$ for $\perp$, $\perp$ for $\top$, $=$ for $\neq$, $\neq$ for $=$.

11 Let p and q be boolean expressions. Suppose p is both a theorem and an antitheorem (the theory is inconsistent).

(a) Prove, using the rules of proof presented, that q is both a theorem and an antitheorem.

(b) Is $q=q$ a theorem or an antitheorem?

12 Formalize each of the following statements as a boolean expression. Start by staying as close as possible to the English, then simplify as much as possible (sometimes no simplification is possible). You will have to introduce new basic boolean expressions like (the door can be opened) for the parts that cannot make use of boolean operators, but for words like “only if” you should use boolean operators. You translate meanings from words to boolean symbols; the meaning of the words may depend on their context and even on facts not explicitly stated. Formalization is not a simple substitution of symbols for words.

(a) The door can only be opened if the elevator is stopped.

(b) Neither the elevator door nor the floor door will open unless both of them do.

(c) Either the motor is jammed or the control is broken.

(d) Either the light is on or it is off.

(e) If you press the button, the elevator will come.

(f) If the power switch is on, the system is operating.

(g) Where there's smoke, there's fire; and there's no smoke; so there's no fire.

(h) Where there's smoke, there's fire; and there's no fire; so there's no smoke.

(i) You can't score if you don't shoot.

(j) If you have a key, only then can you open the door.

(k) No pain, no gain.

(l) No shirt? No shoes? No service!

(m) If it happens, it happens.

13 Formalize each of the following statements. For each pair, either prove they are equivalent or prove they differ.

(a) Don't drink and drive.

(b) If you drink, don't drive.

(c) If you drive, don't drink.

(d) Don't drink and don't drive.

(e) Don't drink or don't drive.

14 Formalize and prove the following argument. If it is raining and Jane does not have her umbrella with her, then she is getting wet. It is raining. Jane is not getting wet. Therefore Jane has her umbrella with her.

15 A sign says:

NO PARKING 7-9am 4-6pm Mon-Fri

Using variable t for time of day and d for day of week, write a boolean expression that says when there is no parking.

16 (tennis) An advertisement for a tennis magazine says “If I'm not playing tennis, I'm watching tennis. And if I'm not watching tennis, I'm reading about tennis.”. Assuming the speaker cannot do more than one of these activities at a time,

(a) prove that the speaker is not reading about tennis.

(b) what is the speaker doing?

17 (maid and butler) The maid said she saw the butler in the living room. The living room adjoins the kitchen. The shot was fired in the kitchen, and could be heard in all nearby rooms. The butler, who had good hearing, said he did not hear the shot. Given these facts, prove that someone lied. Use the following abbreviations.

mtt = (the maid told the truth)
btt = (the butler told the truth)
blr = (the butler was in the living room)
bnk = (the butler was near the kitchen)
bhs = (the butler heard the shot)

18 (knights and knaves) There are three inhabitants of an island, named P, Q, and R. Each is either a knight or a knave. Knights always tell the truth. Knaves always lie. For each of the following, write the given information formally, and then answer the questions, with proof.

(a) You ask P: “Are you a knight?”. P replies: “If I am a knight, I'll eat my hat.”. Does P eat his hat?

(b) P says: “If Q is a knight, then I am a knave.”. What are P and Q?

(c) P says: “There is gold on this island if and only if I am a knight.”. Can it be determined whether P is a knight or a knave? Can it be determined whether there is gold on the island?

(d) P, Q, and R are standing together. You ask P: “Are you a knight or a knave?”. P mumbles his reply, and you don't hear it. So you ask Q: “What did P say?”. Q replies: “P said that he is a knave.”. Then R says: “Don't believe Q, he's lying.”. What are Q and R?

(e) You ask P: “How many of you are knights?”. P mumbles. So Q says: “P said there is exactly one knight among us.”. R says: “Don't believe Q, he's lying.”. What are Q and R?

(f) P says: “We're all knaves.”. Q says: “No, exactly one of us is a knight.”. What are P, Q, and R?

19 Islands X and Y contain knights who always tell the truth, knaves who always lie, and possibly also some normal people who sometimes tell the truth and sometimes lie. There is gold on at least one of the islands, and the people know which island(s) it is on. You find a message from the pirate who buried the gold, with the following clue (which we take as an axiom): “If there are any normal people on these islands, then there is gold on both islands.”. You are allowed to dig on only one island, and you are allowed to ask one question of one random person. What should you ask in order to find out which island to dig on?

20 (caskets) The princess had two caskets, one gold and one silver. Into one she placed her portrait and into the other she placed a dagger. On the gold casket she wrote the inscription: the portrait is not in here. On the silver casket she wrote the inscription: exactly one of these inscriptions is true. She explained to her suitor that each inscription is either true or false (not both), but on the basis of the inscriptions he must choose a casket. If he chooses the one with the portrait, he can marry her; if he chooses the one with the dagger, he must kill himself. Assuming marriage is preferable to death, which casket should he choose?

21 (the unexpected egg) There are two boxes, one red and one blue. One box has an egg in it; the other is empty. You are to look first in the red box, then if necessary in the blue box, to find the egg. But you will not know which box the egg is in until you open the box and see the egg. You reason as follows: “If I look in the red box and find it empty, I'll know that the egg is in the blue box without opening it. But I was told that I would not know which box the egg is in until I open the box and see the egg. So it can't be in the blue box. Now I know it must be in the red box without opening the red box. But again, that's ruled out, so it isn't in either box.”. Having ruled out both boxes, you open them and find the egg in one unexpectedly, as originally stated. Formalize the given statements and the reasoning, and thus explain the paradox.

22 A number can be written as a sequence of decimal digits. For the sake of generality, let us consider using the sequence notation with arbitrary expressions, not just digits. For example, $1(2+3)4$ could be allowed, and be equal to 154. What changes are needed to the number axioms?

23 (scale) There is a tradition in programming languages to use a scale operator, e , in the limited context of digit sequences. Thus $12e3$ is equal to 12×10^3. For the sake of generality, let us consider using the scale notation with arbitrary expressions, not just digits. For example, $(6+6)e(5-2)$ could be allowed, and be equal to $12e3$. What changes are needed to the number axioms?

24 When we defined number expressions, we included complex numbers such as $(-1)^{1/2}$, not because we particularly wanted them, but because it was easier than excluding them. If we were interested in complex numbers, we would find that the number axioms given in Subsection 11.4.2 do not allow us to prove many things we might like to prove. For example, we cannot prove $(-1)^{1/2} \times 0 = 0$. How can the axioms be made strong enough to prove things about complex numbers, but weak enough to leave room for ∞ ?

25 Express formally
(a) the absolute value of a real number x.
(b) the sign of a real number x, which is -1, 0, or $+1$ depending on whether x is negative, zero, or positive.

26 Prove $-\infty<y<\infty \ \wedge\ y\neq0 \ \Rightarrow\ (x/y=z \ = \ x=z\times y)$.

27 Show that the number axioms become inconsistent when we add the axiom

$$-\infty<y<\infty \ \Rightarrow\ x/y\times y = x$$

28 (circular numbers) Redesign the axioms for the extended number system to make it circular, so that $+\infty = -\infty$. Be careful with the transitivity of $<$.

29 Is there any harm in adding the axiom $0/0=5$ to Number Theory?

30 (bracket algebra) Here is a new way to write boolean expressions. An expression can be empty; in other words, nothing is already an expression. If you put a pair of parentheses around an expression, you get another expression. If you put two expressions next to each other, you get another expression. For example,

()(())((())())

is an expression. The empty expression is bracket algebra's way of writing $\top$; putting parentheses around an expression is bracket algebra's way of negating it, and putting expressions next to each other is bracket algebra's way of conjoining them. So the example expression is bracket algebra's way of saying

$\neg\top\wedge\neg\neg\top\wedge\neg(\neg\neg\top\wedge\neg\top)$

We can also have variables anywhere in a bracket expression. There are three rules of bracket algebra. If x , y , and z are any bracket expressions, then

$((x))$	can replace or be replaced by	x	double negation rule
$x()y$	can replace or be replaced by	$()$	base rule
$x\ y\ z$	can replace or be replaced by	$x'\ y\ z'$	context rule

where x' is x with occurrences of y added or deleted, and similarly z' is z with occurrences of y added or deleted. The context rule does not say how many occurrences of y are added or deleted; it could be any number from none to all of them. To prove, you just follow the rules until the expression disappears. For example,

	$((a)b((a)b))$	context rule: empty for x , $(a)b$ for y , $((a)b)$ for z
becomes	$((a)b(\quad))$	base rule: $(a)b$ for x and empty for y
becomes	$(\quad(\quad))$	double negation rule: empty for x
becomes		

Since the last expression is empty, all the expressions are proven.

(a) Rewrite the boolean expression

$\neg(\neg(a\wedge b)\wedge\neg(\neg a\wedge b)\wedge\neg(a\wedge\neg b)\wedge\neg(\neg a\wedge\neg b))$

as a bracket expression, and then prove it following the rules of bracket algebra.

(b) As directly as possible, rewrite the boolean expression

$(\neg a\Rightarrow\neg b)\ \wedge\ (a\ne b)\ \ \vee\ \ (a\wedge c\ \Rightarrow\ b\wedge c)$

as a bracket expression, and then prove it following the rules of bracket algebra.

(c) Can all boolean expressions be rewritten reasonably directly as bracket expressions?

(d) Can $x\,y$ become $y\,x$ using the rules of bracket algebra?

(e) Can all theorems of boolean algebra, rewritten reasonably directly as bracket expressions, be proven using the rules of bracket algebra?

(f) We interpret empty as $\top$, parentheses as negation, and juxtaposition as conjunction. Is there any other consistent way to interpret the symbols and rules of bracket algebra?

31 Let $\bullet$ be a two-operand infix operator (let's give it precedence 3) whose operands and result are of some type T . Let $\Diamond$ be a two-operand infix operator (let's give it precedence 7) whose operands are of type T and whose result is boolean, defined by the axiom

$a\Diamond b\ \ =\ \ a\bullet b=a$

(a) Prove if $\bullet$ is idempotent then $\Diamond$ is reflexive.

(b) Prove if $\bullet$ is associative then $\Diamond$ is transitive.

(c) Prove if $\bullet$ is symmetric then $\Diamond$ is antisymmetric.

(d) If T is the booleans and $\bullet$ is $\wedge$, what is $\Diamond$?

(e) If T is the booleans and $\bullet$ is $\vee$, what is $\Diamond$?

(f) If T is the natural numbers and $\Diamond$ is $\le$, what is $\bullet$?

(g) The axiom defines $\Diamond$ in terms of $\bullet$. Can it be inverted, so that $\bullet$ is defined in terms of $\Diamond$?

32 (family theory) Design a theory of personal relationships. Invent person expressions such as *Jack* , *Jill* , *father of p* , *mother of p* . Invent boolean expressions that use person expressions, such as *p is male* , *p is female* , *p is a parent of q* , *p is a son of q* , *p is a daughter of q* , *p is a child of q* , *p is married to q* , $p=q$. Invent axioms such as (*p is male*) ⧧ (*p is female*) . Formulate and prove an interesting theorem.

End of Basic Theories

10.2 Basic Data Structures

33 Simplify

(a) $(1, 7–3) + 4 – (2, 6, 8)$

(b) $nat×nat$

(c) $nat–nat$

(d) $(nat+1)×(nat+1)$

34 Prove $¬ 7: null$.

35 We defined bunch *null* with the axiom *null*: A . Is there any harm in defining bunch *all* with the axiom A: *all* ?

36 Let A be a bunch of booleans such that $A = ¬A$. What is A ?

37 Show that some of the axioms of Bunch Theory listed in Section 2.0 are provable from the other axioms. How many of the axioms can you remove without losing any theorems?

38 (hyperbunch) A hyperbunch is like a bunch except that each element can occur a number of times other than just zero times (absent) or one time (present). The order of elements remains insignificant. (A hyperbunch does not have a characteristic predicate, but a characteristic function with numeric result.) Design notations and axioms for each of the following kinds of hyperbunch.

(a) multibunch: an element can occur any natural number of times. For example, a multibunch can consist of one 2, two 7s, three 5s, and zero of everything else. (Note: the equivalent for sets is called either a multiset or a bag.)

(b) wholebunch: an element can occur any integer number of times.

(c) fuzzybunch: an element can occur any real number of times from 0 to 1 inclusive.

39 A composite number is a natural number with 2 or more (not necessarily distinct) prime factors. Express the composite numbers as simply as you can.

40 For this question only, let # be a two-operand infix operator (precedence 3) with natural operands and an extended natural result. Informally, $n\#m$ means "the number of times that n is a factor of m ". It is defined by the following two axioms.

$$m: n×nat \;\lor\; n\#m = 0$$

$$n≠0 \;\Rightarrow\; n\#(m×n) \;=\; n\#m + 1$$

(a) Make a 3×3 chart of the values of $(0,..3)\#(0,..3)$.

(b) Show that the axioms become inconsistent if the antecedent of the second axiom is removed.

(c) How should we change the axioms to allow # to have extended natural operands?

41 For naturals n and m, we can express the statement " n is a factor of m " formally as follows:

$$m: n \times nat$$

(a) What are the factors of 0 ?
(b) What is 0 a factor of?
(c) What are the factors of 1 ?
(d) What is 1 a factor of?

42 Let $B = 1, 3, 5$. What is
(a) $¢(B + B)$
(b) $¢(B \times 2)$
(c) $¢(B \times B)$
(d) $¢(B^2)$

43 The compound axiom says

$$x: A, B \;=\; x: A \;\lor\; x: B$$

There are 16 two-operand boolean operators that could sit where $\lor$ sits in this axiom if we just replace bunch union (,) by a corresponding bunch operator. Which of the 16 two-operand boolean operators correspond to useful bunch operators?

44 (von Neumann numbers)
(a) Is there any harm in adding the axioms

$0 = \{null\}$ the empty set
$n+1 = \{n, \sim n\}$ for each natural n

(b) What correspondence is induced by these axioms between the arithmetic operations and the set operations?
(c) Is there any harm in adding the axioms

$0 = \{null\}$ the empty set
$i+1 = \{i, \sim i\}$ for each integer i

45 (Cantor's paradise) Show that $¢⇂S > ¢S$ is neither a theorem nor an antitheorem.

46 The strings defined in Section 2.2 have natural indexes and extended natural lengths. Add a new operator, the inverse of catenation, to obtain strings that have negative indexes and lengths.

47 Prove the trichotomy for strings of numbers. For strings S and T, prove that exactly one of $S<T$, $S=T$, $S>T$ is a theorem.

48 In Section 2.3 there is a self-describing expression. Make it into a self-printing program. To do so, you need to know that $c!e$ outputs the value of expression e on channel c .

49 Simplify (no proof)
(a) $null, nil$
(b) $null; nil$
(c) $*nil$
(d) $[null]$
(e) $[*null]$

50 What is the difference between $[0, 1, 2]$ and $[0; 1; 2]$?

51 (prefix order) Give axioms to define the prefix partial order on strings. String S comes before string T in this order if and only if S is an initial segment of T .

52 Simplify, assuming i: 0,..#L

(a) $i\rightarrow Li \mid L$

(b) $L\ [0;..i] + [x] + L\ [i+1;..\#L]$

53 Simplify (no proof)

(a) $0\rightarrow 1 \mid 1\rightarrow 2 \mid 2\rightarrow 3 \mid 3\rightarrow 4 \mid 4\rightarrow 5 \mid [0;..5]$

(b) $(4\rightarrow 2 \mid [-3;..3])\ 3$

(c) $((3;2)\rightarrow[10;..15] \mid 3\rightarrow[5;..10] \mid [0;..5])\ 3$

(d) $([0;..5]\ [3;\ 4])\ 1$

(e) (2;2)→"j" | [["abc"]; ["de"]; ["fghi"]]

(f) $\#[nat]$

(g) $\#[*3]$

(h) $[3;\ 4]: [3*4*int]$

(i) $[3;\ 4]: [3;\ int]$

(j) $[3, 4;\ 5]: [2*int]$

(k) $[(3, 4);\ 5]: [2*int]$

(l) $[3;\ (4, 5);\ 6;\ (7, 8, 9)]$ ‘ $[3;\ 4;\ (5, 6);\ (7, 8)]$

54 Let i and j be indexes of list L . Express $i\rightarrow Lj \mid j\rightarrow Li \mid L$ without using $\mid$.

End of Basic Data Structures

10.3 Function Theory

55 In each of the following, replace p by

$\langle x: int\rightarrow\langle y: int\rightarrow\langle z: int\rightarrow x\geq 0 \wedge x^2\leq y \wedge \forall z: int\cdot\ z^2\leq y \Rightarrow z\leq x\rangle\rangle\rangle$

and simplify, assuming $x, y, z, u, w: int$.

(a) $p\ (x+y)\ (2\times u + w)\ z$

(b) $p\ (x+y)\ (2\times u + w)$

(c) $p\ (x+z)\ (y+y)\ (2+z)$

56 Some mathematicians like to use a notation like $\exists!x: D\cdot\ Px$ to mean “there is a unique x in D such that Px holds”. Define $\exists!$ formally.

57 Write an expression equivalent to each of the following without using § .

(a) ¢(§x: D· Px) = 0

(b) ¢(§x: D· Px) = 1

(c) ¢(§x: D· Px) = 2

58 (cat) Define function cat so that it applies to a list of lists and produces their catenation. For example,

$cat\ [[0;\ 1;\ 2];\ [nil];\ [[3]];\ [4;\ 5]]\ =\ [0;\ 1;\ 2;\ [3];\ 4;\ 5]$

59 Express formally that L is a sublist (not necessarily consecutive items) of list M . For example, $[0;\ 2;\ 1]$ is a sublist of $[0;\ 1;\ 2;\ 2;\ 1;\ 0]$, but $[0;\ 2;\ 1]$ is not a sublist of $[0;\ 1;\ 2;\ 3]$.

60 Express formally that L is a longest sorted sublist of M where
(a) the sublist must be consecutive items (a segment).
(b) the sublist must be consecutive (a segment) and nonempty.
(c) the sublist contains items in their order of appearance in M, but not necessarily consecutively (not necessarily a segment).

61 Express formally that natural n is the length of a longest palindromic segment in list L. A palindrome is a list that equals its reverse.

62 Using the syntax x **can fool** y **at time** t formalize the statements
(a) You can fool some of the people all of the time.
(b) You can fool all of the people some of the time.
(c) You can't fool all of the people all of the time.
for each of the following interpretations of the word “You”:
(i) Someone
(ii) Anyone
(iii) The person I am talking to

63 (whodunit) Here are ten statements.
(i) Some criminal robbed the Russell mansion.
(ii) Whoever robbed the Russell mansion either had an accomplice among the servants or had to break in.
(iii) To break in one would have to either smash the door or pick the lock.
(iv) Only an expert locksmith could pick the lock.
(v) Anyone smashing the door would have been heard.
(vi) Nobody was heard.
(vii) No one could rob the Russell mansion without fooling the guard.
(viii) To fool the guard one must be a convincing actor.
(ix) No criminal could be both an expert locksmith and a convincing actor.
(x) Some criminal had an accomplice among the servants.
(a) Choosing good abbreviations, translate each of these statements into formal logic.
(b) Taking the first nine statements as axioms, prove the tenth.

64 (arity) The arity of a function is the number of variables (parameters) it introduces, and the number of arguments it can be applied to. Write axioms to define αf (arity of f).

65 There are some people, some keys, and some doors. Let p *holds* k mean that person p holds key k. Let k *unlocks* d mean that key k unlocks door d. Let p *opens* d mean that person p can open door d. Formalize
(a) Anyone can open any door if they have the appropriate key.
(b) At least one door can be opened without a key (by anyone).
(c) The locksmith can open any door even without a key.

66 Prove that if variables i and j do not appear in predicates P and Q, then

$$(\forall i\cdot Pi) \Rightarrow (\exists i\cdot Qi) \;=\; (\exists i, j\cdot Pi \Rightarrow Qj)$$

67 There are four boolean two-operand associative symmetric operators with an identity. We used two of them to define quantifiers. What happened to the other two?

68 Which operator can be used to define a quantifier to give the range of a function?

69 We have defined several quantifiers by starting with an associative symmetric operator with an identity. Bunch union is also such an operator. Does it yield a quantifier?

70 Exercise 13 talks about drinking and driving, but not about time. It's not all right to drink first and then drive soon after, but it is all right to drive first and then drink soon after. It is also all right to drink first and then drive 6 hours after. Let *drink* and *drive* be predicates of time, and formalize the rule that you can't drive for 6 hours after drinking. What does your rule say about drinking and driving at the same time?

71 Formalize each of the following statements as a boolean expression.
(a) Everybody loves somebody sometime.
(b) Every 10 minutes someone in New York City gets mugged.
(c) Every 10 minutes someone keeps trying to reach you.
(d) Whenever the altititude is below 1000 feet, the landing gear must be down.
(e) I'll see you on Tuesday, if not before.
(f) No news is good news.

72 Express formally that
(a) natural n is the largest proper (neither 1 nor m) factor of natural m .
(b) g is the greatest common divisor of naturals a and b .
(c) m is the lowest common multiple of naturals a and b .
(d) p is a prime number.
(e) n and m are relatively prime numbers.
(f) there is at least one and at most a finite number of naturals satisfying predicate p .
(g) there is no smallest integer.
(h) between every two rational numbers there is another rational number.
(i) list L is a longest segment of list M that does not contain item x .
(j) the segment of list L from (including) index i to (excluding) index j is a segment whose sum is smallest.
(k) a and b are items of lists A and B (respectively) whose absolute difference is least.
(l) p is the length of a longest plateau (segment of equal items) in a nonempty sorted list L .
(m) all items that occur in list L occur in a segment of length 10 .
(n) all items of list L are different (no two items are equal).
(o) at most one item in list L occurs more than once.
(p) the maximum item in list L occurs m times.
(q) list L is a permutation of list M .

73 (bitonic list) A list is bitonic if it is monotonic up to some index, and antimonotonic after that. For example, $[1; 3; 4; 5; 5; 6; 4; 4; 3]$ is bitonic. Express formally that L is bitonic.

74 Formalize and disprove the statement “There is a natural number that is not equal to any natural number.”.

75 (friends) Formalize and prove the statement “The people you know are those known by all who know all whom you know.”.

76 (swapping partners) There is a finite bunch of couples. Each couple consists of a man and a woman. The oldest man and the oldest woman have the same age. If any two couples swap partners, forming two new couples, the younger partners of the two new couples have the same age. Prove that in each couple, the partners have the same age.

77 Express $\forall$ and $\exists$ in terms of $¢$ and $§$.

78 Simplify

(a) $\Sigma ((0,..n) \to m)$

(b) $\Pi ((0,..n) \to m)$

(c) $\forall ((0,..n) \to b)$

(d) $\exists ((0,..n) \to b)$

79 Are the boolean expressions

$nil \to x \ = \ x$

$(S;T) \to x \ = \ S \to T \to x$

(a) consistent with the theory in Chapters 2 and 3?

(b) theorems according to the theory in Chapters 2 and 3?

80 (unicorns) The following statements are made.

All unicorns are white.

All unicorns are black.

No unicorn is both white and black.

Are these statements consistent? What, if anything, can we conclude about unicorns?

81 (Russell's barber) Bertrand Russell stated: “In a small town there is a male barber who shaves the men in the town who do not shave themselves.”. Then Russell asked: “Does the barber shave himself?”. If we say yes, then we can conclude from the statement that he does not, and if we say no, then we can conclude from the statement that he does. Formalize this paradox, and thus explain it.

82 (Russell's paradox) Define $rus \ = \ \langle f \colon (null \to bool) \to \neg\, f f \rangle$.

(a) Can we prove $rus\ rus \ = \ \neg\ rus\ rus$?

(b) Is this an inconsistency?

(c) Can we add the axiom $\neg\ f \colon \Delta f$? Would it help?

83 Prove that the square of an odd natural number is odd, and the square of an even natural number is even.

84 (Gödel/Turing incompleteness) Prove that we cannot consistently and completely define an interpreter. An interpreter is a predicate $\mathcal{I}$ that applies to texts; when applied to a text representing a boolean expression, its result is equal to the represented expression. For example,

$\mathcal{I}\ "\forall s\colon [*char]\cdot \#s \ge 0" \ = \ \forall s\colon [*char]\cdot \#s \ge 0$

85 Let f and g be functions from nat to nat . For what f do we have the theorem $g f = g$? For what f do we have the theorem $f g = g$?

86 What is the difference between $\#[n*\top]$ and $¢§[n*\top]$?

87 Without using the Bounding Laws, prove

(a) $\forall i\cdot Li \le m \ = \ (MAX\ L) \le m$

(b) $\exists i\cdot Li \le m \ = \ (MIN\ L) \le m$

88 (pigeon-hole) Prove $(\Sigma L) > n \times \#L \ \Rightarrow \ \exists i\colon \Delta L\cdot Li > n$.

89 If $f: A \to B$ and $p: B \to bool$, prove

(a) $\exists b: fA \cdot pb \;=\; \exists a: A \cdot pfa$

(b) $\forall b: fA \cdot pb \;=\; \forall a: A \cdot pfa$

90 This question explores a simpler, more elegant function theory than the one presented in Chapter 3. We separate the notion of local variable introduction from the notion of domain, and we generalize the latter to become local axiom introduction. Variable introduction has the form $\langle v \to b \rangle$ where v is a variable and b is any expression (the body; no domain). There is an Application Law

$\langle v \to b \rangle\, x = $ (substitute x for v in b)

and an Extension Law

$f = \langle v \to fv \rangle$

Let a be boolean, and let b be any expression. Then $a \gg b$ is an expression of the same type as b . The $\gg$ operator has precedence level 12 and is right-associating. Its axioms include:

$\top \gg b \;=\; b$

$a \gg b \gg c \;=\; a \land b \gg c$

The expression $a \gg b$ is a “one-tailed if-expression”, or “asserted expression”; it introduces a as a local axiom within b . A function is a variable introduction whose body is an asserted expression in which the assertion has the form $v: D$. In this case, we allow an abbreviation: for example, the function $\langle n \to n: nat \gg n+1 \rangle$ can be abbreviated $\langle n: nat \to n+1 \rangle$. Applying this function to 3 , we find

$\langle n \to n: nat \gg n+1 \rangle\; 3$

$=\;\; 3: nat \gg 3+1$

$=\;\; \top \gg 4$

$=\;\; 4$

Applying it to -3 we find

$\langle n \to n: nat \gg n+1 \rangle\; (-3)$

$=\;\; -3: nat \gg -3+1$

$=\;\; \bot \gg -2$

and then we are stuck; no further axiom applies. In the example, we have used variable introduction and axiom introduction together to give us back the kind of function we had; but in general, they are independently useful.

(a) Show how function-valued variables can be introduced in this new theory.

(b) What expressions in the old theory have no equivalent in the new? How closely can they be approximated?

(c) What expressions in the new theory have no equivalent in the old? How closely can they be approximated?

91 Is there any harm in defining relation R with the following axioms?

$\forall x \cdot \exists y \cdot Rxy$	totality
$\forall x \cdot \neg Rxx$	irreflexivity
$\forall x, y, z \cdot Rxy \land Ryz \Rightarrow Rxz$	transitivity
$\exists u \cdot \forall x \cdot x{=}u \lor Rxu$	unity

92 Let n be a natural number, and let R be a relation on $0,..n$. In other words,

$R: (0,..n) \to (0,..n) \to bool$

We say that from x we can reach x in zero steps. If Rxy we say that from x we can reach y in one step. If Rxy and Ryz we say that from x we can reach z in two steps. And so on. Express formally that from x we can reach y in some number of steps.

93 Relation R is transitive if $\forall x, y, z \cdot Rxy \wedge Ryz \Rightarrow Rxz$. Express formally that relation R is the transitive closure of relation Q (R is the strongest transitive relation that is implied by Q).

End of Function Theory

10.4 Program Theory

94 Prove specification S is satisfiable for prestate σ if and only if $S.\top$ (note: $\top$ is the “true” boolean).

95 Let x be an integer state variable. Which of the following specifications are implementable?
(a) $x \ge 0 \Rightarrow x'^2 = x$
(b) $x' \ge 0 \Rightarrow x = 0$
(c) $\neg(x \ge 0 \wedge x' = 0)$
(d) $\neg(x \ge 0 \vee x' = 0)$

96 A specification is transitive if, for all states a , b , and c , if it allows the state to change from a to b , and it allows the state to change from b to c , then it allows the state to change from a to c . Prove S is transitive if and only if S is refined by $S.S$.

97√ Simplify each of the following (in integer variables x and y).
(a) $x:= y+1.\ y'>x'$
(b) $x:= x+1.\ y'>x \wedge x'>x$
(c) $x:= y+1.\ y'=2x$
(d) $x:= 1.\ x\ge1 \Rightarrow \exists x\cdot y'=2x$
(e) $x:= y.\ x\ge1 \Rightarrow \exists y\cdot y'=x\times y$
(f) $x:= 1.\ ok$
(g) $x:= 1.\ y:= 2$
(h) $x:= 1.\ P$ where $P = y:= 2$
(i) $x:= 1.\ y:= 2.\ x:= x+y$
(j) $x:= 1.$ **if** $y>x$ **then** $x:= x+1$ **else** $x:= y$
(k) $x:= 1.\ x'>x.\ x'=x+1$

98 Prove
(a) $x:= x = ok$
(b) $x:= e.\ x:= f\,x = x:= f\,e$

99 Prove or disprove
(a) $R.$ **if** b **then** P **else** Q $=$ **if** b **then** $(R.\ P)$ **else** $(R.\ Q)$
(b) **if** b **then** $P\Rightarrow Q$ **else** $R\Rightarrow S$ $=$ (**if** b **then** P **else** R) $\Rightarrow$ (**if** b **then** Q **else** S)
(c) **if** b **then** $(P.\ Q)$ **else** $(R.\ S)$ $=$ **if** b **then** P **else** $R.$ **if** b **then** Q **else** S

100 Prove
(a) P and Q are each refined by R if and only if their conjunction is refined by R .
(b) $P\Rightarrow Q$ is refined by R if and only if Q is refined by $P\wedge R$.

101 (rolling)
(a) Can we always unroll a loop? If $S \Leftarrow A.\ S.\ Z$, can we conclude $S \Leftarrow A.\ A.\ S.\ Z.\ Z$?
(b) Can we always roll up a loop? If $S \Leftarrow A.\ A.\ S.\ Z.\ Z$, can we conclude $S \Leftarrow A.\ S.\ Z$?

102 What is wrong with the following proof:

$$\begin{array}{lll} & (R \Leftarrow R.\,S) & \text{use context rule} \\ = & (R \Leftarrow \bot.\,S) & \bot \text{ is base for } . \\ = & (R \Leftarrow \bot) & \text{base law for } \Leftarrow \\ = & \top & \end{array}$$

103 For which kinds of specifications P and Q is the following a theorem:

(a) $\neg(P.\,\neg Q) \Leftarrow P.\,Q$

(b) $P.\,Q \Leftarrow \neg(P.\,\neg Q)$

(c) $P.\,Q = \neg(P.\,\neg Q)$

104 Write a formal specification of the following problem: “Change the value of list variable L so that each item is repeated. For example, if L is $[6; 3; 5; 5; 7]$ then it should be changed to $[6; 6; 3; 3; 5; 5; 5; 5; 7; 7]$.”.

105 Let P and Q be specifications. Let C be a precondition and let C' be the corresponding postcondition. Prove the condition law

$$P.\,Q \Leftarrow P{\wedge}C'.\; C{\Rightarrow}Q$$

106 Let P and Q be specifications. Let C be a precondition and let C' be the corresponding postcondition. Which three of the following condition laws can be turned around, switching the problem and the solution?

$$C \wedge (P.\,Q) \Leftarrow C{\wedge}P.\,Q$$
$$C \Rightarrow (P.Q) \Leftarrow C{\Rightarrow}P.\,Q$$
$$(P.Q) \wedge C' \Leftarrow P.\,Q{\wedge}C'$$
$$(P.Q) \Leftarrow C' \Leftarrow P.\,Q{\Leftarrow}C'$$
$$P.\,C{\wedge}Q \Leftarrow P{\wedge}C'.\,Q$$
$$P.\,Q \Leftarrow P{\wedge}C'.\; C{\Rightarrow}Q$$

107 Let S be a specification. Let C be a precondition and let C' be the corresponding postcondition. How does the exact precondition for C' to be refined by S differ from $(S.\,C)$? Hint: consider prestates in which S is unsatisfiable, then deterministic, then nondeterministic.

108 We have Refinement by Steps, Refinement by Parts, and Refinement by Cases. In this question we propose Refinement by Alternatives:

If $A \Leftarrow$ **if** b **then** C **else** D and $E \Leftarrow$ **if** b **then** F **else** G are theorems,
then $A{\vee}E \Leftarrow$ **if** b **then** $C{\vee}F$ **else** $D{\vee}G$ is a theorem.

If $A \Leftarrow B.C$ and $D \Leftarrow E.F$ are theorems, then $A{\vee}D \Leftarrow B{\vee}E.\,C{\vee}F$ is a theorem.

If $A \Leftarrow B$ and $C \Leftarrow D$ are theorems, then $A{\vee}C \Leftarrow B{\vee}D$ is a theorem.

Discuss the merits and demerits of this proposed law.

109 Let x and y be real variables. Prove that if $y{=}x^2$ is true before

$$x{:=}\;x{+}1.\;\; y{:=}\;y + 2{\times}x - 1$$

is executed, then it is still true after.

110√ In one integer variable x ,

(a) find the exact precondition A for $x'{>}5$ to be refined by $x{:=}\;x{+}1$.

(b) find the exact postcondition for A to be refined by $x{:=}\;x{+}1$, where A is your answer from part (a).

111 Let all variables be integer except L is a list of integers. What is the exact precondition for

(a) $x'+y' > 8$ to be refined by $x:= 1$

(b) $x'=1$ to be refined by $x:= 1$

(c) $x'=2$ to be refined by $x:= 1$

(d) $x'=y$ to be refined by $y:= 1$

(e) $x' \ge y'$ to be refined by $x:= y+z$

(f) $y'+z' \ge 0$ to be refined by $x:= y+z$

(g) $x' \le 1 \lor x' \ge 5$ to be refined by $x:= x+1$

(h) $x'<y' \land \exists x\cdot Lx<y'$ to be refined by $x:= 1$

(i) $\exists y\cdot Ly<x'$ to be refined by $x:= y+1$

(j) $L'\ 3 = 4$ to be refined by $L:= i\to 4 \mid L$

(k) $x'=a$ to be refined by **if** $a > b$ **then** $x:= a$ **else** ok

(l) $x'=y \land y'=x$ to be refined by $(z:= x.\ x:= y.\ y:= z)$

(m) $a\times x'^2 + b\times x' + c = 0$ to be refined by $(x:= a\times x + b.\ x:= -x/a)$

(n) $f' = n'!$ to be refined by $(n:= n+1.\ f:= f\times n)$ where n is natural and $!$ is factorial.

(o) $7 \le c' < 28 \land odd\ c'$ to be refined by $(a:= b-1.\ b:= a+3.\ c:= a+b)$

(p) $s' = \Sigma L\ [0;..i']$ to be refined by $(s:= s + Li.\ i:= i+1)$

112 For what exact precondition and postcondition does the following assignment move integer variable x farther from zero?

(a) $x:= x+1$

(b) $x:= abs\ (x+1)$

(c)√ $x:= x^2$

113 For what exact precondition and postcondition does the following assignment move integer variable x farther from zero staying on the same side of zero?

(a) $x:= x+1$

(b) $x:= abs\ (x+1)$

(c) $x:= x^2$

114 Prove

(a) the Precondition Law: C is a sufficient precondition for specification P to be refined by specification S if and only if $C \Rightarrow P$ is refined by S .

(b) the Postcondition Law: C' is a sufficient postcondition for specification P to be refined by specification S if and only if $C' \Rightarrow P$ is refined by S .

115 (weakest prespecification, weakest postspecification) Given specifications P and Q , find the weakest specification S (in terms of P and Q) such that P is refined by

(a) $S.\ Q$

(b) $Q.\ S$

116 Let a , b , and c be integer variables. Simplify

(a) $b:= a-b.\ b:= a-b$

(b) $a:= a+b.\ b:= a-b.\ a:= a-b$

(c) $c:= a-b-c.\ b:= a-b-c.\ a:= a-b-c.\ c:= a+b+c$

117 Let x and y be boolean variables. Simplify

(a) $x:= x=y.\ x:= x=y$

(b) $x:= x\ne y.\ y:= x\ne y.\ x:= x\ne y$

118 Let x be an integer variable. Prove the refinement

(a)√ $x'=0 \Leftarrow$ **if** $x=0$ **then** ok **else** $(x:= x-1.\ x'=0)$

(b) $P \Leftarrow$ **if** $x=0$ **then** ok **else** $(x:= x-1.\ t:= t+1.\ P)$

where $P = x'=0 \land$ **if** $x\ge 0$ **then** $t' = t+x$ **else** $t'=\infty$

119 Let x be an integer variable. Prove the refinement

(a) $x'=1 \Leftarrow$ **if** $x=1$ **then** ok **else** $(x:= div\ x\ 2.\ x'=1)$

(b)√ $R \Leftarrow$ **if** $x=1$ **then** ok **else** $(x:= div\ x\ 2.\ t:= t+1.\ R)$

where $R = x'=1 \land$ **if** $x\ge 1$ **then** $t' \le t + log\ x$ **else** $t'=\infty$

120 In natural variables s and n prove

$P \Leftarrow$ **if** $n=0$ **then** ok **else** $(n:= n-1.\ s:= s+2^n-n.\ t:= t+1.\ P)$

where $P = s' = s + 2^n - n\times(n-1)/2 - 1 \land n'=0 \land t' = t+n$.

121 Let x be an integer variable. Is the refinement

$P \Leftarrow$ **if** $x=0$ **then** ok **else** $(x:= x-1.\ t:= t+1.\ P)$

a theorem when

$P = x<0 \Rightarrow x'=1 \land t'=\infty$

Is this reasonable? Explain.

122 (factorial) In natural variables n and f prove

$f:= n! \Leftarrow$ **if** $n=0$ **then** $f:= 1$ **else** $(n:= n-1.\ f:= n!.\ n:= n+1.\ f:= f\times n)$

where $n! = 1\times 2\times 3\times ...\times n$.

123 In natural variables n and m prove

$P \Leftarrow$ $n:= n+1.$
if $n=10$ **then** ok
else $(m:= m-1.\ P)$

where $P = m:= m+n-9.\ n:= 10$.

124 Let x and n be natural variables. Find a specification P such that both the following hold:

$x = x'\times 2^{n'} \Leftarrow n:= 0.\ P$

$P \Leftarrow$ **if** $even\ x$ **then** $(x:= x/2.\ n:= n+1.\ P)$ **else** ok

125 (square) Let s and n be natural variables. Find a specification P such that both the following hold:

$s' = n^2 \Leftarrow s:= n.\ P$

$P \Leftarrow$ **if** $n=0$ **then** ok **else** $(n:= n-1.\ s:= s+n+n.\ P)$

This program squares using only addition, subtraction, and test for zero.

126 Let a and b be positive integers. Let x , u , and v be integer variables. Let

$P = u\ge 0 \land v\ge 0 \land x = u\times a - v\times b \Rightarrow x'=0$

(a) Prove

$P \Leftarrow$ **if** $x>0$ **then** $(x:= x-a.\ u:= u-1.\ P)$
else if $x<0$ **then** $(x:= x+b.\ v:= v-1.\ P)$
else ok

(b) Find an upper bound for the execution time of the program in part (a).

127 Let i be an integer variable. Add time according to the recursive measure, and then find the strongest P you can such that

(a) $P \Leftarrow$ **if** *even* i **then** $i:= i/2$ **else** $i:= i+1$.
if $i=1$ **then** *ok* **else** P

(b) $P \Leftarrow$ **if** *even* i **then** $i:= i/2$ **else** $i:= i-3$.
if $i=0$ **then** *ok* **else** P

128 Find a finite function f of natural variables i and j to serve as an upper bound on the execution time of the following program, and prove

$t' \le t + fij \Leftarrow$ **if** $i=0 \wedge j=0$ **then** *ok*
else if $i=0$ **then** $(i:= j\times j.\ j:= j-1.\ t:= t+1.\ t' \le t + fij)$
else $(i:= i-1.\ t:= t+1.\ t' \le t + fij)$

129 Let P mean that the final values of natural variables a and b are the largest exponents of 2 and 3 respectively such that both powers divide evenly into the initial value of positive integer x.

(a) Define P formally.

(b) Define Q suitably and prove

$P \Leftarrow a:= 0.\ b:= 0.\ Q$

$Q \Leftarrow$ **if** $x: 2\times nat$ **then** $(x:= x/2.\ a:= a+1.\ Q)$
else if $x: 3\times nat$ **then** $(x:= x/3.\ b:= b+1.\ Q)$
else *ok*

(c) Find an upper bound for the execution time of the program in part (b).

130 Express formally that specification R is satisfied by any number (including 0) of repetitions of behavior satisfying specification S.

131 (Zeno) Here is a loop.

$R \Leftarrow x:= x+1.\ R$

Suppose we charge time 2^{-x} for the recursive call, so that each iteration takes half as long as the one before. Prove that the execution time is finite.

132 Let t be the time variable. Can we prove the refinement

$P \Leftarrow t:= t+1.\ P$

for $P = t'=5$? Does this mean that execution will terminate at time 5 ? What is wrong?

133 Let n and r be natural variables in the refinement

$P \Leftarrow$ **if** $n=1$ **then** $r:= 0$ **else** $(n:= div\ n\ 2.\ P.\ r:= r+1)$

Suppose the operations div and $+$ each take time 1 and all else is free (even the call is free). Insert appropriate time increments, and find an appropriate P to express the execution time in terms of

(a) the initial values of the memory variables. Prove the refinement for your choice of P.

(b) the final values of the memory variables. Prove the refinement for your choice of P.

134 (running total) Given list variable L and any other variables you need, write a program to convert L into a list of cumulative sums. Formally,

(a) $\forall n: 0,..\#L\cdot L'n = \Sigma L\ [0;..n]$

(b) $\forall n: 0,..\#L\cdot L'n = \Sigma L\ [0;..n+1]$

135 (cube) Write a program that cubes using only addition, subtraction, and test for zero.

136 (cube test) Write a program to determine if a given natural number is a cube without using exponentiation.

137 ($mod\ 2$) Let n be a natural variable. The problem to reduce n modulo 2 can be solved as follows:

$$n' = mod\ n\ 2 \;\Leftarrow\; \textbf{if } n<2 \textbf{ then } ok \textbf{ else } (n:= n-2.\ \ n' = mod\ n\ 2)$$

Using the recursive time measure, find and prove an upper time bound. Make it as small as you can.

138 (fast $mod\ 2$) Let n and p be natural variables. The problem to reduce n modulo 2 can be solved as follows:

$$n' = mod\ n\ 2 \;\Leftarrow\; \textbf{if } n<2 \textbf{ then } ok \textbf{ else } (even\ n' = even\ n.\ \ n' = mod\ n\ 2)$$

$$even\ n' = even\ n \;\Leftarrow\; p:= 2.\ \ even\ p \;\Rightarrow\; even\ p' \;\land\; even\ n' = even\ n$$

$$even\ p \;\Rightarrow\; even\ p' \;\land\; even\ n' = even\ n \;\Leftarrow$$

$$n:= n-p.\ \ p:= p+p.$$

$$\textbf{if } n<p \textbf{ then } ok \textbf{ else } even\ p \;\Rightarrow\; even\ p' \;\land\; even\ n' = even\ n$$

(a) Prove these refinements.

(b) Using the recursive time measure, find and prove a sublinear upper time bound.

139 Given a specification P and a prestate σ with t as time variable, we might define “the exact precondition for termination” as follows:

$$\exists n\text{: } nat\cdot\ \forall\sigma'\cdot\ t' \le t+n \;\Leftarrow\; P$$

Letting x be an integer variable, find the exact precondition for termination of the following, and comment on whether it is reasonable.

(a) $x \ge 0 \;\Rightarrow\; t' \le t+x$

(b) $\exists n\text{: } nat\cdot\ t' \le t+n$

(c) $\exists f\text{: } int\to nat\cdot\ t' \le t + fx$

140√ (maximum item) Write a program to find the maximum item in a list.

141 (list comparison) Using item comparison but not list comparison, write a program to determine whether one list comes before another in the list order.

142√ (list summation) Write a program to find the sum of a list of numbers.

143 (alternating sum) Write a program to find the alternating sum $L0 - L1 + L2 - L3 + ...$ of a list L of numbers.

144 (combinations) Write a program to find the number of ways to partition $a+b$ things into a things in the left part and b things in the right part. Include recursive time.

145 (earliest meeting time) Write a program to find the earliest meeting time acceptable to three people. Each person is willing to state their possible meeting times by means of a function that tells, for each time t, the earliest time at or after t that they are available for a meeting. (Do not confuse this t with the execution time variable. You may ignore execution time for this problem.)

146 (polynomial) You are given $n\text{: } nat$, $c\text{: } [n*rat]$, $x\text{: } rat$ and variable $y\text{: } rat$. c is a list of coefficients of a polynomial (“of degree $n-1$”) to be evaluated at x . Write a program for

$$y' = \Sigma i\text{: } 0,..n\cdot\ ci\times x^i$$

147 (multiplication table) Given n: *nat* and variable M: [*[*nat]] , write a program to assign to M a multiplication table of size n without using multiplication. For example, if $n = 4$, then

M' = [[0];
[0; 1];
[0; 2; 4];
[0; 3; 6; 9]]

148 (Pascal's triangle) Given n: *nat* and variable P: [*[*nat]] , write a program to assign to P a Pascal's triangle of size n . For example, if $n = 4$, then

P' = [[1];
[1; 1];
[1; 2; 1];
[1; 3; 3; 1]]

The left side and diagonal are all 1s; each interior item is the sum of the item above it and the item diagonally above and left.

149√ (binary exponentiation) Given natural variables x and y , write a program for $y' = 2^x$ without using exponentiation.

150 Write a program to find the smallest power of 2 that is bigger than or equal to a given positive integer without using exponentiation.

151√ (fast exponentiation) Given rational variables x and z and natural variable y , write a program for $z' = x^y$ that runs fast without using exponentiation.

152 (sort test) Write a program to assign a boolean variable to indicate whether a given list is sorted.

153√ (linear search) Write a program to find the first occurrence of a given item in a given list. The execution time must be linear in the length of the list.

154√ (binary search) Write a program to find a given item in a given nonempty sorted list. The execution time must be logarithmic in the length of the list. The strategy is to identify which half of the list contains the item if it occurs at all, then which quarter, then which eighth, and so on.

155 (binary search with test for equality) The problem is binary search (Exercise 154), but each iteration tests to see if the item in the middle of the remaining segment is the item we seek.
(a) Write the program, with specifications and proofs.
(b) Find the execution time according to the recursive measure.
(c) Find the execution time according to a measure that charges time 1 for each test.
(d) Compare the execution time to binary search without the test for equality each iteration.

156 (ternary search) The problem is the same as binary search (Exercise 154). The strategy this time is to identify which third of the list contains the item if it occurs at all, then which ninth, then which twenty-seventh, and so on.

157√ (two-dimensional search) Write a program to find a given item in a given 2-dimensional array. The execution time must be linear in the product of the dimensions.

158 (sorted two-dimensional search) Write a program to find a given item in a given 2-dimensional array in which each row is sorted and each column is sorted. The execution time must be linear in the sum of the dimensions.

159 (sorted two-dimensional count) Write a program to count the number of occurrences of a given item in a given 2-dimensional array in which each row is sorted and each column is sorted. The execution time must be linear in the sum of the dimensions.

160 (pattern search) Let *subject* and *pattern* be two texts. Write a program to do the following. If *pattern* occurs somewhere within *subject*, natural variable h is assigned to indicate the beginning of its first occurrence

(a) using any list operators given in Section 2.3.

(b) using list indexing, but no other list operators.

161 (fixed point) Let L be a nonempty sorted list of n different integers. Write a program to find a fixed-point of L, that is an index i such that $Li = i$, or to report that no such index exists. Execution time should be at most $log\ n$ where n is the length of the list.

162 (all present) Given a natural number and a list, write a program to determine if every natural number up to the given number is an item in the list.

163 (missing number) You are given an unsorted list of length n whose items are the numbers $0,..n+1$ with one number missing. Write a program to find the missing number.

164 (text length) You are given a text (string of characters) that begins with zero or more "ordinary" characters, and then ends with zero or more "padding" characters. A padding character is not an ordinary character. Write a program to find the number of ordinary characters in the text. Execution time should be logarithmic in the text length.

165 (ordered pair search) Given a list of at least two items whose first item is less than or equal to its last item, write a program to find an adjacent pair of items such that the first of the pair is less than or equal to the second of the pair. Execution time should be logarithmic in the length of the list.

166 (convex equal pair) A list of numbers is convex if its length is at least 2, and every item (except the first and last) is less than or equal to the average of its two neighbors. Given a convex list, write a program to determine if it has a pair of consecutive equal items. Execution should be logarithmic in the length of the list.

167 Define a partial order « on pairs of integers as follows:

$$[a; b] \ll [c; d] \;=\; a<c \wedge b<d$$

Given n: nat+1 and L: [n*[int; int]] write a program to find the index of a minimal item in L. That is, find j: $0,..\#L$ such that $\neg\exists i\cdot Li \ll Lj$. The execution time should be at most $n \times log\ n$.

168 (n sort) Given a list L such that $L\,(0,..\#L) = 0,..\#L$, write a program to sort L in linear time and constant space. The only change permitted to L is to swap two items.

169√ (n^2 sort) Write a program to sort a list. Execution time should be at most n^2 where n is the length of the list.

170 ($n \times log\ n$ sort) Write a program to sort a list. Execution time should be at most $n \times log\ n$ where n is the length of the list.

171 (reverse) Write a program to reverse the order of the items of a list.

172 (next sorted list) Given a nonempty sorted list of naturals, write a program to find the next (in list order) sorted list having the same length and sum.

173 (next combination) You are given a sorted list of m different numbers, all in the range $0,..n$. Write a program to find the lexicographically next sorted list of m different numbers, all in the range $0,..n$.

174 (next permutation) You are given a list of the numbers $0,..n$ in some order. Write a program to find the lexicographically next list of the numbers $0,..n$.

175 (permutation inverse) You are given a list variable P of different items in $0,..\#P$. Write a program for $P\ P' = [0;..\#P]$.

176 (idempotent permutation) You are given a list variable L of items in $0,..\#L$ (not necessarily all different). Write a program to permute the list so that finally $L'\ L' = L'$.

177 (local minimum) You are given a list L of at least 3 numbers such that $L0 \ge L1$ and $L(\#L-2) \le L(\#L-1)$. A local minimum is an interior index i: $1,..\#L-1$ such that

$$L(i-1) \ge Li \le L(i+1)$$

Write a program to find a local minimum of L .

178 (natural division) The natural quotient of natural n and positive integer p is the natural number q satisfying

$$q \le n/p < q+1$$

Write a program to find the natural quotient of n and p in $log\ n$ time without using any functions (*div* , *mod* , *floor* , *ceil* , ...).

179 (remainder) Write a program to find the remainder after natural division (Exercise 178), using only comparison, addition, and subtraction (not multiplication or division or *mod*).

180 (natural binary logarithm) The natural binary logarithm of a positive integer p is the natural number b satisfying

$$2^b \le p < 2^{b+1}$$

Write a program to find the natural binary logarithm of a given positive integer p in $log\ p$ time.

181 (natural square root) The natural square root of a natural number n is the natural number s satisfying

$$s^2 \le n < (s+1)^2$$

(a) Write a program to find the natural square root of a given natural number n in $log\ n$ time.

(b) Write a program to find the natural square root of a given natural number n in $log\ n$ time using only addition, subtraction, doubling, halving, and comparisons (no multiplication or division).

182 (factor count) Write a program to find the number of factors (not necessarily prime) of a given natural number.

183 (Fermat's last program) Given natural c, write a program to find the number of unordered pairs of naturals a and b such that $a^2 + b^2 = c^2$ in time proportional to c. (An unordered pair is really a bunch of size 1 or 2. If we have counted the pair a and b, we don't want to count the pair b and a.) Your program may use addition, subtraction, multiplication, division, and comparisons, but not exponentiation or square root.

184 (flatten) Write a program to flatten a list. The result is a new list just like the old one but without the internal structure. For example,

L = [[3; 5]; 2; [5; [7]; [*nil*]]]
L' = [3; 5; 2; 5; 7]

Your program may employ a test Li: *int* to see if an item is an integer or a list.

185 (diagonal) Some points are arranged around the perimeter of a circle. The distance from each point to the next point going clockwise around the perimeter is given by a list. Write a program to find two points that are farthest apart.

186 (minimum sum segment) Given a list of integers, possibly including negatives, write a program to find

(a)√ the minimum sum of any segment (sublist of consecutive items).

(b) the segment (sublist of consecutive items) whose sum is minimum.

187 (maximum product segment) Given a list of integers, possibly including negatives, write a program to find

(a) the maximum product of any segment (sublist of consecutive items).

(b) the segment (sublist of consecutive items) whose product is maximum.

188 (segment sum count)

(a) Write a program to find, in a given list of naturals, the number of segments whose sum is a given natural.

(b) Write a program to find, in a given list of positive naturals, the number of segments whose sum is a given natural.

189 (longest plateau) You are given a nonempty sorted list of numbers. A plateau is a segment (sublist of consecutive items) of equal items. Write a program to find

(a) the length of a longest plateau.

(b) the number of longest plateaus.

190 (longest smooth segment) In a list of integers, a smooth segment is a sublist of consecutive items in which no two adjacent items differ by more than 1. Write a program to find a longest smooth segment.

191 (longest balanced segment) Given a list of booleans, write a program to find a longest segment (sublist of consecutive items) having an equal number of $\top$ and $\bot$ items.

192 (longest palindrome) A palindrome is a list that equals its reverse. Write a program to find a longest palindromic segment in a given list.

193 (greatest subsequence) Given a list, write a program to find the sublist that is largest according to list order. (A sublist contains items drawn from the list, in the same order of appearance, but not necessarily consecutive items.)

194 Given a list whose items are all 0 , 1 , or 2 , write a program

(a) to find the length of a shortest segment (consecutive items) that contains all three numbers in any order.

(b) to count the number of sublists (not necessarily consecutive items) that are 0 then 1 then 2 in that order.

195 Let L and M be sorted lists of numbers. Write a program to find the number of pairs of indexes i: 0,..$\#L$ and j: 0,..$\#M$ such that $Li \le Mj$.

196 (heads and tails) Let L be a list of positive integers. Write a program to find the number of pairs of indexes i and j such that

$$\Sigma\, L\, [0;..i] \;=\; \Sigma\, L\, [j;..\#L]$$

197 (pivot) You are given a nonempty list of positive numbers. Write a program to find the balance point, or pivot. Each item contributes its value (weight) times its distance from the pivot to its side of the balance. Item i is considered to be located at point $i + 1/2$, and the pivot point may likewise be noninteger.

198 (inversion count) Given a list, write a program to find how many pairs of items (not necessarily consecutive items) are out of order, with the larger item before the smaller item.

199 (minimum difference) Given two nonempty sorted lists of numbers, write a program to find a pair of items, one from each list, whose absolute difference is smallest.

200 (earliest quitter) In a nonempty list find the first item that is not repeated later. In list [13; 14; 15; 14; 15; 13] the earliest quitter is 14 because the other items 13 and 15 both occur after the last occurrence of 14 .

201 (interval union) A collection of intervals along a real number line is given by the list of left ends L and the corresponding list of right ends R . List L is sorted. The intervals might sometimes overlap, and sometimes leave gaps. Write a program to find the total length of the number line that is covered by these intervals.

202 (bit sum) Write a program to find the number of ones in the binary representation of a given natural number.

203 (digit sum) Write a program to find the sum of the digits in the decimal representation of a given natural number.

204 (parity check) Write a program to find whether the number of ones in the binary representation of a given natural number is even or odd.

205 (approximate search) Given a nonempty sorted list of numbers and a number, write a program to determine the index of an item in the list that is closest in value to the given number.

206 Given two natural numbers s and p, write a program to find four natural numbers a, b, c, and d whose sum is s and product p, in time s^2, if such numbers exist.

207 Given three natural numbers n, s, and p, write a program to find a list of length n of natural numbers whose sum is s and product p, if such a list exists.

208 (transitive closure) A relation R: $(0,..n) \rightarrow (0,..n) \rightarrow bool$ can be represented by a square boolean array of size n. Given a relation in the form of a square boolean array, write a program to find

(a) its transitive closure (the strongest transitive relation that is implied by the given relation).

(b) its reflexive transitive closure (the strongest reflexive and transitive relation that is implied by the given relation).

209 (reachability) You are given a finite bunch of places; and a successor function S on places that tells, for each place, those places that are directly reachable from it; and a special place named h (for home). Write a program to find all places that are reachable (reflexively, directly, or indirectly) from h.

210 (shortest path) You are given a square extended rational array in which item $i\ j$ represents the direct distance from place i to place j. If it is not possible to go directly from i to j, then item $i\ j$ is ∞. Write a program to find the square extended rational array in which item $i\ j$ represents the shortest, possibly indirect, distance from place i to place j.

211 (McCarthy's 91 problem) Let i be an integer variable. Let

$M = \textbf{if } i>100 \textbf{ then } i:= i-10 \textbf{ else } i:= 91$

(a) Prove $M \Leftarrow \textbf{if } i>100 \textbf{ then } i:= i-10 \textbf{ else } (i:= i+11.\ M.\ M)$.

(b) Find the execution time of M as refined in part (a).

212 (Towers of Hanoi) There are 3 towers and n disks. The disks are graduated in size; disk 0 is the smallest and disk $n-1$ is the largest. Initially tower A holds all n disks, with the largest disk on the bottom, proceding upwards in order of size to the smallest disk on top. The task is to move all the disks from tower A to tower B, but you can move only one disk at a time, and you must never put a larger disk on top of a smaller one. In the process, you can make use of tower C as intermediate storage.

(a)√ Using the command *MoveDisk from to* to cause a robot arm to move the top disk from tower *from* to tower *to*, write a program to move all the disks from tower A to tower B.

(b)√ Find the execution time, counting *MoveDisk* as time 1, and all else free.

(c) Suppose that the posts where the disks are placed are arranged in an equilateral triangle, so that the distance the arm moves each time is constant (one side of the triangle to get into position plus one side to move the disk), and not dependent on the disk being moved. Suppose the time to move a disk varies with the weight of the disk being moved, which varies with its area, which varies with the square of its radius, which varies with the disk number. Find the execution time.

(d)√ Find the maximum memory space required by the program, counting a recursive call as 1 location (for the return address) and all else free.

(e)√ Find the average memory space required by the program, counting a recursive call as 1 location (for the return address) and all else free.

(f) Find a simple upper bound on the average memory space required by the program, counting a recursive call as 1 location (for the return address) and all else free.

213 (Ackermann) Function ack of two natural variables is defined as follows.

$ack\ 0\ 0\ =\ 2$
$ack\ 1\ 0\ =\ 0$
$ack\ (m+2)\ 0\ =\ 1$
$ack\ 0\ (n+1)\ =\ ack\ 0\ n + 1$
$ack\ (m+1)\ (n+1)\ =\ ack\ m\ (ack\ (m+1)\ n)$

(a) Suppose that functions and function application are not implemented expressions; in that case $n{:=}\ ack\ m\ n$ is not a program. Refine $n{:=}\ ack\ m\ n$ to obtain a program.

(b) Find a time bound. Hint: you may use function ack in your time bound.

(c) Find a space bound.

214 (alternate Ackermann) For each of the following functions f, refine $n{:=} f\ m\ n$, find a time bound (possibly involving f), and find a space bound.

(a)
$f\ 0\ n\ =\ n+2$
$f\ 1\ 0\ =\ 0$
$f\ (m+2)\ 0\ =\ 1$
$f\ (m+1)\ (n+1)\ =\ f\ m\ (f\ (m+1)\ n)$

(b)
$f\ 0\ n\ =\ n\times 2$
$f\ (m+1)\ 0\ =\ 1$
$f\ (m+1)\ (n+1)\ =\ f\ m\ (f\ (m+1)\ n)$

(c)
$f\ 0\ n\ =\ n+1$
$f\ 1\ 0\ =\ 2$
$f\ 2\ 0\ =\ 0$
$f\ (m+3)\ 0\ =\ 1$
$f\ (m+1)\ (n+1)\ =\ f\ m\ (f\ (m+1)\ n)$

215 Let n be a natural variable. Add time according to the recursive measure, and find a finite upper bound on the execution time of

$P\ \Leftarrow\ $ **if** $n \ge 2$ **then** $(n{:=}\ n-2.\ P.\ n{:=}\ n+1.\ P.\ n{:=}\ n+1)$ **else** ok

216√ (roller-coaster) Let n be a natural variable. It is easy to prove

$n'=1\ \Leftarrow\ $ **if** $n=1$ **then** ok
else if $even\ n$ **then** $(n{:=}\ n/2.\ n'=1)$
else $(n{:=}\ 3\times n + 1.\ n'=1)$

The problem is to find the execution time. Warning: this problem has never been solved.

217√ (Fibonacci) The Fibonacci numbers $fib\ n$ are defined as follows.

$fib\ 0 = 0$
$fib\ 1 = 1$
$fib\ (n+2) = fib\ n + fib\ (n+1)$

Write a program to find $fib\ n$ in time $log\ n$. Hint: see Exercise 301.

218 (Fibolucci) Let a and b be integers. Then the Fibolucci numbers for a and b are

$f\ 0 = 0$
$f\ 1 = 1$
$f\ (n+2) = a \times f\ n + b \times f\ (n+1)$

(The Fibonacci numbers are Fibolucci numbers for 1 and 1.) Given natural k, without using any list variables, write a program to compute

$\Sigma n\colon 0,..k\cdot fn \times f(k-n)$

219 (item count) Write a program to find the number of occurrences of a given item in a given list.

220 (duplicate count) Write a program to find how many items are duplicates (repeats) of earlier items
(a) in a given sorted nonempty list.
(b) in a given list.

221 (z-free subtext) Given a text, write a program to find the longest subtext that does not contain the letter "z" .

222 (merge) Given two sorted lists, write a program to merge them into one sorted list.

223 (arithmetic) Let us represent a natural number as a list of naturals, each in the range $0,..b$ for some natural base $b>1$, in reverse order. For example, if $b=10$, then [9; 2; 7] represents 729 . Write programs for each of the following.
(a) Find the list representing a given natural in a given base.
(b) Given a base and two lists representing natural numbers, find the list representing their sum.
(c) Given a base and two lists representing natural numbers, find the list representing their difference. You may assume the first list represents a number greater than or equal to the number represented by the second list. What is the result if this is not so?
(d) Given a base and two lists representing natural numbers, find the list representing their product.
(e) Given a base and two lists representing natural numbers, find the lists representing their quotient and remainder.

224 (machine multiplication) Given two natural numbers, write a program to find their product using only addition, subtraction, doubling, halving, test for even, and test for zero.

225 (machine division) Given two natural numbers, write a program to find their quotient using only addition, subtraction, doubling, halving, test for even, and test for zero.

226 (machine squaring) Given a natural number, write a program to find its square using only addition, subtraction, doubling, halving, test for even, and test for zero.

227 Given a list of roots of a polynomial, write a program to find the list of coefficients.

228 (longest sorted sublist) Write a program to find the length of a longest sorted sublist of a given list, where
(a) the sublist must be consecutive items (a segment).
(b) the sublist consists of items in their order of appearance in the given list, but not necessarily consecutively.

229 (almost sorted segment) An almost sorted list is a list in which at most one adjacent pair of elements is out of order. Write a program to find the length of a longest almost sorted segment of a given list.

230 (edit distance) Given two lists, write a program to find the minimum number of item insertions, item deletions, and item replacements to change one list into the other.

231 (ultimately periodic sequence) You are given function f: *int*→*int* such that the sequence

$$x_0 = 0$$
$$x_{n+1} = f(x_n)$$

generated by f starting at 0 is ultimately periodic:

$$\exists p: nat+1\cdot \exists n: nat\cdot x_n = x_{n+p}$$

The smallest positive p such that $\exists n: nat\cdot x_n = x_{n+p}$ is called the period. Write a program to find the period. Your program should use an amount of storage that is bounded by a constant, and not dependent on f.

232 (partitions) A list of positive integers is called a partition of natural number n if the sum of its items is n. Write a program to find

(a) a list of all partitions of a given natural n. For example, if n=3 then an acceptable answer is [[3]; [1; 2]; [2; 1]; [1; 1; 1]] .

(b) a list of all sorted partitions of a given natural n. For example, if n=3 then an acceptable answer is [[3]; [1; 2]; [1; 1; 1]] .

(c) the sorted list of all partitions of a given natural n. For example, if n=3 then the answer is [[1; 1; 1]; [1; 2]; [2; 1]; [3]] .

(d) the sorted list of all sorted partitions of a given natural n. For example, if n=3 then the answer is [[1; 1; 1]; [1; 2]; [3]] .

233 (largest true square) Write a program to find, within a boolean array, a largest square subarray consisting entirely of items with value ⊤ .

234 (*P*-list) Given a nonempty list S of natural numbers, define a *P*-list as a nonempty list P of natural numbers such that each item of P is an index of S, and

$$\forall i: 1,..\#P\cdot P\,(i-1) < P\,i \le S\,(P\,(i-1))$$

Write a program to find the length of a longest *P*-list for a given list S.

235 (*J*-list) For natural number n, a *J*-list of order n is a list of $2\times n$ naturals in which each m: 0,..n occurs twice, and between the two occurrences of m there are m items.

(a) Write a program that creates a *J*-list of order n if there is one, for given n.

(b) For which n do *J*-lists exist?

236 (diminished *J*-list) For positive integer n, a diminished *J*-list of order n is a list of $2\times n-1$ naturals in which 0 occurs once and each m: 1,..n occurs twice, and between the two occurrences of m there are m items.

(a) Write a program that creates a diminished *J*-list of order n if there is one, for given n.

(b) For which n do diminished *J*-lists exist?

237 (greatest common divisor) Given two positive integers, write a program to find their greatest common divisor.

238 (least common multiple) Given two positive integers, write a program to find their least common multiple.

239 Given two integers (not necessarily positive ones) that are not both zero, write a program to find their greatest common divisor.

240 (common items) Let A be a sorted list of different integers. Let B be another such list. Write a program to find the number of integers that occur in both lists.

241 (unique items) Let A be a sorted list of different integers. Let B be another such list. Write a program to find the sorted list of integers that occur in exactly one of A or B .

242 (smallest common item) Given two sorted lists having at least one item in common, write a program to find the smallest item occurring in both lists.

243 Given three sorted lists having at least one item common to all three, write a program to find the smallest item occurring in all three lists.

244 Given three positive integers, write a program to find their greatest common divisor. One method is to find the greatest common divisor of two of them, and then find the greatest common divisor of that and the remaining number, but there is a better way.

245 (longest common prefix) A natural number can be written as a sequence of decimal digits with a single leading zero. Given two natural numbers, write a program to find the number that is written as their longest common prefix of digits. For example, given 025621 and 02547 , the result is 025 . Hint: this question is about numbers, not about strings or lists.

246 (museum) You are given natural n , rationals s and f (start and finish), and lists A, D: $[n*rat]$ (arrive and depart) such that

$$\forall i\cdot\ s \le Ai \le Di \le f$$

They represent a museum that opens at time s , is visited by n people with person i arriving at time Ai and departing at time Di and closes at time f . Write a program to find the total amount of time during which at least one person is inside the museum, and the average number of people in the museum during the time it is open, in time linear in n , if

(a) list A is sorted.

(b) list D is sorted.

247 (rotation test) Given two lists, write a program to determine if one list is a rotation of the other. You may use item comparisons, but not list comparisons. Execution time should be linear in the length of the lists.

248 (smallest rotation) Given a text variable t , write a program to reassign t its alphabetically (lexicographically) smallest rotation. You may use character comparisons, but not text comparisons.

249 You are given a list variable L assigned a nonempty list. All changes to L must be via procedure $swap$, defined as

$$swap \ = \ \langle i, j\text{: } 0,..\#L \rightarrow L\text{:= } i \rightarrow Lj \mid j \rightarrow Li \mid L\rangle$$

(a) Write a program to reassign L a new list obtained by rotating the old list one place to the right (the last item of the old list is the first item of the new).

(b) (rotate) Given an integer r , write a program to reassign L a new list obtained by rotating the old list r places to the right. (If $r<0$, rotation is to the left $-r$ places.) Recursive execution time must be at most $\#L$.

(c) (segment swap) Given an index p , swap the initial segment up to p with the final segment beginning at p .

250 (squash) Let L be a list variable assigned a nonempty list. Reassign it so that any run of two or more identical items is collapsed to a single item.

251 Let n and p be natural variables. Write a program to solve

$$n\geq 2 \Rightarrow p': 2^{2^{nat}} \wedge n\leq p'<n^2$$

Include a finite upper bound on the execution time, but it doesn't matter how small.

252 (greatest square under a histogram) You are given a histogram in the form of a list H of natural numbers. Write a program to find the longest segment of H in which the height (each item) is at least as large as the segment length.

253 (long texts) A particular computer has a hardware representation for texts of length n characters or less, for some constant n. Longer texts must be represented in software as a string of packaged short texts. (The long text represented is the catenation of the short texts.) A long text is called “packed” if all its items except possibly the last have length n. Write a program to pack a string of packaged short texts without changing the long text represented.

254 (Knuth, Morris, Pratt)

(a) Given list P, find list L such that for every index n of list P, Ln is the length of the longest list that is both a proper prefix and a proper suffix of $P\,[0;..n+1]$. Here is a program to find L.

$A \Leftarrow i:=0.\ L:=[\#P*0].\ j:=1.\ B$

$B \Leftarrow$ **if** $j\geq\#P$ **then** ok **else** $(C.\ L:=j\rightarrow i \mid L.\ j:=j+1.\ B)$

$C \Leftarrow$ **if** $Pi=Pj$ **then** $i:=i+1$
  else if $i=0$ **then** ok
  else $(i:=L\,(i-1).\ C)$

Find specifications A, B, and C so that A is the problem and the three refinements are theorems.

(b) Given list S (subject), list P (pattern), and list L (as in part (a)), determine if P is a segment of S, and if so, where it occurs. Here is a program.

$D \Leftarrow m:=0.\ n:=0.\ E$

$E \Leftarrow$ **if** $m=\#P$ **then** $h:=n-\#P$ **else** F

$F \Leftarrow$ **if** $n=\#S$ **then** $h:=\infty$
  else if $Pm=Sn$ **then** $(m:=m+1.\ n:=n+1.\ E)$
  else G

$G \Leftarrow$ **if** $m=0$ **then** $(n:=n+1.\ F)$ **else** $(m:=L\,(m-1).\ G)$

Find specifications D, E, F, and G so that D is the problem and the four refinements are theorems.

End of Program Theory

10.5 Programming Language

255 (nondeterministic assignment) Generalize the assignment notation $x:=e$ to allow the expression e to be a bunch, with the meaning that x is assigned an arbitrary element of the bunch. For example, $x:=nat$ assigns x an arbitrary natural number. Show the standard boolean notation for this form of assignment. Show what happens to the Substitution Law.

256 Suppose variable declaration is defined as

var $x: T\cdot P \quad = \quad \exists x: undefined\cdot \exists x': T\cdot P$

What are the characteristics of this kind of declaration? Look at the example

var $x: int\cdot ok$

257 What is wrong with defining local variable declaration as follows:

$$\textbf{var}\ x{:}\ T\cdot P \quad = \quad \forall x{:}\ T\cdot \exists x'{:}\ T\cdot P$$

258 Suppose variable declaration with initialization is defined as

$$\textbf{var}\ x{:}\ T := e\cdot\ P \quad = \quad \textbf{var}\ x{:}\ T\cdot\ x{:=}\ e.\ P$$

In what way does this differ from the definition given in Subsection 5.0.0?

259 Here are two different definitions of variable declaration with initialization.

$$\textbf{var}\ x{:}\ T := e\cdot\ P \quad = \quad \exists x, x'{:}\ T\cdot x{=}e \land P$$

$$\textbf{var}\ x{:}\ T := e\cdot\ P \quad = \quad \exists x'{:}\ T\cdot (\text{substitute } e \text{ for } x \text{ in } P\,)$$

Show how they differ with an example.

260 The specification

$$\textbf{var}\ x{:}\ nat\cdot\ x{:=}\ {-1}$$

introduces a local variable and then assigns it a value that is out of bounds. Is this specification implementable? (Proof required.)

261 (frame problem) Suppose there is one nonlocal variable x, and we define $P = x'{=}0$. Can we prove

$$P \Longleftarrow \textbf{var}\ y{:}\ nat\cdot\ y{:=}\ 0.\ P.\ x{:=}\ y$$

The problem is that y was not part of the state space where P was defined, so does P leave y unchanged? Hint: consider the definition of dependent composition. Is it being used properly?

262 Let the state variables be x, y, and z. Rewrite $\textbf{frame}\ x\cdot \top$ without using **frame**. Say in words what the final value of x is.

263 In a language with array element assignment, the program

$$x{:=}\ i.\ \ i{:=}\ A\ i.\ \ A\ i{:=}\ x$$

was written with the intention to swap the values of i and $A\ i$. Assume that all variables and array elements are of type nat, and that i has a value that is an index of A.

(a) In variables x, i, and A, specify that i and $A\ i$ should be swapped, the rest of A should be unchanged, but x might change.

(b) Find the exact precondition for which the program refines the specification of part (a).

(c) Find the exact postcondition for which the program refines the specification of part (a).

264 In a language with array element assignment, what is the exact precondition for $A'\ i' = 1$ to be refined by $(A(A\ i){:=}\ 0.\ A\ i{:=}\ 1.\ i{:=}\ 2)$?

265√ (unbounded bound) Find a time bound for the following program in natural variables x and y.

while $\neg\ x{=}y{=}0$ **do**
  if $y{>}0$ **then** $y{:=}\ y{-}1$
  else $(x{:=}\ x{-}1.\ \ \textbf{var}\ n{:}\ nat\cdot\ y{:=}\ n)$

266 Let $W \Longleftarrow \textbf{while}\ b\ \textbf{do}\ P$ be an abbreviation of $W \Longleftarrow \textbf{if}\ b\ \textbf{then}\ (P.\ W)\ \textbf{else}\ ok$. Let $R \Longleftarrow \textbf{repeat}\ P\ \textbf{until}\ b$ be an abbreviation of $R \Longleftarrow P.\ \textbf{if}\ b\ \textbf{then}\ ok\ \textbf{else}\ R$. Now prove

$$(R \Longleftarrow \textbf{repeat}\ P\ \textbf{until}\ b) \land (W \Longleftarrow \textbf{while}\ \neg b\ \textbf{do}\ P)$$

$$\Leftarrow\ (R \Longleftarrow P.\ W) \land (W \Longleftarrow \textbf{if}\ b\ \textbf{then}\ ok\ \textbf{else}\ R)$$

267 (guarded command) In “Dijkstra's little language” there is a conditional program with the syntax

if $b \rightarrow P$ [] $c \rightarrow Q$ **fi**

where b and c are boolean and P and Q are programs. It can be executed as follows. If exactly one of b and c is true initially, then the corresponding program is executed; if both b and c are true initially, then either one of P or Q (arbitrary choice) is executed; if neither b nor c is true initially, then execution is completely arbitrary.

(a) Express this program in the notations of this book as succinctly as possible.

(b) Refine this program using only the programming notations introduced in Chapter 4.

268√ Using a **for**-loop, write a program to add 1 to every item of a list.

269 Here is one way that we might consider defining the **for**-loop. Let j, n, k and m be integer expressions, and let i be a fresh name.

for i:= nil **do** P = ok

for i:= j **do** P = (substitute j for i in P)

for i:= n;..k ; k;..m **do** P = **for** i:= n;..k **do** P. **for** i:= k;..m **do** P

(a) From this definition, what can we prove about **for** i:= 0;..n **do** n:= n+1 where n is an integer variable?

(b) What kinds of **for**-loop are in the programming languages you know?

270 (majority vote) The problem is to find, in a given list, the majority item (the item that occurs in more than half the places) if there is one. Letting L be the list and m be a variable whose final value is the majority item, prove that the following program solves the problem.

(a)
var e: nat := 0·
for i:= 0;..#L **do**
  if $m = L\ i$ **then** e:= e+1
  else if $i = 2\times e$ **then** (m:= $L\ i$. e:= e+1)
  else ok

(b)
var s: nat := 0·
for i:= 0;..#L **do**
  if $m = L\ i$ **then** ok
  else if $i = 2\times s$ **then** m:= $L\ i$
  else s:= s+1

271 We defined the programmed expression P **result** e with the axiom

$x' = (P$ **result** $e)$ = $P.\ x'=e$

Why don't we define it instead with the axiom

$x' = (P$ **result** $e)$ = $P \Rightarrow x'=e'$

272 Let a and b be rational variables. Define procedure P as

P = $\langle x, y$: $rat \rightarrow$ **if** x=0 **then** a:= x **else** (a:= $x\times y$. a:= $a\times y$)$\rangle$

(a) What is the exact precondition for $a'=b'$ to be refined by $P\ a\ (1/b)$?

(b) Discuss the difference between “eager” and “lazy” evaluation of arguments as they affect both the theory of programming and programming language implementation.

273 “Call-by-value-result” describes a parameter that gets its initial value from an argument, is then a local variable, and gives its final value back to the argument, which therefore must be a variable. Define “call-by-value-result” formally. Discuss its merits and demerits.

274 (call-by-name) Here is a procedure applied to an argument.

$\langle x: int \rightarrow a:= x.\ b:= x \rangle\ (a+1)$

Suppose, by mistake, we replace both occurrences of x in the body with the argument. What do we get? What should we get? (This mistake is known as "call-by-name".)

275 We defined **wait until** w $=$ $t:= max\ t\ w$ where t is an extended integer time variable, and w is an integer expression.

(a)√ Prove **wait until** w $\Leftarrow$ **if** $t \ge w$ **then** ok **else** ($t:= t+1$. **wait until** w)

(b) Now suppose that t is an extended real time variable, and w is an extended real expression. Redefine **wait until** w appropriately, and refine it using the real time measure (assume any positive operation time you need).

276 The specification **wait** w where w is a length of time, not an instant of time, describes a delay in execution of time w. Formalize and implement it using

(a) the recursive time measure.

(b) the real time measure (assume any positive operation times you need).

277 We propose to define a new programming connective $P \blacklozenge Q$. What properties of $\blacklozenge$ are essential? Why?

278 (Boole's booleans) If $\top=1$ and $\bot=0$, express

(a) $\neg a$

(b) $a \wedge b$

(c) $a \vee b$

(d) $a \Rightarrow b$

(e) $a \Leftarrow b$

(f) $a=b$

(g) $a \neq b$

using only the following symbols (in any quantity)

(i) $0\ 1\ a\ b\ (\ )\ +\ -\ \times$

(ii) $0\ 1\ a\ b\ (\ )\ -\ max\ min$

279 Prove that the average value of

(a) n^2 as n varies over $nat+1$ according to probability 2^{-n} is 6.

(b) n as it varies over nat according to probability $(5/6)^n \times 1/6$ is 5.

280 (coin) Repeatedly flip a coin until you get a head. Prove that it takes n flips with probability 2^{-n}. With an appropriate definition of R, the program is

$R \Leftarrow t:= t+1.$ **if** $rand\ 2$ **then** ok **else** R

281 (blackjack) You are dealt a card from a deck; its value is in the range 1 through 13 inclusive. You may stop with just one card, or have a second card if you want. Your object is to get a total as near as possible to 14, but not over 14. Your strategy is to take a second card if the first is under 7. Assuming each card value has equal probability,

(a)√ find the probability for each value of your total.

(b) find the average value of your total.

282√ (dice) If you repeatedly throw a pair of six-sided dice until they are equal, how long does it take?

283 (drunk) A drunkard is trying to walk home. At each time unit, the drunkard may go forward one distance unit, stay in the same position, or go back one distance unit. After n time units, where is the drunkard?

(a) At each time unit, there is 2/3 probability of going forward, and 1/3 probability of staying in the same position. The drunkard does not go back.

(b) At each time unit, there is 1/4 probability of going forward, 1/2 probability of staying in the same position, and 1/4 probability of going back.

(c) At each time unit, there is 1/2 probability of going forward, 1/4 probability of staying in the same position, and 1/4 probability of going back.

284 (Mr.Bean's socks) Mr.Bean is trying to get a matching pair of socks from a drawer containing an inexhaustible supply of red and blue socks. He begins by withdrawing two socks at random. If they match, he is done. Otherwise, he throws away one of them at random, withdraws another sock at random, and repeats. How long will it take him to get a matching pair? Assume that a sock withdrawn from the drawer has 1/2 probability of being each color, and that a sock that is thrown away also has a 1/2 probability of being each color.

End of Programming Language

10.6 Recursive Definition

285 Prove $\neg -1: nat$. Hint: You will need induction.

286 (Cantor's diagonal) Prove $\neg\exists f: nat\to nat\to nat\cdot \forall g: nat\to nat\cdot \exists n: nat\cdot fn = g$.

287 Prove $\forall n: nat\cdot Pn \;=\; \forall n: nat\cdot \forall m: 0,..n\cdot Pm$.

288√ Prove that the square of an odd natural number is $8\times m + 1$ for some natural m .

289 Prove that every positive integer is a product of primes. By “product” we mean the result of multiplying together any natural number of (not necessarily distinct) numbers. By “prime” we mean a natural number with exactly two factors.

290 Here is an argument to “prove” that in any group of people, all the people are the same age. The “proof” is by induction on the size of groups. The induction base is that in any group of size 1 , clearly all the people are the same age. Or we could equally well use groups of size 0 as the induction base. The induction hypothesis is, of course, to assume that in any group of size n , all the people are the same age. Now consider a group of size $n+1$. Let its people be $p_0,\ p_1,\ ..., p_n$. By the induction hypothesis, in the subgroup $p_0,\ p_1,\ ..., p_{n-1}$ of size n , all the people are the same age; to be specific, they are all the same age as p_1 . And in the subgroup $p_1,\ p_2,\ ..., p_n$ of size n , all the people are the same age; again, they are the same age as p_1 . Hence all $n+1$ people are the same age. Formalize this argument and find the flaw.

291 Here is a possible alternative construction axiom for nat .

$$0, 1, nat+nat: nat$$

(a) What induction axiom goes with it?

(b) Are the construction axiom given and your induction axiom of part (a) satisfactory as a definition if nat?

292 Chapter 6 gives four predicate versions of *nat* induction. Prove that they are equivalent.

293 Prove $nat = 0,..\infty$.

294 Here are a construction axiom and an induction axiom for bunch *bad* .

$$(\S n: nat\cdot \neg\ n: bad) : bad$$
$$(\S n: nat\cdot \neg\ n: B) : B \implies bad: B$$

(a)√ Are these axioms consistent?

(b) From these axioms, can we prove the fixed-point equation

$$bad = \S n: nat\cdot \neg\ n: bad$$

295 Prove the following; quantifications are over *nat* .

(a) $\neg\exists i, j\cdot\ j{\neq}0\ \wedge\ 2^{1/2} = i/j$ The square root of 2 is irrational.

(b) $\forall n\cdot\ (\Sigma i: 0,..n\cdot\ 1)\ =\ n$

(c) $\forall n\cdot\ (\Sigma i: 0,..n\cdot\ i)\ =\ n \times (n{-}1) / 2$

(d) $\forall n\cdot\ (\Sigma i: 0,..n\cdot\ i^3)\ =\ (\Sigma i: 0,..n\cdot\ i)^2$

(e) $\forall n\cdot\ (\Sigma i: 0,..n\cdot\ 2^i)\ =\ 2^n - 1$

(f) $\forall n\cdot\ (\Sigma i: 0,..n\cdot\ i{\times}2^i)\ =\ (n{-}2){\times}2^n + 2$

(g) $\forall n\cdot\ (\Sigma i: 0,..n\cdot\ ({-}2)^i)\ =\ (1 - ({-}2)^n) / 3$

(h) $\forall n\cdot\ n{\ge}10\ \Rightarrow\ 2^n > n^3$

(i) $\forall n\cdot\ n{\ge}4\ \Rightarrow\ 3^n > n^3$

(j) $\forall n\cdot\ n{\ge}3\ \Rightarrow\ 2{\times}n^3 > 3{\times}n^2 + 3{\times}n$

(k) $\forall a, d\cdot\ \exists q, r\cdot\ d{\neq}0\ \Rightarrow\ r{<}d\ \wedge\ a = q{\times}d + r$

(l) $\forall a, b\cdot\ a{\le}b\ \Rightarrow\ (\Sigma i: a,..b\cdot\ 3^i) = (3^b{-}3^a)/2$

296 Show that we can define *nat* by fixed-point construction together with

(a) $\forall n: nat\cdot\ 0 \le n < n{+}1$

(b) $\exists m: nat\cdot\ \forall n: nat\cdot\ m \le n < n{+}1$

297√ Suppose we define *nat* by ordinary construction and induction.

$$0, nat{+}1:\ nat$$
$$0, B{+}1:\ B\ \Rightarrow\ nat: B$$

Prove that fixed-point construction and induction

$$nat\ =\ 0, nat{+}1$$
$$B\ =\ 0, B{+}1\ \Rightarrow\ nat: B$$

are theorems.

298 (fixed-point theorem) Suppose we define *nat* by fixed-point construction and induction.

$$nat\ =\ 0, nat{+}1$$
$$B\ =\ 0, B{+}1\ \Rightarrow\ nat: B$$

Prove that ordinary construction and induction

$$0, nat{+}1:\ nat$$
$$0, B{+}1:\ B\ \Rightarrow\ nat: B$$

are theorems. Warning: this is hard, and requires the use of limits.

299 (rulers) Rulers are formed as follows. A vertical stroke | is a ruler. If you append a horizontal stroke — and then a vertical stroke | to a ruler you get another ruler. Thus the first few rulers are | , |—| , |—|—| , |—|—|—| , and so on. No two rulers formed this way are equal. There are no other rulers. What axioms are needed to define bunch *ruler* consisting of all and only the rulers?

300 Function f is called monotonic if $\forall i,j\cdot\ i\le j\ \Rightarrow\ f\,i\le f\,j$.
(a) Prove f is monotonic if and only if $\forall i,j\cdot\ f\,i<f\,j\ \Rightarrow\ i<j$.
(b) Let f: $int\to int$. Prove f is monotonic if and only if $\forall i\cdot\ f\,i\le f\,(i+1)$.
(c) Let f: $nat\to nat$ be such that $\forall n\cdot\ f\,f\,n<f\,(n+1)$. Prove f is the identity function. Hints: First prove $\forall n\cdot\ n\le f\,n$. Then prove f is monotonic. Then prove $\forall n\cdot\ f\,n\le n$.

301 The Fibonacci numbers $fib\ n$ are defined as follows.

$$fib\ 0\ =\ 0$$
$$fib\ 1\ =\ 1$$
$$fib\ (n+2)\ =\ fib\ n+fib\ (n+1)$$

Prove

(a) $$fib\ (gcd\ n\ m)\ =\ gcd\ (fib\ n)\ (fib\ m)$$

where gcd is the greatest common divisor.

(b) $$fib\ n\times fib\ (n+2)\ \ =\ \ fib\ (n+1)^2\ -\ (-1)^n$$
(c) $$fib\ (n+m+1)\ \ =\ \ fib\ n\times fib\ m\ +\ fib\ (n+1)\times fib\ (m+1)$$
(d) $$fib\ (n+m+2)\ \ =\ \ fib\ n\times fib\ (m+1)\ +\ fib\ (n+1)\times fib\ m\ +\ fib\ (n+1)\times fib\ (m+1)$$
(e) $$fib\ (2\times n+1)\ \ =\ \ fib\ n^2\ +\ fib\ (n+1)^2$$
(f) $$fib\ (2\times n+2)\ \ =\ \ 2\times fib\ n\times fib\ (n+1)\ +\ fib\ (n+1)^2$$

302 Let R be a relation of naturals R: $nat\to nat\to bool$ that is monotonic in its second parameter

$$\forall i,j\cdot\ R\ i\ j\Rightarrow R\ i\ (j+1)$$

Prove

$$\exists i\cdot\ \forall j\cdot\ R\ i\ j\ \ =\ \ \forall j\cdot\ \exists i\cdot\ R\ i\ j$$

303 What is the smallest bunch satisfying

(a) $$B\ =\ 0,\ 2\times B+1$$
(b) $$B\ =\ 2,\ B\times B$$

304 What elements can be proven in P from the axiom $P\ =\ 1,\ x,\ -P,\ P+P,\ P\times P$? Prove

$$2\times x^2-1:\ P$$

305 Bunch $this$ is defined by the construction and induction axioms

$$2,\ 2\times this:\ \ this$$
$$2,\ 2\times B:\ \ B\ \ \Rightarrow\ \ this:\ B$$

Bunch $that$ is defined by the construction and induction axioms

$$2,\ that\times that:\ \ that$$
$$2,\ B\times B:\ \ B\ \ \Rightarrow\ \ that:\ B$$

Prove $this = that$.

306 Express 2^{int} without using exponentiation. You may introduce auxiliary names.

307 Let n be a natural number. From the fixed-point equation

$$ply = n,\ ply+ply$$

we obtain a sequence of bunches ply_i by recursive construction.

(a) State ply_i formally (no proof needed).
(b) State ply_i in English.
(c) What is ply_∞ ?
(d) Is ply_∞ a solution? If so, is it the only solution?

308 For each of the following fixed-point equations, what does recursive construction yield? Does it satisfy the fixed-point equation?

(a) $M = [*int], [*M]$

(b) $T = [nil], [T; int; T]$

(c) $A = bool, rat, char, [*A]$

309 Let $A \backslash B$ be the difference between bunch A and bunch B. The operator $\backslash$ has precedence level 4, and is defined by the axiom

$x: A \backslash B \;=\; x: A \land \neg x: B$

For each of the following fixed-point equations, what does recursive construction yield? Does it satisfy the fixed-point equation?

(a) $Q = nat \backslash (Q+3)$

(b) $D = 0, (D+1) \backslash (D-1)$

(c) $E = nat \backslash (E+1)$

(d) $F = 0, (nat \backslash F)+1$

310 For each of the following fixed-point equations, what does recursive construction yield? Does it satisfy the fixed-point equation?

(a) $P = \S n: nat\cdot\ n=0 \land P=null \;\lor\; n: P+1$

(b) $Q = \S x: xnat\cdot\ x=0 \land Q=null \;\lor\; x: Q+1$

311 Here is a pair of mutually recursive equations.

$even = 0, odd+1$

$odd = even+1$

(a) What does recursive construction yield? Show the construction.

(b) Are further axioms needed to ensure that *even* consists of only the even naturals, and *odd* consists of only the odd naturals? If so, what axioms?

312(a) Considering E as the unknown, find three solutions of $E, E+1 = nat$.

(b) Now add the induction axiom $B, B+1 = nat \;\Rightarrow\; E: B$. What is E ?

313 From the construction axiom $0, 1–few: few$

(a) what elements are constructed?

(b) give three solutions (considering *few* as the unknown).

(c) give the corresponding induction axiom.

(d) state which solution is specified by construction and induction.

314 Investigate the fixed-point equation

$strange = \S n: nat\cdot\ \forall m: strange\cdot\ \neg\ m+1: n\times nat$

315 Let *truer* be a bunch of strings of booleans defined by the construction and induction axioms

$\top, \bot; truer; truer:\ truer$

$\top, \bot; B; B:\ B \;\Rightarrow\; truer: B$

Given a string of booleans, write a program to determine if the string is in *truer* .

316 (strings) If S is a bunch of strings, then $*S$ is the bunch of all strings formed by catenating together any number of any strings in S in any order. Define $*S$ by construction and induction.

317 Here are the construction and induction axioms for lists of items of type T .

$[nil], [T], list+list: list$

$[nil], [T], L+L: L \Longrightarrow list: L$

Prove $list = [*T]$.

318 (decimal-point numbers) Using recursive data definition, define the bunch of all decimal-point numbers. These are the rationals that can be expressed as a finite string of decimal digits containing a decimal point. Note: you are defining a bunch of numbers, not a bunch of texts.

319 (Backus-Naur Form) Backus-Naur Form is a grammatical formalism in which grammatical rules are written as in the following example.

$\langle exp\rangle ::= \langle exp\rangle + \langle exp\rangle \mid \langle exp\rangle \times \langle exp\rangle \mid 0 \mid 1$

In our formalism, it would be written

$exp \ = \ exp; \text{"+"}; exp, \ \ exp; \text{"×"}; exp, \ \ \text{"0"}, \ \ \text{"1"}$

In a similar fashion, write axioms to define each of the following.

(a) palindromes: texts that read the same forward and backward. Use a two-symbol alphabet.
(b) palindromes of odd length.
(c) all texts consisting of "a"s followed by the same number of "b"s.
(d) all texts consisting of "a"s followed by at least as many "b"s.

320 Section 6.1 defines program zap by the fixed-point equation

$zap \ = \ \textbf{if } x{=}0 \textbf{ then } y{:=}\ 0 \textbf{ else } (x{:=}\ x{-}1.\ \ t{:=}\ t{+}1.\ \ zap)$

(a) Prove $zap \Longrightarrow x{\ge}0 \Rightarrow x'{=}y'{=}0 \land t' = t{+}x$.
(b) Prove $x{\ge}0 \land x'{=}y'{=}0 \land t' = t{+}x \Longrightarrow zap$.
(c) What axiom is needed to make zap the weakest fixed-point?
(d) What axiom is needed to make zap the strongest fixed-point?
(e) Section 6.1 gives six solutions to this equation. Find more solutions. Hint: strange things can happen at time ∞ .

321 Let all variables be integer. Add recursive time. Using recursive construction, find a fixed-point of

(a) $skip \ = \ \textbf{if } i{\ge}0 \textbf{ then } (i{:=}\ i{-}1.\ \ skip.\ \ i{:=}\ i{+}1) \textbf{ else } ok$
(b) $inc \ = \ ok \lor (i{:=}\ i{+}1.\ \ inc)$
(c) $sqr \ = \ \textbf{if } i{=}0 \textbf{ then } ok \textbf{ else } (s{:=}\ s + 2{\times}i - 1.\ \ i{:=}\ i{-}1.\ \ sqr)$
(d) $fac \ = \ \textbf{if } i{=}0 \textbf{ then } f{:=}\ 1 \textbf{ else } (i{:=}\ i{-}1.\ \ fac.\ \ i{:=}\ i{+}1.\ \ f{:=}\ f{\times}i)$
(e) $chs \ = \ \textbf{if } a{=}b \textbf{ then } c{:=}\ 1 \textbf{ else } (a{:=}\ a{-}1.\ \ chs.\ \ a{:=}\ a{+}1.\ \ c{:=}\ c{\times}a/(a{-}b)\ )$

322 Let all variables be integer. Add recursive time. Any way you can, find a fixed-point of

(a) $walk \ = \ \textbf{if } i{\ge}0 \textbf{ then } (i{:=}\ i{-}2.\ \ walk.\ \ i{:=}\ i{+}1.\ \ walk.\ \ i{:=}\ i{+}1) \textbf{ else } ok$
(b) $crawl \ = \ \textbf{if } i{\ge}0 \textbf{ then } (i{:=}\ i{-}1.\ \ crawl.\ \ i{:=}\ i{+}2.\ \ crawl.\ \ i{:=}\ i{-}1) \textbf{ else } ok$
(c) $run \ = \ \textbf{if } even\ i \textbf{ then } i{:=}\ i/2 \textbf{ else } i{:=}\ i{+}1.$
$\quad\quad \textbf{if } i{=}1 \textbf{ then } ok \textbf{ else } run$

323 Investigate how recursive construction is affected when we start with

(a) $t' = \infty$
(b) $t{:=}\ \infty$

324 Let x be an integer variable. Using the recursive time measure, add time and then find the strongest implementable specifications P and Q that you can find for which

$$P \Leftarrow x' \ge 0.\ Q$$
$$Q \Leftarrow \textbf{if } x{=}0 \textbf{ then } ok \textbf{ else } (x{:=}\ x{-}1.\ Q)$$

Assume that $x' \ge 0$ takes no time.

325 Let x be an integer variable.

(a) Using the recursive time measure, add time and then find the strongest implementable specification S that you can find for which

$$S \Leftarrow \textbf{if } x{=}0 \textbf{ then } ok$$
$$\textbf{else if } x{>}0 \textbf{ then } (x{:=}\ x{-}1.\ S)$$
$$\textbf{else } (x' \ge 0.\ S)$$

Assume that $x' \ge 0$ takes no time.

(b) What do we get from recursive construction starting with $t' \ge t$?

326 Prove that the following three ways of defining R are equivalent.

$$R = ok \vee (R.\ S)$$
$$R = ok \vee (S.\ R)$$
$$R = ok \vee S \vee (R.\ R)$$

327 Prove the laws of Refinement by Steps and Refinement by Parts for **while**-loops.

328 Prove that

$$\forall\sigma, \sigma'\cdot\ (t'{\ge}t \wedge (\textbf{if } b \textbf{ then } (P.\ t{:=}\ t{+}1.\ W) \textbf{ else } ok) \Leftarrow W)$$
$$\Leftarrow\ \forall\sigma, \sigma'\cdot\ (\textbf{while } b \textbf{ do } P \Leftarrow W)$$

is equivalent to the **while** construction axioms, and hence that construction and induction can be expressed together as

$$\forall\sigma, \sigma'\cdot\ (t'{\ge}t \wedge (\textbf{if } b \textbf{ then } (P.\ t{:=}\ t{+}1.\ W) \textbf{ else } ok) \Leftarrow W)$$
$$=\ \forall\sigma, \sigma'\cdot\ (\textbf{while } b \textbf{ do } P \Leftarrow W)$$

329 The notation **do** P **while** b has been used as a loop construct that is executed as follows. First P is executed; then b is evaluated, and if $\top$ execution is repeated, and if $\bot$ execution is finished. Define **do** P **while** b by construction and induction axioms.

330 Using the definition of Exercise 329, but ignoring time, prove

(a) $\textbf{do } P \textbf{ while } b\ =\ P.\ \textbf{while } b \textbf{ do } P$

(b) $\textbf{while } b \textbf{ do } P\ =\ \textbf{if } b \textbf{ then do } P \textbf{ while } b \textbf{ else } ok$

(c)
$$(\forall\sigma, \sigma'\cdot\ (D = \textbf{do } P \textbf{ while } b)) \wedge (\forall\sigma, \sigma'\cdot\ (W = \textbf{while } b \textbf{ do } P))$$
$$=\ (\forall\sigma, \sigma'\cdot\ (D = P.\ W)) \wedge (\forall\sigma, \sigma'\cdot\ (W = \textbf{if } b \textbf{ then } D \textbf{ else } ok))$$

331 Let P: $nat \rightarrow bool$.

(a) Define quantifier $FIRST$ so that $FIRST\ m{:}\ nat\cdot\ Pm$ is the smallest natural m such that Pm , and ∞ if there is none.

(b) Prove $n{:=}\ FIRST\ m{:}\ nat\cdot\ Pm \Leftarrow n{:=}\ 0.\ \textbf{while } \neg Pn \textbf{ do } n{:=}\ n{+}1$.

332 Let the state consist of boolean variables b and c . Let

$$W = \textbf{if } b \textbf{ then } (P.\ W) \textbf{ else } ok$$
$$X = \textbf{if } b{\vee}c \textbf{ then } (P.\ X) \textbf{ else } ok$$

(a) Find a counterexample to $W.\ X = X$.

(b) Now let W and X be the weakest solutions of those equations, and prove $W.\ X = X$.

333 In real variable x , consider the equation

$$P = P.\ x:= x^2$$

(a) Find 7 distinct solutions for P .

(b) Which solution does recursive construction give starting from $\top$? Is it the weakest solution?

(c) If we add a time variable, which solution does recursive construction give starting from $t' \ge t$? Is it a strongest implementable solution?

(d) Now let x be an integer variable, and redo the question.

334 Suppose we define **while** b **do** P by ordinary construction and induction, ignoring time.

$$\textbf{if } b \textbf{ then } (P.\ \textbf{while } b \textbf{ do } P) \textbf{ else } ok \ \Leftarrow\ \textbf{while } b \textbf{ do } P$$

$$\forall\sigma, \sigma'\cdot (\textbf{if } b \textbf{ then } (P.\ W) \textbf{ else } ok \Leftarrow W) \Rightarrow \forall\sigma, \sigma'\cdot (\textbf{while } b \textbf{ do } P \Leftarrow \mathrm{W})$$

Prove that fixed-point construction and induction

$$\textbf{while } b \textbf{ do } P = \textbf{if } b \textbf{ then } (P.\ \textbf{while } b \textbf{ do } P) \textbf{ else } ok$$

$$\forall\sigma, \sigma'\cdot (W = \textbf{if } b \textbf{ then } (P.\ W) \textbf{ else } ok) \Rightarrow \forall\sigma, \sigma'\cdot (\textbf{while } b \textbf{ do } P \Leftarrow \mathrm{W})$$

are theorems.

335 Suppose we define **while** b **do** P by fixed-point construction and induction, ignoring time.

$$\textbf{while } b \textbf{ do } P = \textbf{if } b \textbf{ then } (P.\ \textbf{while } b \textbf{ do } P) \textbf{ else } ok$$

$$\forall\sigma, \sigma'\cdot (W = \textbf{if } b \textbf{ then } (P.\ W) \textbf{ else } ok) \Rightarrow \forall\sigma, \sigma'\cdot (\textbf{while } b \textbf{ do } P \Leftarrow \mathrm{W})$$

Prove that ordinary construction and induction

$$\textbf{if } b \textbf{ then } (P.\ \textbf{while } b \textbf{ do } P) \textbf{ else } ok \ \Leftarrow\ \textbf{while } b \textbf{ do } P$$

$$\forall\sigma, \sigma'\cdot (\textbf{if } b \textbf{ then } (P.\ W) \textbf{ else } ok \Leftarrow W) \Rightarrow \forall\sigma, \sigma'\cdot (\textbf{while } b \textbf{ do } P \Leftarrow \mathrm{W})$$

are theorems. Warning: this is hard, and requires the use of limits.

End of Recursive Definition

10.7 Theory Design and Implementation

336 (widgets) A theory of widgets is presented in the form of some new syntax and some axioms. An implementation of widgets is written.

(a) How do we know whether the theory of widgets is consistent or inconsistent?

(b) How do we know whether the theory of widgets is complete or incomplete?

(c) How do we know whether the implementation of widgets is correct or incorrect?

337√ Implement data-stack theory to make the two boolean expressions

$$pop\ empty = empty$$

$$top\ empty = 0$$

antitheorems.

338 Prove that the following definitions implement the simple data-stack theory.

$$stack = [nil], [stack; X]$$

$$push = \langle s: stack \rightarrow \langle x: X \rightarrow [s; x]\rangle\rangle$$

$$pop = \langle s: stack \rightarrow s\ 0\rangle$$

$$top = \langle s: stack \rightarrow s\ 1\rangle$$

339 (weak data-stack) In Subsection 7.1.3 we designed a program-stack theory so weak that we could add axioms to count pushes and pops without inconsistency. Design a similarly weak data-stack theory.

340 (data-queue implementation) Implement the data-queue theory presented in Section 7.0.

341 (slip) The slip data structure introduces the name *slip* with the following axioms:

$slip = [X; slip]$

$B = [X; B] \Rightarrow B: slip$

where X is some given type. Can you implement it?

342 Prove that the program-stack implementation given in Subsection 7.1.1 satisfies the program-stack axioms of Subsection 7.1.0.

343 Implement weak program-stack theory as follows: the implementer's variable is a list that grows and never shrinks. A popped item must be marked as garbage.

344 You are given a program-stack. Can you write a program composed from the programs

push "A" *push* "B" *push* "C" *push* "D" *push* "E"

in that order, with the programs *print top* and *pop* interspersed wherever needed as many times as needed, to obtain the following output?

(a) B D E C A
(b) B C D E A
(c) C A D E B
(d) A B E C D
(e) A B C D E

345 (brackets) You are given a text t of characters drawn from the alphabet "x", "(", ")", "[", "]" . Write a program to determine if t has its brackets properly paired and nested.

346 (limited-stack) A stack, according to our axioms, has an unlimited capacity to have items pushed onto it. A limited-stack is a similar data structure but with a limited capacity to have items pushed onto it.

(a) Design axioms for a limited-data-stack.
(b) Design axioms for a limited-program-stack.
(c) Can the limit be 0 ?

347 (limited-queue) A queue, according to our axioms, has an unlimited capacity to have items joined onto it. A limited-queue is a similar data structure but with a limited capacity to have items joined onto it.

(a) Design axioms for a limited-data-queue.
(b) Design axioms for a limited-program-queue.
(c) Can the limit be 0 ?

348 You are given a program-queue. Can you write a program composed from the programs

join "A" *join* "B" *join* "C" *join* "D" *join* "E"

in that order, with the programs *print front* and *leave* interspersed wherever needed as many times as needed, to obtain the following output?

(a) B D E C A
(b) B C D E A
(c) C A D E B
(d) A B E C D
(e) A B C D E

349 Each of the program theories provides a single, anonymous instance of a data structure. How can a program theory be made to provide many instances of a data structure, like data theories do?

350 (circular list) Design axioms for a circular list. There should be operations to create an empty list, to move along one position in the list (the first item comes after the last, in circular fashion), to insert an item at the current position, to delete the current item, and to give the current item.

351 (resettable variable) A resettable variable is defined as follows. There are three new names: *value* (of type X), *set* (a procedure with one parameter of type X), and *reset* (a program). Here are the axioms:

$value'=x \Leftarrow set\ x$

$value'=value \Leftarrow set\ x.\ reset$

$reset.\ reset = reset$

Implement this data structure, with proof.

352 A particular program-list has the following operations:

- the operation *mkempty* makes the list empty
- the operation *extend x* catenates item x to the end of the list
- the operation *swap i j* swaps the items at indexes i and j
- the expression *length* tells the length of the list
- the expression *item i* tells the item at index i

(a) Write axioms to define this program-list.
(b) Implement this program-list, with proof.

353 (linear algebra) Design a theory of linear algebra. It should include scalar, vector, and matrix sums, products, and inner products. Implement the theory, with proof.

354 (leafy tree) A leafy tree is a tree with information residing only at the leaves. Design appropriate axioms for a binary leafy data-tree.

355 A tree can be implemented by listing its items in breadth order.
(a) Implement a binary tree by a list of its items such that the root is at index 0 and the left and right subtrees of an item at index n are rooted at indexes $2\times n+1$ and $2\times n+2$.
(b) Prove your implementation.
(c) Generalize this implementation to trees in which each item can have at most k branches for arbitrary (but constant) k .

356 (hybrid-tree) Chapter 7 presented data-tree theory and program-tree theory. Design a hybrid-tree theory in which there is only one tree structure, so it can be an implementer's variable with program operations on it, but there can be many pointers into the tree, so they are data-pointers (they may be data-stacks).

357 (heap) A heap is a tree with the property that the root is the largest item and the subtrees are heaps.
(a) Specify the heap property formally.
(b) Write a function *heapgraft* that makes a heap from two given heaps and a new item. It may make use of *graft* , and may rearrange the items as necessary to produce a heap.

358 (leaf count) Write a program to count the number of leaves in a tree.

359 (binary search tree) A binary search tree is a binary tree with the property that all items in the left subtree are less than the root item, all items in the right subtree are greater than the root item, and the subtrees are also binary search trees.
(a) Specify the binary search tree property formally.
(b) How many binary search trees are there with three items?
(c) How many binary search trees are there with three distinct items?
(d) Write a program to find an item in a binary search tree.
(e) Write a program to add an item to a binary search tree as a new leaf.
(f) Write a program to make a list of the items in a binary search tree in order.
(g) Write a program to determine whether two binary search trees have the same items.

360 (party) A company is structured as a tree, with employees at the nodes. Each employee, except the one at the root, has a boss represented by their parent in the tree. Each employee has a conviviality rating (a number) representing how much fun they are at a party. But no-one will be at a party with their boss. Write a program to find the bunch of employees to invite to a party so that the total convivialty is maximized.

361 (insertion list) An insertion list is a data structure similar to a list, but with an associated insertion point.

[...; 4 ; 7 ; 1 ; 0 ; 3 ; 8 ; 9 ; 2 ; 5 ; ...]
↑
insertion point

insert puts an item at the insertion point (between two existing items), leaving the insertion point at its right. *erase* removes the item to the left of the insertion point, closing up the list. *item* gives the item to the left of the insertion point. *forward* moves the insertion point one item to the right. *back* moves the insertion point one item to the left.
(a) Design axioms for a doubly-infinite data-insertion list.
(b) Design axioms for a doubly-infinite program-insertion list.
(c) Design axioms for a finite data-insertion list.
(d) Design axioms for a finite program-insertion list.

362√ (parsing) Define E as a bunch of strings of lists of characters satisfying

$E = ["x"],\ ["if"]; E; ["then"]; E; ["else"]; E$

Given a string of lists of characters, write a program to determine if the string is in the bunch E.

363 A theory provides three names: *zero*, *increase*, and *inquire*. It is presented by an implementation. Let u: *bool* be the user's variable, and let v: *nat* be the implementer's variable. The axioms are

$zero = v:= 0$
$increase = v:= v+1$
$inquire = u:= even\ v$

Use data transformation to replace v with w: *bool* according to the transformer
(a)√ $w = even\ v$
(b) $\top$
(c) $\bot$ (this isn't a data transformer, since $\forall w\cdot \exists v\cdot \bot$ isn't a theorem, but apply it anyway to see what happens)

364 A theory provides three names: *set* , *flip* , and *ask* . It is presented by an implementation. Let u: *bool* be the user's variable, and let v: *bool* be the implementer's variable. The axioms are

$set = v := \top$
$flip = v := \neg v$
$ask = u := v$

(a)√ Replace v with w: *nat* according to the data transformer $v = even\ w$.
(b) Replace v with w: *nat* according to the data transformer $(w=0 \Rightarrow v) \wedge (w=1 \Rightarrow \neg v)$. Is anything wrong?
(c) Replace v with w: *nat* according to $(v \Rightarrow w=0) \wedge (\neg v \Rightarrow w=1)$. Is anything wrong?

365 Let a , b and c be boolean variables. Variables a and b are implementer's variables, and c is a user's variable for the operations

$seta = a := \top$
$reseta = a := \bot$
$flipa = a := \neg a$
$setb = b := \top$
$resetb = b := \bot$
$flipb = b := \neg b$
$and = c := a \wedge b$
$or = c := a \vee b$

This theory must be reimplemented using integer variables, with 0 for $\bot$ and all other integers for $\top$.
(a) What is the data transformer?
(b) Transform *seta* .
(c) Transform *flipa* .
(d) Transform *and* .

366 Find a data transformer to transform the program of Exercise 270(a) into the program of Exercise 270(b).

367√ (security switch) A security switch has three boolean user's variables a , b , and c . The users assign values to a and b as input to the switch. The switch's output is assigned to c . The output changes when both inputs have changed. More precisely, the output changes when both inputs differ from what they were the previous time the output changed. The idea is that one user might flip their input indicating a desire for the output to change, but the output does not change until the other user flips their input indicating agreement that the output should change. If the first user changes back before the second user changes, the output does not change.
(a) Implement a security switch to correspond as directly as possible to the informal description.
(b) Transform the implementation of part (a) to obtain an efficient implementation.

368 The user's variable is boolean b . The implementer's variables are natural x and y . The operations are:

$done = b := x=y=0$
$step = \textbf{if } y>0 \textbf{ then } y := y-1 \textbf{ else } (x := x-1.\ \textbf{var } n: nat\cdot\ y := n)$

Replace the two implementer's variables x and y with a single new implementer's variable: natural z .

369 Let p be a user's boolean variable, and let m be an implementer's natural variable. The operations allow the user to assign a value n to the implementer's variable, and to test whether the implementer's variable is a prime number.

$$assign\ n\ =\ m:=n$$
$$check\ =\ p:=prime\ m$$

assuming $prime$ is suitably defined. If $prime$ is an expensive function, and the $check$ operation is more frequent than the $assign$ operation, we can improve the solution by making $check$ less expensive even if that makes $assign$ more expensive. Using data transformation, make this improvement.

370√ (take a number) Maintain a list of natural numbers standing for those that are “in use”. The three operations are:

- make the list empty (for initialization)
- assign to variable n a number that is not in use, and add this number to the list (now it is in use)
- given a number n that is in use, remove it from the list (now it is no longer in use, and it can be reused later)

(a) Implement the operations in terms of bunches.
(b) Use a data transformer to replace all bunch variables with natural variables.
(c) Use a data transformer to obtain a distributed solution.

371√ A limited queue is a queue with a limited number of places for items. Let the limit be positive natural n, and let Q: $[n*X]$ and p: nat be implementer's variables. Here is an implementation.

$$mkemptyq\ =\ p:=0$$
$$isemptyq\ =\ p=0$$
$$isfullq\ =\ p=n$$
$$join\ x\ =\ Qp:=x.\ \ p:=p+1$$
$$leave\ =\ \textbf{for}\ i:=1;..p\ \textbf{do}\ Q(i-1):=Qi.\ \ p:=p-1$$
$$front\ =\ Q0$$

Removing the front item from the queue takes time $p-1$ to shift all remaining items down one index. Transform the queue so that all operations are instant.

372 A binary tree can be stored as a list of nodes in breadth order. Traditionally, the root is at index 1, the node at index n has its left child at index $2\times n$ and its right child at index $2\times n+1$. Suppose the user's variable is x: X, and the implementer's variables are s: $[*X]$ and p: $nat+1$, and the operations are

$$goHome\ =\ p:=1$$
$$goLeft\ =\ p:=2\times p$$
$$goRight\ =\ p:=2\times p+1$$
$$goUp\ =\ p:=div\ p\ 2$$
$$put\ =\ s:=p\rightarrow x\mid s$$
$$get\ =\ x:=s\ p$$

Now suppose we decide to move the entire list down one index so that we do not waste index 0. The root is at index 0, its children are at indexes 1 and 2, and so on. Develop the necessary data transform, and use it to transform the operations.

373 (sparse array) An array A: [*[*rat]] is said to be sparse if many of its items are 0. We can represent such an array compactly as a list of triples $[i; j; x]$ of all nonzero items $A\ i\ j = x \neq 0$. Using this idea, find a data transformer and transform the programs

(a) $A := [100*[100*0]]$

(b) $x := A\ i\ j$

(c) $A := (i;j) \rightarrow x \mid A$

374 (transformation incompleteness) The user's variable is i and the implementer's variable is j, both of type nat. The operations are:

$initialize \;=\; i' = 0 \le j' < 3$

$step \;=\;$ **if** $j>0$ **then** $(i := i+1.\ \ j := j-1)$ **else** ok

The user can look at i but not at j. The user can $initialize$, which starts i at 0 and starts j at any of 3 values. The user can then repeatedly $step$ and observe that i increases 0 or 1 or 2 times and then stops increasing, which effectively tells the user what value j started with.

(a) Show that there is no data transformer to replace j with boolean variable b so that

$initialize$ is transformed to $i'=0$

$step$ is transformed to **if** $b \wedge i<2$ **then** $i' = i+1$ **else** ok

The transformed $initialize$ starts b either at $\top$, meaning that i will be increased, or at $\bot$, meaning that i will not be increased. Each use of the transformed $step$ tests b to see if we might increase i, and checks $i<2$ to ensure that i will remain below 3. If i is increased, b is again assigned either of its two values. The user will see i start at 0 and increase 0 or 1 or 2 times and then stop increasing, exactly as in the original specification.

(b) Use the data transformer $b=(j>0)$ to transform $initialize$ and $i+j=k \Rightarrow step$ where k: $0, 1, 2$.

End of Theory Design and Implementation

10.8 Concurrency

375 Let x and y be natural variables. Rewrite the following program as a program that does not use $\|$.

(a) $x := x+1 \ \|$ **if** $x=0$ **then** $y := 1$ **else** ok

(b) **if** $x>0$ **then** $y := x-1$ **else** ok $\|$ **if** $x=0$ **then** $x := y+1$ **else** ok

376 If we ignore time, then

$x := 3.\ \ y := 4 \;=\; x := 3 \,\|\, y := 4$

Some dependent compositions could be executed in parallel if we ignore time. But the time for $P.Q$ is the sum of the times for P and Q, and that forces the execution to be sequential.

$t := t+1.\ \ t := t+2 \;=\; t := t+3$

Likewise some independent compositions could be executed sequentially, ignoring time. But the time for $P\|Q$ is the maximum of the times for P and Q, and that forces the execution to be parallel.

$t := t+1 \,\|\, t := t+2 \;=\; t := t+2$

Invent another form of composition, intermediate between dependent and independent composition, whose execution is sequential to the extent necessary, and parallel to the extent possible. Warning: this is a research question.

377 (disjoint composition) Independent composition $P||Q$ requires that P and Q have no variables in common, although each can make use of the initial values of the other's variables by making a private copy. An alternative, let's say disjoint composition, is to allow both P and Q to use all the variables with no restrictions, and then to choose disjoint sets of variables v and w and define

$$P\ |v|w|\ Q \ = \ (P.\ v'=v) \wedge (Q.\ w'=w)$$

(a) Describe how $P\ |v|w|\ Q$ can be executed.

(b) Prove that if P and Q are implementable specifications, then $P\ |v|w|\ Q$ is implementable.

378 (semi-dependent composition) Independent composition $P||Q$ requires that P and Q have no state variables in common, although each can make use of the initial values of the other's state variables by making a private copy. In this question we explore another kind of composition, let's say semi-dependent composition $P|||Q$. Like dependent composition, it requires P and Q to have the same state variables. Like independent composition, it can be executed by executing the processes in parallel, but each process makes its assignments to local copies of state variables. Then, when both processes are finished, the final value of a state variable is determined as follows: if both processes left it unchanged, it is unchanged; if one process changed it and the other left it unchanged, its final value is the changed one; if both processes changed it, its final value is arbitrary. This final rewriting of state variables does not require coordination or communication between the processes; each process rewrites those state variables it has changed. In the case when both processes have changed a state variable, we do not even require that the final value be one of the two changed values; the rewriting may mix the bits.

(a) Formally define semi-dependent composition, including time.

(b) What laws apply to semi-dependent composition?

(c) Under what circumstances is it unnecessary for a process to make private copies of state variables?

(d) In variables x , y , and z , without using $|||$, express

$$x:=z\ |||\ y:=z$$

(e) In variables x , y , and z , without using $|||$, express

$$x:=y\ |||\ y:=x$$

(f) In variables x , y , and z , without using $|||$, express

$$x:=y\ |||\ x:=z$$

(g) In variables x , y , and z , prove

$$x:=y\ |||\ x:=z \ = \ \textbf{if } x=y \textbf{ then } x:=z \textbf{ else if } x=z \textbf{ then } x:=y \textbf{ else } (x:=y\ |||\ x:=z)$$

(h) In boolean variables x , y and z , without using $|||$, express

$$x:=x\wedge z\ \ |||\ \ y:=y\wedge\neg z\ \ |||\ \ x:=x\wedge\neg z\ \ |||\ \ y:=y\wedge z$$

(i) Let w: 0,..4 and z: 0, 1 be variables. Without using $|||$, express

$$w:=2\times max\ (div\ w\ 2)\ z\ +\ max\ (mod\ w\ 2)\ (1-z)$$
$$|||\quad w:=2\times max\ (div\ w\ 2)\ (1-z)\ +\ max\ (mod\ w\ 2)\ z$$

379 Extend the definition of semi-dependent composition $P|||Q$ (Exercise 378) from variables to list items.

380 Redefine semi-dependent composition $P|||Q$ (Exercise 378) so that if P and Q agree on a changed value for a variable, then it has that final value, and if they disagree on a changed value for a variable, then its final value is

(a) arbitrary.

(b) either one of the two changed values.

381 We want to find the smallest number in $0,..n$ with property p. Linear search solves the problem. But evaluating p is expensive; let us say it takes time 1, and all else is free. The fastest solution is to evaluate p on all n numbers concurrently, and then find the smallest number that has the property. Write a program without concurrency for which the sequential to parallel transformation gives the desired computation.

382 Exercise 134 asks for a program to compute cumulative sums (running total). Write a program that can be transformed from sequential to parallel execution with $log\ n$ time where n is the length of the list.

383 (sieve) Given variable p: [n*$bool$] := [⊥; ⊥; (n–2)*⊤] , the following program is the sieve of Eratosthenes for determining if a number is prime.

for i:= 2;..$ceil$ ($n^{1/2}$) **do**
  if $p\ i$ **then for** j:= i;..$ceil$ (n/i) **do** p:= (j×i)→⊥ | p
  else ok

(a) Show how the program can be transformed for concurrency. State your answer by drawing the execution pattern.

(b) What is the execution time, as a function of n, with maximum concurrency?

384√ (dining philosophers) Five philosophers are sitting around a round table. At the center of the table is an infinite bowl of noodles. Between each pair of neighboring philosophers is a chopstick. Whenever a philosopher gets hungry, the hungry philosopher reaches for the two chopsticks on the left and right, because it takes two chopsticks to eat. If either chopstick is unavailable because the neighboring philosopher is using it, then this hungry philosopher will have to wait until it is available again. When both chopsticks are available, the philosopher eats for a while, then puts down the chopsticks, and goes back to thinking, until the philosopher gets hungry again. The problem is to write a program whose execution simulates the life of these philosophers with the maximum concurrency that does not lead to deadlock.

End of Concurrency

10.9 Interaction

385√ Suppose a and b are integer boundary variables, x and y are integer interactive variables, and t is an extended integer time variable. Suppose that each assignment takes time 1. Express the following using ordinary boolean operators, without using any programming notations.

(x:= 2. x:= x+y. x:= x+y) || (y:= 3. y:= x+y)

386 Let a and b be boolean interactive variables. Define

$loop$ = **if** b **then** $loop$ **else** ok

Add a time variable according to any reasonable measure, and then express

b:= ⊥ || $loop$

as an equivalent program but without using || .

387 The Substitution Law does not work for interactive variables.

(a) Show an example of the failure of the law.

(b) Develop a new Substitution Law for interactive variables.

388√ (thermostat) Specify a thermostat for a gas burner. The thermostat operates in parallel with other processes

$$thermometer \;||\; control \;||\; thermostat \;||\; burner$$

The thermometer and the control are typically located together, but they are logically distinct. The inputs to the thermostat are:

- real *temperature* , which comes from the thermometer and indicates the actual temperature.
- real *desired* , which comes from the control and indicates the desired temperature.
- boolean *flame* , which comes from a flame sensor in the burner and indicates whether there is a flame.

The outputs of the thermostat are:

- boolean *gas* ; assigning it $\top$ turns the gas on and $\bot$ turns the gas off.
- boolean *spark* ; assigning it $\top$ causes sparks for the purpose of igniting the gas.

Heat is wanted when the desired temperature falls ε below the actual temperature, and not wanted when the desired temperature rises ε above the actual temperature, where ε is small enough to be unnoticeable, but large enough to prevent rapid oscillation. To obtain heat, the spark should be applied to the gas for at least 1 second to give it a chance to ignite and to allow the flame to become stable. But a safety regulation states that the gas must not remain on and unlit for more than 3 seconds. Another regulation says that when the gas is shut off, it must not be turned on again for at least 20 seconds to allow any accumulated gas to clear. And finally, the gas burner must respond to its inputs within 1 second.

389√ (grow slow) Suppose *alloc* allocates 1 unit of memory space and takes time 1 to do so. Then the following computation slowly allocates memory.

$$GrowSlow \;\Leftarrow\; \textbf{if } t{=}2{\times}x \textbf{ then } (alloc \;||\; x{:=}\,t) \textbf{ else } t{:=}\,t{+}1.\;\; GrowSlow$$

If the time is equal to $2{\times}x$, then one space is allocated, and in parallel x becomes the time stamp of the allocation; otherwise the clock ticks. The process is repeated forever. Prove that if the space is initially less than the logarithm of the time, and x is suitably initialized, then at all times the space is less than the logarithm of the time.

390 According to the definition of assignment to an interactive variable, writing to the variable takes some time during which the value of the variable is unknown. But any variables in the expression being assigned are read instantaneously at the start of the assignment. Modify the definition of assignment to an interactive variable so that

(a) writing takes place instantaneously at the end of the assignment.

(b) reading the variables in the expression being assigned takes the entire time of the assignment, just as writing does.

391 (interactive data transformation) Section 7.2 presented data transformation for boundary variables. How do we do data transformation when there are interactive variables? Warning: this is a research question.

392 (telephone) Specify the control of a simple telephone. Its inputs are those actions you can perform: picking up the phone, dialing a digit, and putting down (hanging up) the phone. Its output is a list of digits (the number dialed). The end of dialing is indicated by 5 seconds during which no further digit is dialed. If the phone is put down without waiting 5 seconds, then there is no output. But, if the phone is put down and then picked up again within 2 seconds, this is considered to be an accident, and it does not affect the output.

393 (consensus) Some parallel processes are connected in a ring. Each process has a local integer variable with an initial value. These initial values may differ, but otherwise the processes are identical. Execution of all processes must terminate in time linear in the number of processes, and in the end the values of these local variables must all be the same, and equal to one of the initial values. Write the processes.

394 Many programming languages require a variable for input, with a syntax such as **read** x . Define this form of input formally. When is it more convenient than the input described in Section 9.1? When is it less convenient?

395 Write a program to print the sequence of natural numbers, one per time unit.

396 Write a program to repeatedly print the current time, up until some given time.

397 Given a finite string S of different characters sorted in increasing order, write a program to print the strings $*(S_{0,..\leftrightarrow S})$ in the following order: shorter strings come before longer strings; strings of equal length are in string (alphabetical, lexicographic) order.

398 (T-strings) Let us call a string S: *("a", "b", "c") a T-string if no two adjacent nonempty segments are identical:

$$\neg\exists i, j, k\cdot\ 0\le i<j<k\le\leftrightarrow S\ \land\ S_{i;..j} = S_{j;..k}$$

Write a program to output all T-strings in alphabetical order. (The mathematician Thue proved that there are infinitely many T-strings.)

399 (reformat) Write a program to read, reformat, and write a sequence of characters. The input includes a line-break character at arbitrary places; the output should include a line-break character just after each semicolon. Whenever the input includes two consecutive stars, or two stars separated only by line-breaks, the output should replace the two stars by an up-arrow. Other than that, the output should be identical to the input. Both input and output end with a special end-character.

400 According to the definition of **result** expression given in Subsection 5.5.0, what happens to any output that occurs in the program part of programmed data? Can input be read and used? What happens to it?

401 (Huffman code) You are given a finite set of messages, and for each message, the probability of its occurrence.

(a) Write a program to find a binary code for each message. It must be possible to unambiguously decode any sequence of 0s and 1s into a sequence of messages, and the average code length (according to message frequency) must be minimum.

(b) Write the accompanying program to produce the decoder for the codes produced in part (a).

402 (matrix multiplication) Write a program to multiply two $n\times n$ matrices that uses n^2 processes, with $2\times n^2$ local channels, with execution time n .

403 (coin weights) You are given some coins, all of which have a standard weight except possibly for one of them, which may be lighter or heavier than the standard. You are also given a balance scale, and as many more standard coins as you need. Write a program to determine whether there is a nonstandard coin, and if so which, and whether it is light or heavy, in the minimum number of weighings.

404 How should "deterministic" and "nondeterministic" be defined in the presence of channels?

405 From the fixed-point equation

$twos \;=\; c!\,2.\;\; t{:=}\,t{+}1.\;\; twos$

use recursive construction to find

(a) the weakest fixed-point.

(b) a strongest implementable fixed-point.

(c) the strongest fixed-point.

406 Here are two definitions.

$$\begin{aligned} A \;=\;& \textbf{if}\ \surd c \wedge \surd d\ \textbf{then}\ c? \vee d? \\ & \textbf{else if}\ \surd c\ \textbf{then}\ c? \\ & \textbf{else if}\ \surd d\ \textbf{then}\ d? \\ & \textbf{else if}\ \mathbb{T}c_{rc} < \mathbb{T}d_{rd}\ \textbf{then}\ (t{:=}\ \mathbb{T}c_{rc}+1.\ c?) \\ & \textbf{else if}\ \mathbb{T}d_{rd} < \mathbb{T}c_{rc}\ \textbf{then}\ (t{:=}\ \mathbb{T}d_{rd}+1.\ d?) \\ & \textbf{else}\ (t{:=}\ \mathbb{T}c_{rc}+1.\ c? \vee d?) \\ B \;=\;& \textbf{if}\ \surd c \wedge \surd d\ \textbf{then}\ c? \vee d? \\ & \textbf{else if}\ \surd c\ \textbf{then}\ c? \\ & \textbf{else if}\ \surd d\ \textbf{then}\ d? \\ & \textbf{else}\ (t{:=}\ t{+}1.\ B) \end{aligned}$$

Letting time be an extended integer, prove $A = B$.

407 (input implementation) Let W be "wait for input on channel c and then read it".

(a)√ $W \;=\; t{:=}\ max\ t\ (\mathbb{T}_r + 1).\ \ c?$

Prove $W \;\Leftarrow\; \textbf{if}\ \surd c\ \textbf{then}\ c?\ \textbf{else}\ (t{:=}\ t{+}1.\ \ W)$ assuming time is an extended integer.

(b) Now let time be an extended real, redefine W appropriately, and reprove the refinement.

408 (input with timeout) As in Exercise 407, let W be "wait for input on channel c and then read it", except that if input is still not available by a deadline, an alarm should be raised.

$W \;\Leftarrow\; \textbf{if}\ t \le deadline\ \textbf{then if}\ \surd c\ \textbf{then}\ c?\ \textbf{else}\ (t{:=}\ t{+}1.\ \ W)\ \textbf{else}\ alarm$

Define W appropriately, and prove the refinement.

409 Define relation $partmerge$: $nat \to nat \to bool$ as follows:

$$\begin{aligned} & partmerge\ 0\ 0 \\ & partmerge\ (m{+}1)\ 0 \;=\; partmerge\ m\ 0\ \wedge\ \mathbb{M}c_{wc+m} = \mathbb{M}a_{ra+m} \\ & partmerge\ 0\ (n{+}1) \;=\; partmerge\ 0\ n\ \wedge\ \mathbb{M}c_{wc+n} = \mathbb{M}b_{rb+n} \\ & partmerge\ (m{+}1)\ (n{+}1) \;=\; \quad partmerge\ m\ (n{+}1)\ \wedge\ \mathbb{M}c_{wc+m+n+1} = \mathbb{M}a_{ra+m} \\ & \qquad\qquad\qquad \vee\ partmerge\ (m{+}1)\ n\ \wedge\ \mathbb{M}c_{wc+m+n+1} = \mathbb{M}b_{rb+n} \end{aligned}$$

Now $partmerge\ m\ n$ says that the first $m{+}n$ outputs on channel c are a merge of m inputs from channel a and n inputs from channel b . Define $merge$ as

$merge \;=\; (a?.\ \ c!\ a) \vee (b?.\ \ c!\ b).\ \ merge$

Prove $merge \;=\; (\forall m\cdot\ \exists n\cdot\ partmerge\ m\ n) \vee (\forall n\cdot\ \exists m\cdot\ partmerge\ m\ n)$

410 (perfect shuffle) Write a specification for a computation that repeatedly reads an input on either channel c or d . The specification says that the computation might begin with either channel, and after that it alternates.

411 (time merge) We want to repeatedly read an input on either channel c or channel d, whichever comes first, and write it on channel e. At each reading, if input is available on both channels, read either one; if it is available on just one channel, read that one; if it is available on neither channel, wait for the first one and read that one (in case of a tie, read either one).

(a)√ Write the specification formally, and then write a program.

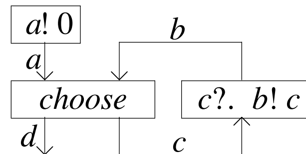
(b) Prove

$$\mathbb{T}e_{we} \;=\; max\ t\ (min\ (\mathbb{T}c_{rc})\ (\mathbb{T}d_{rd}) + 1)$$
$$\forall m, n\cdot\ \mathbb{T}e_{we+m+n+1} \le max\ (max\ (\mathbb{T}c_{rc+m})\ (\mathbb{T}d_{rd+n}))\ (\mathbb{T}e_{we+m+n}) + 1$$

412 (fairer time merge) This question is the same as the time merge (Exercise 411), but if input is available on both channels, the choice must be made the opposite way from the previous read. If, after waiting for an input, inputs arrive on both channels at the same time, the choice must be made the opposite way from the previous read.

413 In the reaction controller in Subsection 9.1.6, it is supposed that the synchronizer receives digital data from the digitizer faster than requests from the controller. Now suppose that the controller is sometimes faster than the digitizer. Modify the synchronizer so that if two or more requests arrive in a row (before new digital data arrives), the same digital data will be sent in reply to each request.

414 (Brock-Ackermann) The following picture shows a network of communicating processes.



The formal description of this network is

chan $a, b, c\cdot\ a!\ 0\ \|\ choose\ \|\ (c?.\ b!\ c)$

Formally define *choose*, add transit time, and state the output message and time if

(a) *choose* either reads from a and outputs a 0 on c and d, or reads from b and outputs a 1 on c and d. The choice is made freely.

(b) As in part (a), *choose* either reads from a and outputs a 0 on c and d, or reads from b and outputs a 1 on c and d. But this time the choice is not made freely; *choose* reads from the channel whose input is available first (if there's a tie, then take either one).

415√ (power series multiplication) Write a program to read from channel a an infinite sequence of coefficients $a_0\ a_1\ a_2\ a_3$ ... of a power series $a_0 + a_1{\times}x + a_2{\times}x^2 + a_3{\times}x^3 + ...$ and in parallel to read from channel b an infinite sequence of coefficients $b_0\ b_1\ b_2\ b_3$ ... of a power series $b_0 + b_1{\times}x + b_2{\times}x^2 + b_3{\times}x^3 + ...$ and in parallel to write on channel c the infinite sequence of coefficients $c_0\ c_1\ c_2\ c_3$ ... of the power series $c_0 + c_1{\times}x + c_2{\times}x^2 + c_3{\times}x^3 + ...$ equal to the product of the two input series. Assume that all inputs are already available; there are no input delays. Produce the outputs one per time unit.

416 (repetition) Write a program to read an infinite sequence, and after every even number of inputs, to output a boolean saying whether the second half of the input sequence is a repetition of the first half.

417 (file update) A master file of records and a transaction file of records are to be read, one record at a time, and a new file of records is to be written, one record at a time. A record consists of two text fields: a "key" field and an "info" field. The master file is kept in order of its keys, without duplicate keys, and with a final record having a sentinel key "zzzzz" guaranteed to be larger than all other keys. The transaction file is also sorted in order of its keys, with the same final sentinel key, but it may have duplicate keys. The new file is like the master file, but with changes as signified by the transaction file. If the transaction file contains a record with a key that does not appear in the master file, that record is to be added. If the transaction file contains a record with a key that does appear in the master file, that record is a change of the "info" field, unless the "info" text is the empty text, in which case it signifies record deletion. Whenever the transaction file contains a repeated key, the last record for each key determines the result.

418 (mutual exclusion) Process P is an endless repetition of a "non-critical section" PN and a "critical section" PC. Process Q is similar.

$$P = PN.\ PC.\ P$$
$$Q = QN.\ QC.\ Q$$

They are executed in parallel $(P \parallel Q)$. Specify formally that the two critical sections are never executed at the same time

(a) by inserting variables that are assigned but never used.

(b) by inserting outputs on channels that are never read.

419 (synchronous communication) A synchronous communication happens when the sender is ready to send and the receiver(s) is(are) ready to receive. Those that are ready must wait for those that are not.

(a) Design a theory of synchronous communication. For each channel, you will need only one cursor, but two (or more) time scripts. An output, as well as an input, increases the time to the maximum of the time scripts for the current message.

(b) Show how it works in some examples, including a deadlock example.

(c) Show an example that is not a deadlock with asynchronous communication, but becomes a deadlock with synchronous communication.

End of Interaction

End of Exercises

11 Reference

11.0 Justifications

This section explains some of the decisions made in choosing and presenting the material in this book. It is probably not of interest to a student whose concern is to learn the material, but it may be of interest to a teacher or researcher.

11.0.0 Notation

Whenever I had to choose between a standard notation that will do and a new notation that's perfect, I chose the standard notation. For example, to express the maximum of two numbers x and y, a function *max* is applied: $max\ x\ y$. Since maximum is symmetric and associative, it would be better to introduce a symmetric symbol like $\uparrow$ as an infix operator: $x\uparrow y$. I always do so privately, but in this book I have chosen to keep the symbols few in number and reasonably traditional. Most people seeing $max\ x\ y$ will know what is meant without prior explanation; most people seeing $x\uparrow y$ would not. In the first edition, I used λ notation for functions, thinking that it was standard. Ten years of students convinced me that it was not standard, freeing me to use a better notation in later editions.

A precedence scheme is chosen on two criteria: to minimize the need for parentheses, and to be easily remembered. The latter is helped by sticking to tradition, by placing related symbols together, and by having as few levels as possible. The two criteria are sometimes conflicting, traditions are sometimes conflicting, and the three suggestions for helping memory are sometimes conflicting. In the end, one makes a decision and lives with it. Extra parentheses can always be used, and should be used whenever structural similarities would be obscured by the precedence scheme. For the sake of structure, it would be better to give $\wedge$ and $\vee$ the same precedence, but I have stayed with tradition. The scheme in this book has more levels than I would like. I could place $\neg$ with one-operand $-$, $\wedge$ with $\times$, $\vee$ with two-operand $+$, and $\Rightarrow$ and $\Leftarrow$ with $=$ and $\neq$. This saves four levels, but is against mathematical tradition and costs a lot of parentheses. The use of large symbols $=$ $\Leftarrow$ $\Rightarrow$ with large precedence level is a novelty; I hope it is both readable and writable. Do not judge it until you have used it awhile; it saves an enormous number of parentheses. One can immediately see generalizations of this convention to all symbols and many sizes (a slippery slope).

End of Notation

11.0.1 Basic Theories

Boolean Theory sometimes goes by other names: Boolean Algebra, Propositional Calculus, Sentential Logic. Its expressions are sometimes called “propositions” or “sentences”. Sometimes a distinction is made between “terms”, which are said to denote values, and “propositions”, which are said not to denote values but instead to be true or false. A similar distinction is made between “functions”, which apply to arguments to produce values, and “predicates”, which are instantiated to become true or false. But slowly, the subject of logic is emerging from its confused, philosophical past. I consider that propositions are just boolean expressions and treat them on a par with number expressions and expressions of other types. I consider that predicates are just boolean functions. I use the same equal sign for booleans as for numbers, characters, sets, and functions. Perhaps in the future we won't feel the need to imagine abstract objects for expressions to denote; we will justify them by their practical applications. We will explain our formalisms by the rules for their use, not by their philosophy.

Why bother with “antiaxioms” and “antitheorems”? They are not traditional (in fact, I made up the words). As stated in Chapter 1, thanks to the negation operator and the Consistency Rule, we don't need to bother with them. Instead of saying that *expression* is an antitheorem, we can say that $\neg$*expression* is a theorem. Why bother with $\bot$? We could instead write $\neg\top$. One reason is just that it is shorter to say “antitheorem” than to say “negation of a theorem”. Another reason is to help make clear the important difference between “disprovable” and “not provable”. Another reason is that some logics do not use the negation operator and the Consistency Rule. The logic in this book is “classical logic”; “constructive logic” omits the Completion Rule; “evaluation logic” omits both the Consistency Rule and the Completion Rule.

Some books present proof rules (and axioms) with the aid of a formal metanotation. In this book, there is no formal metalanguage; the metalanguage is English. A formal metalanguage may be considered helpful (though it is not necessary) for the presentation and comparison of a variety of competing formalisms, and for proving theorems about formalisms. But in this book, only one formalism is presented. The burden of learning another formalism first, for the purpose of presenting the main formalism, is unnecessary. A formal metanotation [/] for substitution would allow me to write the function application rule as

$$\langle v \rightarrow b\rangle\, a \;=\; b[a/v]$$

but then I would have to explain that $b[a/v]$ means “substitute a for v in b”. I may as well say directly

$$\langle v \rightarrow b\rangle\, a \;=\; (\text{substitute } a \text{ for } v \text{ in } b)$$

A proof syntax (formalizing the “hints”) would be necessary if we were using an automated prover, but in this book it is unnecessary and I have not introduced one.

Some authors may distinguish “axiom” from “axiom schema”, the latter having variables which can be instantiated to produce axioms; I have used the term “axiom” for both. I have also used the term “law” as a synonym for “theorem” (I would prefer to reduce my vocabulary, but both words are well established). Other books may distinguish them by the presence or absence of variables, or they may use “law” to mean “we would like it to be a theorem but we haven't yet designed an appropriate theory”.

I have taken a few liberties with the names of some axioms and laws. What I have called “transparency” is often called “substitution of equals for equals”, which is longer and doesn't quite make sense. Each of my Laws of Portation is historically two laws, one an implication in one direction, and the other an implication in the other direction. One was called “Importation”, and the other “Exportation”, but I can never remember which was which.

End of Basic Theories

11.0.2 Basic Data Structures

Why bother with bunches? Don't sets work just as well? Aren't bunches really just sets but using a peculiar notation and terminology? The answer is no, but let's take it slowly. Suppose we just present sets. We want to be able to write $\{1, 3, 7\}$ and similar expressions, and we might describe these set expressions with a little grammar like this:

set = "{" *contents* "}"
contents = *number*
| *set*
| *contents* "," *contents*

We will want to say that the order of elements in a set is irrelevant so that $\{1, 2\} = \{2, 1\}$; the best way to say it is formally: $A,B = B,A$ (comma is symmetric, or commutative). Next, we want to say

that repetitions of elements in a set are irrelevant so that $\{3, 3\} = \{3\}$; the best way to say that is $A,A = A$ (comma is idempotent). What we are doing here is inventing bunches, but calling them "contents" of a set. And note that the grammar is equating bunches; the string catenations (denoted here by juxtaposition) distribute over the elements of their operands, and the alternations (denoted here by vertical bars) are bunch unions.

When a child first learns about sets, there is often an initial hurdle: that a set with one element is not the same as the element. How much easier it would be if a set were presented as packaging: a bag with an apple in it is obviously not the same as the apple. Just as $\{2\}$ and 2 differ, so $\{2,7\}$ and 2,7 differ. Bunch Theory tells us about aggregation; Set Theory tells us about packaging. The two are independent.

We could define sets without relying on bunches (as has been done for many years), and we could use sets wherever I have used bunches. In that sense, bunches are unnecessary. Similarly we could define lists without relying on sets (as I did in this book), and we could always use lists in place of sets. In that sense, sets are unnecessary. But sets are a beautiful data structure that introduces one idea (packaging), and I prefer to keep them. Similarly bunches are a beautiful data structure that introduces one idea (aggregation), and I prefer to keep them. I always prefer to use the simplest structure that is adequate for its purpose.

The subject of functional programming has suffered from an inability to express nondeterminism conveniently. To say something about a value, but not pin it down completely, one can express the set of possible values. Unfortunately, sets do not reduce properly to the deterministic case; in this context it is again a problem that a set containing one element is not equal to the element. What is wanted is bunches. One can always regard a bunch as a "nondeterministic value".

Bunches have also been used in this book as a "type theory". Surely it is discouraging to others, as it is to me, to see type theory duplicating all the operators of its value space: for each operation on values, there is a corresponding operation on type spaces. By using bunches, this duplication is eliminated.

Many mathematicians consider that curly brackets and commas are just syntax, and syntax is annoying and unimportant, though necessary. I have treated them as operators, with algebraic properties (in Section 2.1 on Set Theory, we see that curly brackets have an inverse). This continues a very long, historical trend. For example, $=$ was at first just a syntax for the informal statement that two things are (in some way) the same, but now it is a formal operator with algebraic properties.

In many papers there is a little apology as the author explains that the notation for catenation of lists will be abused by sometimes catenating a list and an item. Or perhaps there are three catenation notations: one to catenate two lists, one to prepend an item to a list, and one to append an item to a list. The poor author has to fight with unwanted packaging provided by lists in order to get the sequencing. I offer these authors strings: sequencing without packaging. (Of course, they can be packaged into lists whenever wanted. I am not taking away lists.)

End of Basic Data Structures

11.0.3 Function Theory

I have used the words "local" and "nonlocal" where others might use the words "bound" and "free", or "local" and "global", or "hidden" and "visible", or "private" and "public". The tradition in logic, which I have not followed, is to begin with all possible variables (infinitely many of them) already "existing". The function notation $\langle\ \rangle$ is said to "bind" variables, and any variable that is not bound remains "free". For example, $\langle x\text{: } int \rightarrow x+y\rangle$ has bound variable x, free variable y, and infinitely many other free variables. In this book, variables do not automatically "exist"; they are introduced (rather than bound) either formally using the function notation, or informally by saying in English what they are.

The quantifier formed from *max* is called *MAX* even though its result may not be any result of the function it is applied to; the name "least upper bound" is traditional. Similarly for *MIN*, which is traditionally called "greatest lower bound".

I have ignored the traditional question of the "existence" of limits; in cases where traditionally a limit does not "exist", the Limit Law does not tell us exactly what the limit is, but it might still tell us something useful.

End of Function Theory

11.0.4 Program Theory

Assignment could have been defined as

$$x{:=}\ e \quad = \quad defined\ \text{"}e\text{"} \land e\text{: } T \ \Rightarrow\ x'{=}e \land y'{=}y \land \ldots$$

where *defined* rules out expressions like $1/0$, and T is the type of variable x. I left out *defined* because a complete definition of it is impossible, a reasonably complete definition is as complicated as all of program theory, and it serves no purpose. The antecedent $e\text{: } T$ would be useful, making the assignment $n{:=}\ n{-}1$ implementable when n is a natural variable. But its benefit is not worth its trouble, since the same check is made at every dependent composition. Even worse, we would lose the Substitution Law; we want $(n{:=}\,{-}1.\ n{\ge}0)$ to be $\bot$.

Since the design of Algol-60, sequential execution has often been represented by a semi-colon. The semi-colon is unavailable to me for this purpose because I used it for string catenation. Dependent composition is a kind of product, so I hope a period will be an acceptable symbol. I considered switching the two, using semi-colon for dependent composition and a period for string catenation, but the latter did not work well.

In English, the word "precondition" means "something that is necessary beforehand". In many programming books, the word "precondition" is used to mean "something that is sufficient beforehand". In those books, "weakest precondition" means "necessary and sufficient precondition", which I have called "exact precondition".

In the earliest and still best-known theory of programming, we specify that variable x is to be increased as follows:

$$\{x = X\}\ S\ \{x > X\}$$

We are supposed to know that x is a state variable, that X is a local variable to this specification whose purpose is to relate the initial and final value of x, and that S is also local to the specification and is a place-holder for a program. Neither X nor S will appear in a program that refines this specification. Formally, X and S are quantified as follows:

$$\S S\cdot\ \forall X\cdot\ \{x = X\}\ S\ \{x > X\}$$

In the theory of weakest preconditions, the equivalent specification looks similar:

$\S S\cdot\ \forall X\cdot\ x{=}X \Rightarrow wp\ S\ (x{>}X)$

There are two problems with these notations. One is that they do not provide any way of referring to both the prestate and the poststate, hence the introduction of X. This is solved in the Vienna Development Method, in which the same specification is

$\S S\cdot\ \{\top\}\ S\ \{x' > x\}$

The other problem is that the programming language and specification language are disjoint, hence the introduction of S. In my theory, the programming language is a sublanguage of the specification language. The specification that x is to be increased is

$x' > x$

The same single-expression double-state specifications are used in Z, but refinement is rather complicated. In Z, P is refined by S if and only if

$\forall\sigma\cdot\ (\exists\sigma'\cdot\ P) \Rightarrow (\exists\sigma'\cdot\ S) \wedge (\forall\sigma'\cdot\ P \Leftarrow S)$

In the early theory, $\S S\cdot\ \{P\}\ S\ \{Q\}$ is refined by $\S S\cdot\ \{R\}\ S\ \{U\}$ if and only if

$\forall\sigma\cdot\ P\ \Rightarrow\ R \wedge (Q \Leftarrow U)$

In my theory, P is refined by S if and only if

$\forall\sigma, \sigma'\cdot\ P \Leftarrow S$

Since refinement is what we must prove when programming, it is best to make refinement as simple as possible.

One might suppose that any type of mathematical expression can be used as a specification: whatever works. A specification of something, whether cars or computations, distinguishes those things that satisfy it from those that don't. Observation of something provides values for certain variables, and on the basis of those values we must be able to determine whether the something satisfies the specification. Thus we have a specification, some values for variables, and two possible outcomes. That is exactly the job of a boolean expression: a specification (of anything) really is a boolean expression. If instead we use a pair of predicates, or a function from predicates to predicates, or anything else, we make our specifications in an indirect way, and we make the task of determining satisfaction more difficult.

One might suppose that any boolean expression can be used to specify any computer behavior: whatever correspondence works. In Z, the expression $\top$ is used to specify (describe) terminating computations, and $\bot$ is used to specify (describe) nonterminating computations. The reasoning is something like this: $\bot$ is the specification for which there is no satisfactory final state; an infinite computation is behavior for which there is no final state; hence $\bot$ represents infinite computation. Although we cannot observe a "final" state of an infinite computation, we can observe, simply by waiting 10 time units, that it satisfies $t' > t{+}10$, and it does not satisfy $t' \le t{+}10$. Thus it ought to satisfy any specification implied by $t' > t{+}10$, including $\top$, and it ought not to satisfy any specification that implies $t' \le t{+}10$, including $\bot$. Since $\bot$ is not true of anything, it does not describe anything. A specification is a description, and $\bot$ is not satisfiable, not even by nonterminating computations. Since $\top$ is true of everything, it describes everything, even nonterminating computations. To say that P refines Q is to say that all behavior satisfying P also satisfies Q, which is just implication. The correspondence between specifications and computer behavior is not arbitrary.

As pointed out in Chapter 4, specifications such as $x'{=}2 \wedge t'{=}\infty$ that talk about the "final" values of variables at time infinity are strange. I could change the theory to prevent any mention of results at time infinity, but I do not for two reasons: it would make the theory more complicated, and I need to distinguish among infinite loops when I introduce interactions (Chapter 9).

---End of Program Theory

11.0.5 Programming Language

The form of variable declaration given in Chapter 5 assigns the new local variable an arbitrary value of its type. Thus, for example, if y and z are integer variables, then

$$\mathbf{var}\ x{:}\ nat\cdot\ y{:=}\ x\ \ =\ \ y'{:}\ nat\ \wedge\ z'{=}z$$

For ease of implementation and speed of execution, this is much better than initialization with "the undefined value". For error detection, it is no worse, assuming that we prove all our refinements. Furthermore, there are circumstances in which arbitrary initialization is exactly what's wanted (see Exercise 270 (majority vote)). However, if we do not prove all our refinements, initialization with *undefined* provides a measure of protection. If we allow the generic operators ($=$, $\neq$, **if then else**) to apply to *undefined* , then we can prove trivialities like $undefined = undefined$. If not, then we can prove nothing at all about *undefined* . Some programming languages seek to eliminate the error of using an uninitialized variable by initializing each variable to a standard value of its type. Such languages achieve the worst of all worlds: they are not as efficient as arbitrary initialization; and they eliminate only the error detection, not the error.

An alternative way to define variable declaration is

$$\mathbf{var}\ x{:}\ T\ \ =\ \ x'{:}\ T\ \wedge\ ok$$

which starts the scope of x , and

$$\mathbf{end}\ x\ \ =\ \ ok$$

which ends the scope of x . In each of these programs, ok maintains the other variables. This kind of declaration does not require scopes to be nested; they can be overlapped.

The most widely known and used rule for **while**-loops is the Method of Invariants and Variants. Let I be a precondition (called the "invariant") and let I' be the corresponding postcondition. Let v be an integer expression (called the "variant" or "bound function") and let v' be the corresponding expression with primed variables. The Rule of Invariants and Variants says:

$$I \Rightarrow I' \wedge \neg b'\ \ \Leftarrow\ \ \mathbf{while}\ b\ \mathbf{do}\ I \wedge b\ \Rightarrow\ I' \wedge 0{\le}v'{<}v$$

The rule says, very roughly, that if the body of the loop maintains the invariant and decreases the variant but not below zero, then the loop maintains the invariant and negates the loop condition. For example, to prove

$$s' = s + \Sigma\, L\, [n;..\#L]\ \ \Leftarrow\ \ \mathbf{while}\ n{\ne}\#L\ \mathbf{do}\ (s{:=}\ s + Ln.\ \ n{:=}\ n{+}1)$$

we must invent an invariant $s + \Sigma\, L\, [n;..\#L] = \Sigma L$ and a variant $\#L - n$ and prove both

$$\begin{array}{ll} & s' = s + \Sigma\, L\, [n;..\#L] \\ \Leftarrow & s + \Sigma\, L\, [n;..\#L] = \Sigma L\ \Rightarrow\ s' + \Sigma\, L\, [n';..\#L] = \Sigma L\ \wedge\ n'{=}\#L \end{array}$$

and

$$\begin{array}{ll} & s + \Sigma\, L\, [n;..\#L] = \Sigma L\ \wedge\ n{\ne}\#L\ \Rightarrow\ s' + \Sigma\, L\, [n';..\#L] = \Sigma L\ \wedge\ 0 \le \#L - n' < \#L - n \\ \Leftarrow & s{:=}\ s + Ln.\ \ n{:=}\ n{+}1 \end{array}$$

The proof method given in Chapter 5 is easier and more information (time) is obtained. Sometimes the Method of Invariants and Variants requires the introduction of extra constants (mathematical variables) not required by the proof method in Chapter 5. For example, to add 1 to each item in list L requires introducing a new list constant to stand for the initial value of L .

Subsection 5.2.0 says that $W\ \Leftarrow\ \mathbf{while}\ b\ \mathbf{do}\ P$ is an abbreviation, but it is a dangerously misleading one. It looks like W is being refined by a program involving only b and P ; in fact, W is being refined by a program involving b , P , and W .

Probability Theory would be simpler if all real numbers were probabilities, instead of just the reals in the closed interval from 0 to 1 , in which case I would add the axioms $\top{=}\infty$ and $\bot{=}{-}\infty$; but it is not my purpose in this book to invent a better probability theory. For probabilistic

programming, my first approach was to reinterpret the types of variables as probability distributions expressed as functions. If x was a variable of type T, it becomes a variable of type $T\rightarrow prob$ such that $\Sigma x = \Sigma x' = 1$. All operators then need to be extended to distributions expressed as functions. Although this approach works, it was too low-level; a distribution expressed as a function tells us about the probability of its variables by their positions in an argument list, rather than by their names.

The subject of programming has often been mistaken for the learning of a large number of programming language "features". This mistake has been made of both imperative and functional programming. Of course, each fancy operator provided in a programming language makes the solution of some problems easy. In functional programming, an operator called "fold" or "reduce" is often presented; it is a useful generalization of some quantifiers. Its symbol might be $/$ and it takes as left operand a two-operand operator and as right operand a list. The list summation problem is solved as $+/L$. The search problem could similarly be solved by the use of an appropriate search operator, and it would be a most useful exercise to design and implement such an operator. This exercise cannot be undertaken by someone whose only programming ability is to find an already implemented operator and apply it. The purpose of this book is to teach the necessary programming skills.

As our examples illustrate, functional programming and imperative programming are essentially the same: the same problem in the two styles requires the same steps in its solution. They have been thought to be different for the following reasons: imperative programmers adhere to clumsy loop notations, complicating proofs; functional programmers adhere to equality, rather than refinement, making nondeterminism difficult.

End of Programming Language

11.0.6 Recursive Definition

Recursive construction has always been done by taking the limit of a sequence of approximations. My innovation is to substitute ∞ for the index in the sequence; this is a lot easier than finding a limit. Substituting ∞ is not guaranteed to produce the desired fixed-point, but neither is finding the limit. Substituting ∞ works well except in examples contrived to show its limitation.

End of Recursive Definition

11.0.7 Theory Design and Implementation

I used the term "data transformation" instead of the term "data refinement" used by others. I don't see any reason to consider one space more "abstract" and another more "concrete". What I call a "data transformer" is sometimes called "abstraction relation", "linking invariant", "gluing relation", "retrieve function", or "data invariant".

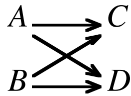
The incompleteness of data transformation is demonstrated with an example carefully crafted to show the incompleteness, not one that would ever arise in practice. I prefer to stay with the simple rule that is adequate for all transformations that will ever arise in any problem other than a demonstration of theoretical incompleteness, rather than to switch to a more complicated rule, or combination of rules, that are complete. To regain completeness, all we need is the normal mathematical practice of introducing local variables. Variables for this purpose have been called "bound variables", "logical constants", "specification variables", "ghost variables", "abstract variables", and "prophesy variables", by different authors.

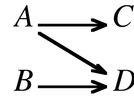
End of Theory Design and Implementation

11.0.8 Concurrency

In FORTRAN (prior to 1977) we could have a sequential composition of **if**-statements, but we could not have an **if**-statement containing a sequential composition. In ALGOL the syntax was fully recursive; sequential and conditional compositions could be nested, each within the other. Did we learn a lesson? Apparently we did not learn a very general one: we now seem happy to have a parallel composition of sequential compositions, but very reluctant to have a sequential composition of parallel compositions. So in currently popular languages, a parallel composition can occur only as the outermost construct.

Here are two execution patterns.





As we saw in Chapter 8, the first pattern can be expressed as $((A \parallel B).\ \ (C \parallel D))$ without any synchronization primitives. But the second pattern cannot be expressed using only parallel and sequential composition. This second pattern occurs in the buffer program.

In the first edition of this book, parallel composition was defined for processes having the same state space (semi-dependent composition). That definition was more complicated than the present one (see Exercise 378), but in theory, it eliminated the need to partition the variables. In practice, however, the variables were always partitioned, so in the present edition the simpler definition (independent composition) is used.

End of Concurrency

11.0.9 Interaction

If x is an interactive variable, $(t'=\infty.\ \ x:=2.\ \ x:=3)$ is unfortunately $\bot$; thus the theory of interactive variables is slightly too strong. Likewise, $(wc'=\infty.\ \ c!\ 2.\ \ c!\ 3)$ is unfortunately $\bot$; thus the theory of communication is slightly too strong. To eliminate these unwanted results we would have to weaken the definition of assignment to an interactive variable with the antecedent $t'<\infty$, and weaken output with the antecedent $wc'<\infty$. But I think these pathological cases are not worth complicating the theory.

In the formula for implementability, there is no conjunct $r' \le w'$ saying that the read cursor must not get ahead of the write cursor. In Subsection 9.1.8 on deadlock we see that it can indeed happen. Of course, it takes infinite time to do so. In the deadlock examples, we can prove that the time is infinite. But there is a mild weakness in the theory. Consider this example.

$$\begin{aligned} & \textbf{chan}\ c\cdot\ t:=max\ t\ (\mathbb{T}_r+1).\ \ c? \\ = \ & \exists \mathbb{M}, \mathbb{T}, r, r', w, w'\cdot\ t' = max\ t\ (\mathbb{T}_0+1)\ \land\ r'=1\ \land\ w'=0 \\ = \ & t' \ge t \end{aligned}$$

We might like to prove $t'=\infty$. To get this answer, we must strengthen the definition of local channel declaration by adding the conjunct $\mathbb{T}_{w'} \ge t'$. I prefer the simpler, weaker theory.

End of Interaction

We could talk about a structure of channels, and about indexed processes. We could talk about a parallel **for**-loop. There is always something more to say, but we have to stop somewhere.

End of Justifications

11.1 Sources

Ideas do not come out of nowhere. They are the result of one's education, one's culture, and one's interactions with acquaintances. I would like to acknowledge all those people who have influenced me and enabled me to write this book. I will probably fail to mention people who have influenced me indirectly, even though the influence may be strong. I may fail to thank people who gave me good ideas on a bad day, when I was not ready to understand. I will fail to give credit to people who worked independently, whose ideas may be the same as or better than those that happened to reach my eyes and ears. To all such people, I apologize. I do not believe anyone can really take credit for an idea. Ideally, our research should be done for the good of everyone, perhaps also for the pleasure of it, but not for the personal glory. Still, it is disappointing to be missed. Here then is the best accounting of my sources that I can provide.

The early work in this subject is due to Alan Turing (1949), Peter Naur (1966), Robert Floyd (1967), Tony Hoare (1969), Rod Burstall (1969), and Dana Scott and Christopher Strachey (1970). (See the Bibliography, which follows.) My own introduction to the subject was a book by Edsger Dijkstra (1976); after reading it I took my first steps toward formalizing refinement (1976). Further steps in that same direction were taken by Ralph Back (1978), though I did not learn of them until 1984. The first textbooks on the subject began to appear, including one by me (1984). That work was based on Dijkstra's weakest precondition predicate transformer, and work continues today on that same basis. I highly recommend the book *Refinement Calculus* by Ralph Back and Joachim vonWright (1998).

In the meantime, Tony Hoare (1978, 1981) was developing communicating sequential processes. During a term at Oxford in 1981 I realized that they could be described as predicates, and published a predicate model (1981, 1983). It soon became apparent that the same sort of description, a single boolean expression, could be used for any kind of computation, and indeed for anything else; in retrospect, it should have been obvious from the start. The result was a series of papers (1984, 1986, 1988, 1989, 1990, 1994, 1998, 1999, 2011) leading to the present book.

The importance of format in expressions and proofs was well expressed by Netty van Gasteren (1990). The symbols ¢ and $ for bunch and set cardinality were suggested by Chris Lengauer. The word “conflation” was suggested by Doug McIlroy. Exercise 30 (bracket algebra) comes from Philip Meguire, who got it from George Spencer-Brown, who got it from Charles Sanders Peirce. The value of indexing from 0 was taught to me by Edsger Dijkstra. Joe Morris and Alex Bunkenburg (2001) found and fixed a problem with bunch theory. The word “apposition” and the idea to which it applies come from Lambert Meertens (1986). Peter Kanareitsev helped with higher-order functions. Alan Rosenthal suggested that I stop worrying about when limits “exist”, and just write the axioms describing them; I hope that removes the last vestige of Platonism from the mathematics, though some remains in the English. My Refinement by Parts law was made more general by Theo Norvell. I learned the use of a timing variable from Chris Lengauer (1981), who credits Mary Shaw; we were using weakest preconditions then, so our time variables ran down instead of up. The recursive measure of time is inspired by the work of Paul Caspi, Nicolas Halbwachs, Daniel Pilaud, and John Plaice (1987); in their language LUSTRE, each iteration of a loop takes time 1 , and all else is free. I learned to discount termination by itself, with no time bound, in discussions with Andrew Malton, and from an example of Hendrik Boom (1982). I was told the logarithmic solution to the Fibonacci number problem by Wlad Turski, who learned it while visiting the University of Guelph. My incorrect version of local variable declaration was corrected by Andrew Malton. Local variable suspension is adapted from Carroll Morgan (1990). The **for**-loop rule was influenced by Victor Kwan and Emil Sekerinski. The backtracking implementation

of unimplementable specifications is an adaptation of a technique due to Greg Nelson (1989) for implementing angelic nondeterminism. Carroll Morgan and Annabelle McIver (1996) suggested probabilities as observable quantities, and Exercise 284 (Mr.Bean's socks) comes from them. The use of bunches for nondeterminism in functional programming and for function refinement is joint work with Theo Norvell (1992). Theo also added the timing to the recursive definition of **while**-loops (1997). The style of data-type theories (data-stack, data-queue, data-tree) comes from John Guttag and Jim Horning (1978). The implementation of data-trees was influenced by Tony Hoare (1975). Program-tree theory went through successive versions due to Theo Norvell, Yannis Kassios, and Peter Kanareitsev. I learned about data transformation from He Jifeng and Carroll Morgan, based on earlier work by Tony Hoare (1972); the formulation here is my own, but I checked it for equivalence with those in Wei Chen and Jan Tijmen Udding (1989). Theo Norvell provided the criterion for data transformers. The second data transformation example (take a number) is adapted from a resource allocation example of Carroll Morgan (1990). The final data transformation example showing incompleteness was invented by Paul Gardiner and Carroll Morgan (1993). For an encyclopedic treatment of data transformers, see the book by Willem-Paul deRoever and Kai Engelhardt (1998). I published various formulations of independent (parallel) composition (1981, 1984, 1990, 1994); the one in the first edition of this book is due to Theo Norvell and appears in this edition as Exercise 378 (semi-dependent composition), and is used in recent work by Hoare and He (1998); for this edition I was persuaded by Leslie Lamport to return to my earlier (1984, 1990) version: simple conjunction. Section 8.1 on sequential to parallel transformation is joint work with Chris Lengauer (1981); he has since made great advances in the automatic production of highly parallel, systolic computations from ordinary sequential, imperative programs. The thermostat example is a simplification and adaptation of a similar example due to Anders Ravn, Erling Sørensen, and Hans Rischel (1990). The form of communication was influenced by Gilles Kahn (1974). Time scripts were suggested by Theo Norvell. The input check is an invention of Alain Martin (1985), which he called the “probe”. Monitors were invented by Per Brinch Hansen (1973) and Tony Hoare (1974). The power series multiplication is from Doug McIlroy (1990), who credits Gilles Kahn. Many of the exercises were given to me by Wim Feijen for my earlier book (1984); they were developed by Edsger Dijkstra, Wim Feijen, Netty van Gasteren, and Martin Rem for examinations at the Technical University of Eindhoven; they have since appeared in a book by Edsger Dijkstra and Wim Feijen (1988). Some exercises come from a series of journal articles by Martin Rem (1983,..1991). Other exercises were taken from a great variety of sources too numerous to mention.

End of Sources

11.2 Bibliography

R.-J.R.Back: “on the Correctness of Refinement Steps in Program Development”, University of Helsinki, Department of Computer Science, Report A-1978-4, 1978

R.-J.R.Back: “a Calculus of Refinement for Program Derivations”, *Acta Informatica*, volume 25, pages 593,..625, 1988

R.-J.R.Back, J.vonWright: *Refinement Calculus: a Systematic Introduction*, Springer, 1998

H.J.Boom: “a Weaker Precondition for Loops”, *ACM Transactions on Programming Languages and Systems*, volume 4, number 4, pages 668,..678, 1982

P.BrinchHansen: “Concurrent Programming Concepts”, *ACM Computing Surveys*, volume 5, pages 223,..246, 1973 December

R.Burstall: “Proving Properties of Programs by Structural Induction”, University of Edinburgh, Report 17 DMIP, 1968; also *Computer Journal*, volume 12, number 1, pages 41,..49, 1969

P.Caspi, N.Halbwachs, D.Pilaud, J.A.Plaice: “LUSTRE: a Declarative Language for Programming Synchronous Systems”, *fourteenth annual ACM Symposium on Principles of Programming Languages*, pages 178,..189, Munich, 1987

K.M.Chandy, J.Misra: *Parallel Program Design: a Foundation*, Addison-Wesley, 1988

W.Chen, J.T.Udding: “Toward a Calculus of Data Refinement”, J.L.A.van de Snepscheut (editor): *Mathematics of Program Construction*, Springer, Lecture Notes in Computer Science, volume 375, pages 197,..219, 1989

E.W.Dijkstra: “Guarded Commands, Nondeterminacy, and Formal Derivation of Programs”, *Communications ACM*, volume 18, number 8, pages 453,..458, 1975 August

E.W.Dijkstra: *a Discipline of Programming*, Prentice-Hall, 1976

E.W.Dijkstra, W.H.J.Feijen: *a Method of Programming*, Addison-Wesley, 1988

R.W.Floyd: “Assigning Meanings to Programs”, *Proceedings of the American Society, Symposium on Applied Mathematics*, volume 19, pages 19,..32, 1967

P.H.B.Gardiner, C.C.Morgan: “a Single Complete Rule for Data Refinement”, *Formal Aspects of Computing*, volume 5, number 4, pages 367,..383, 1993

A.J.M.vanGasteren: “on the Shape of Mathematical Arguments”, Springer-Verlag Lecture Notes in Computer Science, 1990

J.V.Guttag, J.J.Horning: “the Algebraic Specification of Abstract Data Types”, *Acta Informatica*, volume 10, pages 27,..53, 1978

E.C.R.Hehner: "**do** considered **od**: a Contribution to the Programming Calculus", University of Toronto, Technical Report CSRG-75, 1976 November; also *Acta Informatica*, volume 11, pages 287,..305, 1979

E.C.R.Hehner: "Bunch Theory: a Simple Set Theory for Computer Science", University of Toronto, Technical Report CSRG-102, 1979 July; also *Information Processing Letters*, volume 12, number 1, pages 26,..31, 1981 February

E.C.R.Hehner, C.A.R.Hoare: "a More Complete Model of Communicating Processes", University of Toronto, Technical Report CSRG-134, 1981 September; also *Theoretical Computer Science*, volume 26, numbers 1 and 2, pages 105,..121, 1983 September

E.C.R.Hehner: "Predicative Programming", *Communications ACM*, volume 27, number 2, pages 134,..152, 1984 February

E.C.R.Hehner: *the Logic of Programming*, Prentice-Hall International, 1984

E.C.R.Hehner, L.E.Gupta, A.J.Malton: "Predicative Methodology", *Acta Informatica*, volume 23, number 5, pages 487,..506, 1986

E.C.R.Hehner, A.J.Malton: "Termination Conventions and Comparative Semantics", *Acta Informatica*, volume 25, number 1, pages 1,..15, 1988 January

E.C.R.Hehner: "Termination is Timing", Conference on Mathematics of Program Construction, The Netherlands, Enschede, 1989 June; also J.L.A.van de Snepscheut (editor): *Mathematics of Program Construction*, Springer-Verlag, Lecture Notes in Computer Science volume 375, pages 36,..48, 1989

E.C.R.Hehner: "a Practical Theory of Programming", *Science of Computer Programming,* volume 14, numbers 2 and 3, pages 133,..159, 1990

E.C.R.Hehner: "Abstractions of Time", *a Classical Mind,* chapter 12, Prentice-Hall, 1994

E.C.R.Hehner: "Formalization of Time and Space", *Formal Aspects of Computing,* volume 10, pages 290,..307, 1998

E.C.R.Hehner, A.M.Gravell: "Refinement Semantics and Loop Rules", FM'99 World Congress on Formal Methods, pages 20,..25, Toulouse France, 1999 September

E.C.R.Hehner: "Specifications, Programs, and Total Correctness", *Science of Computer Programming* volume 34, pages 191,..206, 1999

E.C.R.Hehner: "a Probability Perspective", *Formal Aspects of Computing* volume 23, number 4, pages 391,..420, 2011

C.A.R.Hoare: "an Axiomatic Basis for Computer Programming", *Communications ACM*, volume 12, number 10, pages 576,..581, 583, 1969 October

C.A.R.Hoare: "Proof of Correctness of Data Representations", *Acta Informatica*, volume 1, number 4, pages 271,..282, 1972

C.A.R.Hoare: “Monitors: an Operating System Structuring Concept”, *Communications ACM*, volume 17, number 10, pages 549,..558, 1974 October

C.A.R.Hoare: “Recursive Data Structures”, *International Journal of Computer and Information Sciences*, volume 4, number 2, pages 105,..133, 1975 June

C.A.R.Hoare: “Communicating Sequential Processes”, *Communications ACM*, volume 21, number 8, pages 666,..678, 1978 August

C.A.R.Hoare: “a Calculus of Total Correctness for Communicating Processes”, *Science of Computer Programming*, volume 1, numbers 1 and 2, pages 49,..73, 1981 October

C.A.R.Hoare: “Programs are Predicates”, in C.A.R.Hoare, J.C.Shepherdson (editors): *Mathematical Logic and Programming Languages*, Prentice-Hall Intenational, pages 141,..155, 1985

C.A.R.Hoare, I.J.Hayes, J.He, C.C.Morgan, A.W.Roscoe, J.W.Sanders, I.H.Sørensen, J.M.Spivey, B.A.Sufrin: “the Laws of Programming”, *Communications ACM*, volume 30, number 8, pages 672,..688, 1987 August

C.A.R.Hoare, J.He: *Unifying Theories of Programming*, Prentice-Hall, 1998

C.B.Jones: *Software Development: a Rigorous Approach*, Prentice-Hall International, 1980

C.B.Jones: *Systematic Software Development using VDM*, Prentice-Hall International, 1990

G.Kahn: “the Semantics of a Simple Language for Parallel Programming”, *Information Processing 74*, North-Holland, Proceeding of IFIP Congress, 1974

C.Lengauer, E.C.R.Hehner: “a Methodology for Programming with Concurrency”, CONPAR 81, Nürnberg, 1981 June 10,..13; also Springer-Verlag, Lecture Notes in Computer Science volume 111, pages 259,..271, 1981 June; also *Science of Computer Programming*, volume 2, pages 1,..53 , 1982

A.J.Martin: “the Probe: an Addition to Communication Primitives”, *Information Processing Letters*, volume 20, number 3, pages 125,..131, 1985

J.McCarthy: “a Basis for a Mathematical Theory of Computation”, *Proceedings of the Western Joint Computer Conference*, pages 225,..239, Los Angeles, 1961 May; also Computer Programming and Formal Systems, North-Holland, pages 33,..71, 1963

M.D.McIlroy: “Squinting at Power Series”, *Software Practice and Experience*, volume 20, number 7, pages 661,..684, 1990 July

L.G.L.T.Meertens: “Algorithmics — towards Programming as a Mathematical Activity”, Proceedings of CWI Symposium on Mathematics and Computer Science, North-Holland, *CWI Monographs*, volume 1, pages 289,..335, 1986

C.C.Morgan: “the Specification Statement”, *ACM Transactions on Programming Languages and Systems*, volume 10, number 3, pages 403,..420, 1988 July

C.C.Morgan: *Programming from Specifications*, Prentice-Hall International, 1990

C.C.Morgan, A.K.McIver, K.Seidel, J.W.Sanders: “Probabilistic Predicate Transformers”, *ACM Transactions on Programming Languages and Systems*, volume 18, number 3, pages 325,..354, 1996 May

J.M.Morris: “a Theoretical Basis for Stepwise Refinement and the Programming Calculus”, *Science of Computer Programming*, volume 9, pages 287,..307, 1987

J.M.Morris, A.Bunkenburg: “a Theory of Bunches”, *Acta Informatica*, volume 37, number 8, pages 541,..563, 2001 May

P.Naur: “Proof of Algorithms by General Snapshots”, *BIT*, volume 6, number 4, pages 310,..317, 1966

G.Nelson: “a Generalization of Dijkstra's Calculus”, *ACM Transactions on Programming Languages and Systems*, volume 11, number 4, pages 517,..562, 1989 October

T.S.Norvell: “Predicative Semantics of Loops”, *Algorithmic Languages and Calculi*, Chapman-Hall, 1997

T.S.Norvell, E.C.R.Hehner: “Logical Specifications for Functional Programs”, International Conference on Mathematics of Program Construction, Oxford, 1992 June

A.P.Ravn, E.V.Sørensen, H.Rischel: “Control Program for a Gas Burner”, Technical University of Denmark, Department of Computer Scence, 1990 March

M.Rem: “Small Programming Exercises”, articles in *Science of Computer Programming*, 1983,..1991

W.-P.deRoever, K.Engelhardt: *Data Refinement: Model-Oriented Proof Methods and their Comparisons*, tracts in Theoretical Computer Science volume 47, Cambridge University Press, 1998

D.S.Scott, C.Strachey: “Outline of a Mathematical Theory of Computation”, technical report PRG-2, Oxford University, 1970; also *Proceedings of the fourth annual Princeton Conference on Information Sciences and Systems*, pages 169,..177, 1970

K.Seidel, C.Morgan, A.K.McIver: “an Introduction to Probabilistic Predicate Transformers”, technical report PRG-TR-6-96, Oxford University, 1996

J.M.Spivey: *the Z Notation – a Reference Manual*, Prentice-Hall International, 1989

A.M.Turing: “Checking a Large Routine”, Cambridge University, Report on a Conference on High Speed Automatic Calculating Machines, pages 67,..70, 1949

End of Bibliography

11.3 Index

abstract space 207
- variable 208

abstraction relation 207
Ackermann 173
algebra, bracket 153
- linear 189

alias 81
all present 168
almost sorted segment 174
alternating sum 166
antecedent 3
antiaxiom 6, 202
antimonotonic 9
antitheorem 3, 202
application 24
apposition 31
approximate search 171
argument 24, 80
arithmetic 12, 174
arity 157
array 22, 68
- element assignment 68
- sparse 193

assertion 77
assignment 36
- array element 68
- initializing 67
- nondeterministic 177

average 84
- space 64

axiom 6
- rule 5
- schema 202

backtracking 77
Backus-Naur Form 185
batch processing 134
binary decision diagram 149
- exponentiation 167, 45
- logarithm natural 169
- search 53, 167
- tree 192

bit sum 171
bitonic list 158
blackjack 85, 180
body 23
Boole's booleans 180
boolean 3
booleans, Boole's 180
bound function 206
- greatest lower 204
- least upper 204
- time 47, 61
- unbounded 178
- variable 204, 208

boundary variable 126, 131
bracket algebra 153
brackets 188
break 71
broadcast 141
Brock-Ackermann 199
buffer 122
bunch 14, 202
- elementary 14
- empty 15

busy-wait loop 76
call-by-value-result 179
Cantor's diagonal 181
- paradise 155

cardinality 14
cases, refinement by 43
caskets 152
catenation 17, 156
- list 20

channel 131
- declaration 138

character 13, 15
check, input 133
- parity 171

circular list 189
- numbers 152

classical logic 202
clock 76
closure, transitive 172
code, Huffman 197
coin 180
- weights 197

combination 166
- next 169

command, guarded 179
common divisor, greatest 175
- item, smallest 175
- items 175
- multiple, least 175
- prefix, longest 176

communication 131
- synchronous 200

comparison list 166
compiler 45
complete 5, 101
completeness 51, 117
completion rule 5, 6
composite number 154
composition conditional 4
    dependent 36, 127
    disjoint 194
    function 31
    independent 118, 119, 126
    list 21
    semi-dependent 194
computing constant 36
    interactive 134
    variable 36
concrete space 207
concurrency 118
    list 120
condition 40
    final 40
    initial 40
conditional composition 4
conjunct 3
conjunction 3
consensus 197
consequent 3
consistency rule 5, 6
consistent 5, 101
constant 23
    computing 36
    logical 208
    mathematical 36
    state 36
construction 16, 91
    fixed-point 94
    recursive data 95
    recursive program 98
constructive logic 202
constructors 91
context 10
continuing 7, 9
contradiction 10
control process 134
controlled iteration 74
controller, reaction 137
convex equal pair 168
count, duplicate 174
    inversion 171
    item 174
    segment sum 170
    two-dimensional sorted 168
cube 165
    test 166
cursor, read 131
    write 131
data construction, recursive 95
    invariant 207
    refinement 207
    structure 14
    structures 100
    transformation 109
    transformation, interactive 196
    transformer 109
deadlock 124, 139
decimal-point numbers 185
declaration, channel 138
    variable 66
dependent composition 35, 127
detachment 6
deterministic 89
    function 29
    specification 35
diagonal 170
    Cantor's 181
dice 86, 180
difference, minimum 171
digit sum 171
digitizer 137
diminished *J*-list 175
dining philosophers 124, 195
disjoint composition 194
disjunct 3
disjunction 3
distribute 15
distribution, probability 82
    one-point 83
division, machine 174
    natural 169
divisor, greatest common 175
domain 23
drunk 181
dual 148
duplicate count 174
earliest meeting time 166
    quitter 171
edit distance 174

element 14
    assignment, array 68
elementary bunch 14
empty bunch 15
    set 17
    string 17
entropy 87
equation 4
evaluation logic 202
    rule 5, 6
exact postcondition 40
    precondition 40
    precondition for termination 166
exclusion, mutual 200
execution, sequential 36
    time 60
existence 204
existential quantification 26
exit 71
exponentiation, binary 45, 167
    fast 57, 167
expression 13
extended integers 15
    naturals 15
    rationals 15
    reals 15
factor 155
    count 169
factorial 164
family theory 154
fast exponentiation 57. 167
Fermat's last program 170
Fibolucci 173
Fibonacci 59, 173, 183
file update 200
final condition 40
state 34
fixed-point 94, 168
    construction 94
    induction 94
    least 94
    theorem 182
flatten 170
follows from 3
formal 12
    proof 7
format, proof 7
frame 67
    problem 178
free 204
friends 158
function 23, 79, 80
    bound 206
    composition 31
    deterministic 29
    higher-order 30
    inclusion 30
    nondeterministic 29
    partial 29
    refinement 89
    retrieve 207
    total 29
functional programming 88, 90
fuzzybunch 154
gas burner 128, 136, 196
general recursion 76
generation 207
generator, random number 84
generic 13
ghost variables 208
gluing relation 207
go to 45, 71, 76
Gödel/Turing incompleteness 159
grammar 94
greatest common divisor 175
    lower bound 204
    square under a histogram 177
    subsequence 171
grow slow 196
guarded command 179
heads and tails 171
heap 189
hidden variable 204
higher-order function 30
Huffman code 197
hyperbunch 154
idempotent permutation 169
imperative programming 88, 90
implementable 34, 35, 89, 132, 127
implementation, input 198
implemented specification 41
implementer's variables 106
implication 3
inclusion 14
    function 30
incomplete 5
incompleteness, Gödel/Turing 159
    transformation 193
inconsistent 5
independent composition 118, 119, 126

index 18
list 20
induction 16, 91
fixed-point 94
proof by 93
infinity 12
infix 3
information 87
initial condition 40
state 34
initializing assignment 67
input 133
check 133
implementation 198
insertion list 190
sort 123
instance rule 5
instantiation 4
integer numbers 15
integers, extended 15
interactive computing 134
data transformation 196
variable 126, 131
intersection 14
interval union 171
invariant 75, 77, 206
data 207
linking 207
inverse permutation 169
inversion count 171
item 17
count 174
maximum 120
smallest common 175
items, common 175
unique 175
iteration, controlled 74
J-list 175
knights and knaves 151
Knuth, Morris, Pratt 177
largest true square 175
law 7
substitution 38
least common multiple 175
fixed-point 94
upper bound 204
left side 4
length list 20
string 17
text 168
lexicographic order 18
limit 33
limited queue 115, 192
linear algebra 189
search 51, 167
linking invariant 207
list 14, 20
bitonic 158
catenation 20
circular 189
comparison 166
composition 21
concurrency 120
diminished *J*- 175
index 20
insertion 190
J- 175
length 20
next sorted 169
P- 175
summation 43, 67, 88, 166
local 25
minimum 169
logarithm natural binary 169
logic 3
classical 202
constructive 202
evaluation 202
logical constants 208
long texts 177
longest balanced segment 170
common prefix 176
palindrome 170
plateau 170
smooth segment170
sorted sublist 174
loop 48, 69
busy-wait 76
lower bound, greatest 204
machine division 174
multiplication 174
squaring 174
maid and butler 151
majority vote 179
mathematical constant 36
variable 36
matrix multiplication 197

maximum item 120, 166
product segment 170
space 63
McCarthy's 91 problem 172
memory variables 46
merge 135, 174
time 199
message script 131
metalanguage 202
minimum difference 171
local 169
sum segment 170
missing number 168
model-checking 1
modification, program 57
modus ponens 6
monitor 136, 138
monotonic 9
Mr.Bean's socks 181
multibunch 154
multidimensional 22
multiple, least common 175
multiplication, machine 174
matrix 197
table 167
museum 176
mutual exclusion 200
natural binary logarithm 169
division 169
numbers 15
square root 169
naturals, extended 15
necessary postcondition 40
precondition 40
negation 3
next combination 169
permutation 169
sorted list 169
nondeterministic 89
assignment 177
function 29
specification 35
nonlocal 25
notation 201
number 12
composite 154
generator, random 84
missing 168
numbers, circular 152
decimal-point 185
Fibonacci 59
integer 15
natural 15
rational 15
real 15
von Neumann 155
one-point law 28
distribution 83
operand 3
operator 3
order lexicographic 18
prefix 156
ordered pair search 168
output 133
P-list 175
package 14
pair search, ordered 168
palindrome, longest 170
parallelism 118
parameter 24, 79, 80
reference 80, 81
parity check 171
parking 151
parsing 113, 190
partial function 29
partition 118
partitions 175
parts, refinement by 43
party 190
Pascal's triangle 167
path, shortest 172
pattern search 168
perfect shuffle 198
periodic sequence, ultimately 175
permutation, idempotent 169
inverse 169
next 169
pigeon-hole 159
pivot 171
pointer 22, 81, 105
polynomial 166
postcondition 40, 77
exact 40
necessary 40
sufficient 40
postspecification, weakest 163
poststate 34
power series 141, 199

powerset 17
precedence 4, 5
precondition 40, 77, 204
exact 40
necessary 40
sufficient 40
weakest 204
predecessor 13
predicate 24
prefix 3
longest common 176
order 156
prespecification, weakest 163
prestate 34
private variable 204
probability 82
distribution 82
uniform 84
problem, frame 178
process 118
control 134
processing, batch 134
program 41
construction, recursive 98
modification 57
programming, functional 88, 90
programming, imperative 88, 90
proof 7
by induction 93
formal 7
format 7
rule 5
prophesy variable 208
proposition 201
public variable 204
quantification, existential 26
universal 26
quantifier 26
queue 103, 108, 188
limited 115, 192
quitter, earliest 171
random number generator 84
range 23
rational numbers 15
rationals, extended 15
reachability 172
reaction controller 137
read cursor 131
real 33
numbers 15
time 46
reals, extended 15
record 69
recursion 42
general 76
tail 76
recursive data construction 95
program construction 98
time 48
reference parameter 80, 81
refinement 39
by cases 43
by parts 43
by steps 43
data 207
function 89
stepwise 43
reformat 197
reification 204
relation 24
abstraction 207
gluing 207
transitive 161
remainder 169
renaming 24
repetition 199
resettable variable 189
retrieve function 207
reverse 169
right side 4
roll up 161
roller coaster 60, 173
root, natural square 169
rotation, smallest 176
test 176
rule, completion 5, 6
consistency 5, 6
evaluation 5, 6
instance 5
proof 5
rulers 182
running total 165, 195
Russell's barber 159
paradox 159
satisfiable 35, 89
scale 152
schema, axiom 202
scope 23, 66

script, message 131
    time 131
search, approximate 171
    binary 167, 53
    linear 167, 51
    ordered pair 168
    pattern 168
    sorted two-dimensional 168
    ternary 167
    two-dimensional 167
    two-dimensional 72
security switch 111, 191
segment 21
    almost sorted 174
    longest balanced 170
    longest smooth 170
    maximum product 170
    minimum sum 170
    sum count 170
selective union 24
self-describing 21
self-reproducing 21
semi-dependent composition 194
sentence 201
sentinel 52, 113, 200
sequence, ultimately periodic 175
sequential execution 36
series, power 141, 199
set 14, 17
    empty 17
shared variable 131, 136
shortest path 172
shuffle, perfect 198
side-effect 78
sieve 195
signal 133
size 14
slip 188
smallest common item 175
    rotation 176
socks, Mr.Bean's 181
solution 28
sort, insertion 123
    test 167
sorted list, next 169
    segment, almost 174
    sublist, longest 174
    two-dimensional count 168
    two-dimensional search 168
soundness 51, 117
space 61, 129
    abstract 207
    average 64
    concrete 207
    maximum 63
    state 34
sparse array 193
specification 34
    deterministic 35
    implemented 41
    nondeterministic 35
    transitive 161
    variable 208
square 164
    greatest under a histogram 177
    largest true 175
    root, natural 169
squaring, machine 174
stack 100, 106, 187, 188
state 34
    constant 36
    final 34
    initial 34
    space 34
    variable 34, 36
steps, refinement by 43
stepwise refinement 43
string 14, 17, 184
    empty 17
    length 17
stronger 3, 9
structure 69
    data 14, 100
sublist 21
    longest sorted 174
subscript 18
substitution 4, 25
    law 38
successor 13, 23
sufficient postcondition 40
    precondition 40
sum, alternating 166
    bit 171
    digit 171
summation, list 43, 67, 88, 166
suspension, variable 67
swapping partners 158
switch, security 111, 191
synchronizer 137
synchronous communication 200

T-string 197
tail recursion 76
take a number 192
telephone 196
tennis 151
termination 34, 50
    exact precondition for 166
term 201
ternary search 167
testing 145
text 19
    length 168
    long 177
theorem 3
thermostat 128, 136, 196
Thue string 197
time 46
    bound 47, 61
    execution 60
    merge 199
    real 46
    recursive 48
    script 131
    transit 134
    variable 46
timeout 198
total function 29
Towers of Hanoi 61, 172
transformation, data 109
    incompleteness 193
    interactive data 196
transformer, data 109
transit time 134
transitive closure 172
    relation 161
    specification 161
tree 104, 108, 189, 190
    binary 192
truth table 3, 4
two-dimensional search 72, 167
    search, sorted 168
ultimately periodic sequence 175
unbounded bound 178
undefined value 66
unequation 4
unexpected egg 152
unicorn 159
uniform probability 84
union 14
    interval 171
    selective 24
unique items 175
universal quantification 26
unsatisfiable 35, 89
update, file 200
upper bound, least 204
user's variables 106
value, undefined 66
variable 4, 23
    abstract 208
    bound 204, 208
    boundary 126, 131
    computing 36
    declaration 66
    ghost 208
    hidden 204
    implementer's 106
    interactive 126, 131
    mathematical 36
    memory 46
    private 204
    prophesy 208
    public 204
    resettable 189
    shared 131, 136
    specification 208
    state 34, 36
    suspension 67
    time 46
    user's 106
    visible 204
variant 206
visible variable 204
von Neumann numbers 155
vote, majority 179
wait 76
weaker 3, 9
weakest postspecification 163
    precondition 204
    prespecification 163
whodunit 157
wholebunch 154
widget 187
write cursor 131
z-free subtext 174
Zeno 165

End of Index

11.4 Laws

11.4.0 Booleans

Let a , b , c , d , and e be boolean.

Boolean Laws

$\top$

$\neg\bot$

Law of Excluded Middle (Tertium non Datur)

$a \lor \neg a$

Law of Noncontradiction

$\neg(a \land \neg a)$

Base Laws

$\neg(a \land \bot)$

$a \lor \top$

$a \Rightarrow \top$

$\bot \Rightarrow a$

Identity Laws

$\top \land a \;=\; a$

$\bot \lor a \;=\; a$

$\top \Rightarrow a \;=\; a$

$\top = a \;=\; a$

Idempotent Laws

$a \land a \;=\; a$

$a \lor a \;=\; a$

Reflexive Laws

$a \Rightarrow a$

$a = a$

Laws of Indirect Proof

$\neg a \Rightarrow \bot \;=\; a$ (Reductio ad Absurdum)

$\neg a \Rightarrow a \;=\; a$

Law of Specialization

$a \land b \Rightarrow a$

Associative Laws

$a \land (b \land c) \;=\; (a \land b) \land c$

$a \lor (b \lor c) \;=\; (a \lor b) \lor c$

$a = (b = c) \;=\; (a = b) = c$

$a \ne (b \ne c) \;=\; (a \ne b) \ne c$

$a = (b \ne c) \;=\; (a = b) \ne c$

Law of Double Negation

$\neg\neg a \;=\; a$

Duality Laws (deMorgan)

$\neg(a \land b) \;=\; \neg a \lor \neg b$

$\neg(a \lor b) \;=\; \neg a \land \neg b$

Laws of Exclusion

$a \Rightarrow \neg b \;=\; b \Rightarrow \neg a$

$a = \neg b \;=\; a \ne b \;=\; \neg a = b$

Laws of Inclusion

$a \Rightarrow b \;=\; \neg a \lor b$ (Material Implication)

$a \Rightarrow b \;=\; (a \land b \;=\; a)$

$a \Rightarrow b \;=\; (a \lor b \;=\; b)$

Absorption Laws

$a \land (a \lor b) \;=\; a$

$a \lor (a \land b) \;=\; a$

Laws of Direct Proof

$(a \Rightarrow b) \land a \Rightarrow b$ (Modus Ponens)

$(a \Rightarrow b) \land \neg b \Rightarrow \neg a$ (Modus Tollens)

$(a \lor b) \land \neg a \Rightarrow b$ (Disjunctive Syllogism)

Transitive Laws

$(a \land b) \land (b \land c) \Rightarrow (a \land c)$

$(a \Rightarrow b) \land (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$

$(a = b) \land (b = c) \Rightarrow (a = c)$

$(a \Rightarrow b) \land (b = c) \Rightarrow (a \Rightarrow c)$

$(a = b) \land (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$

Distributive Laws (Factoring)

$a \land (b \land c) \;=\; (a \land b) \land (a \land c)$

$a \land (b \lor c) \;=\; (a \land b) \lor (a \land c)$

$a \lor (b \land c) \;=\; (a \lor b) \land (a \lor c)$

$a \lor (b \lor c) \;=\; (a \lor b) \lor (a \lor c)$

$a \lor (b \Rightarrow c) \;=\; (a \lor b) \Rightarrow (a \lor c)$

$a \lor (b = c) \;=\; (a \lor b) = (a \lor c)$

$a \Rightarrow (b \land c) \;=\; (a \Rightarrow b) \land (a \Rightarrow c)$

$a \Rightarrow (b \lor c) \;=\; (a \Rightarrow b) \lor (a \Rightarrow c)$

$a \Rightarrow (b \Rightarrow c) \;=\; (a \Rightarrow b) \Rightarrow (a \Rightarrow c)$

$a \Rightarrow (b = c) \;=\; (a \Rightarrow b) = (a \Rightarrow c)$

Symmetry Laws (Commutative Laws)

$a \wedge b \;=\; b \wedge a$
$a \vee b \;=\; b \vee a$
$a = b \;=\; b = a$
$a \neq b \;=\; b \neq a$

Antisymmetry Law (Double Implication)

$(a \Rightarrow b) \wedge (b \Rightarrow a) \;=\; a = b$

Laws of Discharge

$a \wedge (a \Rightarrow b) \;=\; a \wedge b$
$a \Rightarrow (a \wedge b) \;=\; a \Rightarrow b$

Antimonotonic Law

$a \Rightarrow b \;\Longrightarrow\; (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$

Monotonic Laws

$a \Rightarrow b \;\Longrightarrow\; c \wedge a \Rightarrow c \wedge b$
$a \Rightarrow b \;\Longrightarrow\; c \vee a \Rightarrow c \vee b$
$a \Rightarrow b \;\Longrightarrow\; (c \Rightarrow a) \Rightarrow (c \Rightarrow b)$

Law of Generalization

$a \Rightarrow a \vee b$

Antidistributive Laws

$a \wedge b \Rightarrow c \;=\; (a \Rightarrow c) \vee (b \Rightarrow c)$
$a \vee b \Rightarrow c \;=\; (a \Rightarrow c) \wedge (b \Rightarrow c)$

Laws of Portation

$a \wedge b \Rightarrow c \;=\; a \Rightarrow (b \Rightarrow c)$
$a \wedge b \Rightarrow c \;=\; a \Rightarrow \neg b \vee c$

Laws of Conflation

$(a \Rightarrow b) \wedge (c \Rightarrow d) \;\Longrightarrow\; a \wedge c \Rightarrow b \wedge d$
$(a \Rightarrow b) \wedge (c \Rightarrow d) \;\Longrightarrow\; a \vee c \Rightarrow b \vee d$

Contrapositive Law

$a \Rightarrow b \;=\; \neg b \Rightarrow \neg a$

Laws of Equality and Difference

$a{=}b \;=\; (a \wedge b) \vee (\neg a \wedge \neg b)$
$a{\neq}b \;=\; (a \wedge \neg b) \vee (\neg a \wedge b)$

Law of Resolution

$a \wedge c \;\Longrightarrow\; (a \vee b) \wedge (\neg b \vee c) \;=\; (a \wedge \neg b) \vee (b \wedge c) \;\Longrightarrow\; a \vee c$

Case Base Laws

if $\top$ **then** a **else** $b \;=\; a$
if $\bot$ **then** a **else** $b \;=\; b$

Case Reversal Law

if a **then** b **else** $c \;=\;$ **if** $\neg a$ **then** c **else** b

Case Idempotent Law

if a **then** b **else** $b \;=\; b$

Case Absorption Laws

if a **then** b **else** $c \;=\;$ **if** a **then** $a{\wedge}b$ **else** c
if a **then** b **else** $c \;=\;$ **if** a **then** $a{\Rightarrow}b$ **else** c
if a **then** b **else** $c \;=\;$ **if** a **then** $a{=}b$ **else** c
if a **then** b **else** $c \;=\;$ **if** a **then** b **else** $\neg a \wedge c$
if a **then** b **else** $c \;=\;$ **if** a **then** b **else** $a{\vee}c$
if a **then** b **else** $c \;=\;$ **if** a **then** b **else** $a{\neq}c$

Case Analysis Laws

if a **then** b **else** $c \;=\; (a \wedge b) \vee (\neg a \wedge c)$
if a **then** b **else** $c \;=\; (a{\Rightarrow}b) \wedge (\neg a \Rightarrow c)$

One Case Laws

if a **then** $\top$ **else** $b \;=\; a{\vee}b$
if a **then** $\bot$ **else** $b \;=\; \neg a \wedge b$
if a **then** b **else** $\top \;=\; a{\Rightarrow}b$
if a **then** b **else** $\bot \;=\; a{\wedge}b$

Case Creation Laws

$a \;=\;$ **if** b **then** $b{\Rightarrow}a$ **else** $\neg b \Rightarrow a$
$a \;=\;$ **if** b **then** $b{\wedge}a$ **else** $\neg b \wedge a$
$a \;=\;$ **if** b **then** $b{\wedge}a$ **else** $\neg b \wedge a$
$a \;=\;$ **if** b **then** $b{=}a$ **else** $b{\neq}a$

Case Distributive Laws (Case Factoring)

$\neg$ **if** a **then** b **else** $c \;=\;$ **if** a **then** $\neg b$ **else** $\neg c$
(**if** a **then** b **else** c) $\wedge\, d \;=\;$ **if** a **then** $b{\wedge}d$ **else** $c{\wedge}d$
and similarly replacing $\wedge$ by any of $\vee \;=\; \neq \;\Rightarrow\; \Leftarrow$
if a **then** $b \wedge c$ **else** $d \wedge e \;=\;$ (**if** a **then** b **else** d) $\wedge$ (**if** a **then** c **else** e)
and similarly replacing $\wedge$ by any of $\vee \;=\; \neq \;\Rightarrow\; \Leftarrow$

End of Booleans

11.4.1 Generic

The operators $= \ne$ **if then else** apply to every type of expression, with the laws

$x = x$	reflexivity
$x{=}y \;\;=\;\; y{=}x$	symmetry
$x{=}y \wedge y{=}z \;\Rightarrow\; x{=}z$	transitivity
$x{=}y \;\Rightarrow\; f\,x = f\,y$	transparency
$x{\ne}y \;\;=\;\; \neg(x{=}y)$	unequality
$\textbf{if } \top \textbf{ then } x \textbf{ else } y \;\;=\;\; x$	case base
$\textbf{if } \bot \textbf{ then } x \textbf{ else } y \;\;=\;\; y$	case base

The operators $< \le > \ge$ apply to numbers, characters, strings, and lists, with the laws

$\neg\, x{<}x$	irreflexivity
$\neg(x{<}y \wedge x{>}y)$	exclusivity
$\neg(x{<}y \wedge x{=}y)$	exclusivity
$x{\le}y \wedge y{\le}x \;=\; x{=}y$	antisymmetry
$x{<}y \wedge y{<}z \Rightarrow x{<}z$	transitivity
$x{\le}y \;\;=\;\; x{<}y \vee x{=}y$	inclusivity
$x{>}y \;\;=\;\; y{<}x$	mirror
$x{\ge}y \;\;=\;\; y{\le}x$	mirror
$x{<}y \vee x{=}y \vee x{>}y$	totality, trichotomy

End of Generic

11.4.2 Numbers

Let d be a sequence of (zero or more) digits, and let x, y, and z be numbers.

$d0{+}1 = d1$	counting
$d1{+}1 = d2$	counting
$d2{+}1 = d3$	counting
$d3{+}1 = d4$	counting
$d4{+}1 = d5$	counting
$d5{+}1 = d6$	counting
$d6{+}1 = d7$	counting
$d7{+}1 = d8$	counting
$d8{+}1 = d9$	counting
$d9{+}1 = (d{+}1)0$	counting (see Exercise 22)
$x{+}0 = x$	identity
$x{+}y = y{+}x$	symmetry
$x{+}(y{+}z) \;=\; (x{+}y){+}z$	associativity
$-\infty{<}x{<}\infty \Rightarrow (x{+}y = x{+}z \;\;=\;\; y{=}z)$	cancellation
$-\infty{<}x \Rightarrow\;\; \infty{+}x = \infty$	absorption
$x{<}\infty \;\Rightarrow\; -\infty + x = -\infty$	absorption
$-x \;=\; 0 - x$	negation
$- -x = x$	self-inverse
$-(x{+}y) = -x + -y$	distributivity
$-(x{-}y) = -x - -y$	distributivity
$-(x{\times}y) = -x \times y$	semi-distributivity
$-(x/y) = -x / y$	semi-distributivity
$x{-}y = -(y{-}x)$	antisymmetry
$x{-}y \;=\; x + -y$	subtraction

$x + (y - z) \;=\; (x + y) - z$	associativity
$-\infty<x<\infty \;\Rightarrow\; (x-y = x-z \;=\; y=z)$	cancellation
$-\infty<x<\infty \;\Rightarrow\; x-x = 0$	inverse
$x<\infty \;\Rightarrow\; \infty-x = \infty$	absorption
$-\infty<x \;\Rightarrow\; -\infty - x = -\infty$	absorption
$-\infty<x<\infty \;\Rightarrow\; x\times 0 = 0$	base
$x\times 1 = x$	identity
$x\times y = y\times x$	symmetry
$x\times(y+z) = x\times y + x\times z$	distributivity
$x\times(y\times z) = (x\times y)\times z$	associativity
$-\infty<x<\infty \wedge x\ne 0 \;\Rightarrow\; (x\times y = x\times z \;=\; y=z)$	cancellation
$0<x \;\Rightarrow\; x\times\infty = \infty$	absorption
$0<x \;\Rightarrow\; x \times -\infty \;=\; -\infty$	absorption
$x/1 = x$	identity
$-\infty<x<\infty \wedge x\ne 0 \;\Rightarrow\; x/x = 1$	inverse
$x\times(y/z) = (x\times y)/z = x/(z/y)$	multiplication-division
$y\ne 0 \;\Rightarrow\; (x/y)/z = x/(y\times z)$	multiplication-division
$-\infty<x<\infty \;\Rightarrow\; x/\infty = 0 = x/-\infty$	annihilation
$-\infty<x<\infty \;\Rightarrow\; x^0 = 1$	base
$x^1 = x$	identity
$x^{y+z} = x^y \times x^z$	exponents
$x^{y\times z} = (x^y)^z$	exponents
$-\infty<0<1<\infty$	direction
$x<y \;=\; -y<-x$	reflection
$-\infty<x<\infty \;\Rightarrow\; (x+y < x+z \;=\; y<z)$	cancellation, translation
$0<x<\infty \;\Rightarrow\; (x\times y < x\times z \;=\; y<z)$	cancellation, scale
$x<y \vee x=y \vee x>y$	trichotomy
$-\infty \le x \le \infty$	extremes

End of Numbers

11.4.3 Bunches

Let x and y be elements (booleans, numbers, characters, sets, strings and lists of elements).

$x{:}\ y \;=\; x=y$	elementary law
$x{:}\ A,B \;=\; x{:}\ A \;\vee\; x{:}\ B$	compound law
$A,A = A$	idempotence
$A,B = B,A$	symmetry
$A,(B,C) = (A,B),C$	associativity
$A\text{‘}A = A$	idempotence
$A\text{‘}B = B\text{‘}A$	symmetry
$A\text{‘}(B\text{‘}C) = (A\text{‘}B)\text{‘}C$	associativity
$A,B{:}\ C \;=\; A{:}\ C \;\wedge\; B{:}\ C$	antidistributivity
$A{:}\ B\text{‘}C \;=\; A{:}\ B \;\wedge\; A{:}\ C$	distributivity
$A{:}\ A,B$	generalization
$A\text{‘}B{:}\ A$	specialization
$A{:}\ A$	reflexivity
$A{:}\ B \;\wedge\; B{:}\ A \;=\; A=B$	antisymmetry
$A{:}\ B \;\wedge\; B{:}\ C \;\Rightarrow\; A{:}\ C$	transitivity
$\text{¢}\ null = 0$	size
$\text{¢}x = 1$	size

$¢(A, B) + ¢(A‘B) = ¢A + ¢B$ size
$\neg\ x: A \;\Rightarrow\; ¢(A‘x) = 0$ size
$A: B \;\Rightarrow\; ¢A \le ¢B$ size
$A,(A‘B) \;=\; A$ absorption
$A‘(A,B) \;=\; A$ absorption
$A: B \;\;=\;\; A,B = B \;\;=\;\; A = A‘B$ inclusion
$A,(B,C) \;=\; (A,B),(A,C)$ distributivity
$A,(B‘C) \;=\; (A,B)‘(A,C)$ distributivity
$A‘(B,C) \;=\; (A‘B), (A‘C)$ distributivity
$A‘(B‘C) \;=\; (A‘B)‘(A‘C)$ distributivity
$A: B \;\wedge\; C: D \;\Rightarrow\; A,C: B,D$ conflation, monotonicity
$A: B \;\wedge\; C: D \;\Rightarrow\; A‘C: B‘D$ conflation, monotonicity
$null: A$ induction
$A, null = A$ identity
$A \text{ ‘ } null = null$ base
$¢A = 0 \;\;=\;\; A = null$ size
$x: int \;\wedge\; y: xint \;\wedge\; x{\le}y \;\Rightarrow\; (i: x,..y \;\;=\;\; i: int \;\wedge\; x{\le}i{<}y)$
$x: int \;\wedge\; y: xint \;\wedge\; x{\le}y \;\Rightarrow\; ¢(x,..y) \;=\; y{-}x$
$-null \;=\; null$ distribution
$-(A, B) \;=\; -A, -B$ distribution
$A{+}null \;=\; null{+}A \;=\; null$ distribution
$(A, B){+}(C, D) \;=\; A{+}C, A{+}D, B{+}C, B{+}D$ distribution
and similarly for many other operators (see the final page of the book)

End of Bunches

11.4.4 Sets

$\{\sim S\} \;=\; S$
$\sim\{A\} \;=\; A$
$\{A\} \;\ne\; A$
$A \in \{B\} \;\;=\;\; A: B$
$\{A\} \subseteq \{B\} \;\;=\;\; A: B$
$\{A\}: ⁄B \;\;=\;\; A: B$
$\$\{A\} = ¢A$
$\{A\} \cup \{B\} \;=\; \{A, B\}$
$\{A\} \cap \{B\} \;=\; \{A \text{ ‘ } B\}$
$\{A\} = \{B\} \;\;=\;\; A = B$
$\{A\} \ne \{B\} \;\;=\;\; A \ne B$

End of Sets

11.4.5 Strings

Let S, T, and U be strings; let i and j be items (booleans, numbers, characters, bunch of items, sets, lists, functions); let n be extended natural; let x, y, and z be integers.

$nil; S \;=\; S; nil \;=\; S$
$S; (T; U) \;=\; (S; T); U$
$\leftrightarrow nil \;=\; 0$
$\leftrightarrow i \;=\; 1$
$\leftrightarrow(S; T) \;=\; \leftrightarrow S + \leftrightarrow T$
$S_{nil} \;=\; nil$
$\leftrightarrow S{<}\infty \;\Rightarrow\; (S; i; T)_{\leftrightarrow S} \;=\; i$
$S_{T;\, U} \;=\; S_T; S_U$
$S_{(T_U)} \;=\; (S_T)_U$
$S_{\{A\}} \;=\; \{S_A\}$
$\leftrightarrow S{<}\infty \;\Rightarrow\; nil \;\le\; S \;<\; S; i; T$
$\leftrightarrow S{<}\infty \;\Rightarrow\; (i{<}j \;\;=\;\; S; i; T \;<\; S; j; U)$
$\leftrightarrow S{<}\infty \;\Rightarrow\; (i{=}j \;\;=\;\; S; i; T \;=\; S; j; T)$
$0{*}S \;=\; nil$
$(n{+}1){*}S \;=\; n{*}S; S$
$\leftrightarrow S{<}\infty \;\Rightarrow\; S;i;T \triangleleft \leftrightarrow S \triangleright j \;=\; S;j;T$
$x;..x \;=\; nil$
$x;..x{+}1 \;=\; x$
$(x;..y) \;;\; (y;..z) \;=\; x;..z$
$\leftrightarrow(x;..y) \;=\; y{-}x$

End of Strings

11.4.6 Lists

Let S and T be strings; let i and j be items (booleans, numbers, characters, bunch of items, sets, lists, functions); let L, M, and N be lists.

$[S] \ne S$
$\sim[S] = S$
$[\sim L] = L$
$[S]\ T = S_T$
$[S]+[T] = [S;\ T]$
$[S]=[T] \;\;=\;\; S=T$
$[S]<[T] \;\;=\;\; S<T$
$nil \rightarrow i \mid L = i$
$n \rightarrow i \mid [S] = [S \triangleleft n \triangleright i]$
$(S;T) \rightarrow i \mid L = S \rightarrow (T \rightarrow i \mid L@S) \mid L$
$\#[S] = \leftrightarrow S$
$S_{[T]} = [S_T]$
$[S]\ [T] = [S_T]$
$L\ \{A\} = \{L\ A\}$
$L\ [S] = [L\ S]$
$(L\ M)\ N = L\ (M\ N)$
$L@nil = L$
$L@i = L\ i$
$L@(S;\ T) = L@S@T$

End of Lists

11.4.7 Functions

Renaming Law — if v and w do not appear in D and w does not appear in b
$\langle v{:}\ D \rightarrow b\rangle = \langle w{:}\ D \rightarrow \langle v{:}\ D \rightarrow b\rangle w\rangle$

Application Law: if element $x{:}\ D$
$\langle v{:}\ D \rightarrow b\rangle x =$ (substitute x for v in b)

Law of Extension
$f = \langle v{:}\ \Delta f \rightarrow f\ v\rangle$

Domain Law
$\Delta\ \langle v{:}\ D \rightarrow b\rangle = D$

Function Composition Laws: If $\neg\ f{:}\ \Delta g$
$\Delta(g\ f) = \S x{:}\ \Delta f \cdot fx{:}\ \Delta g$
$(g\ f)\ x = g\ (f\ x)$
$f\ (g\ h) = (f\ g)\ h$

Function Inclusion Law
$f{:}\ g \;\;=\;\; \Delta g{:}\ \Delta f \wedge \forall x{:}\ \Delta g \cdot fx{:}\ gx$

Cardinality Law
$\text{¢}A = \Sigma\ (A \rightarrow 1)$

Function Equality Law
$f=g \;\;=\;\; \Delta f = \Delta g \wedge \forall x{:}\ \Delta f \cdot fx = gx$

Laws of Functional Intersection
$\Delta(f\ ‘\ g) = \Delta f, \Delta g$
$(f\ ‘\ g)\ x = (f \mid g)\ x\ ‘\ (g \mid f)\ x$

Laws of Functional Union
$\Delta(f, g) = \Delta f\ ‘\ \Delta g$
$(f, g)\ x = f\ x, g\ x$

Laws of Selective Union
$\Delta(f \mid g) = \Delta f, \Delta g$
$(f \mid g)\ x = \textbf{if}\ x{:}\ \Delta f\ \textbf{then}\ f\ x\ \textbf{else}\ g\ x$
$f \mid (g \mid h) = (f \mid g) \mid h$

Laws of Selective Union
$f \mid f = f$
$(g \mid h) f = g\ f \mid h\ f$
$\langle v{:}\ A \rightarrow x\rangle \mid \langle v{:}\ B \rightarrow y\rangle = \langle v{:}\ A, B \rightarrow \textbf{if}\ v{:}\ A\ \textbf{then}\ x\ \textbf{else}\ y\rangle$

Distributive Laws
$f\ null = null$
$f\ (A, B) = f\ A, f\ B$
$f\ (\S g) = \S y{:}\ f\ (\Delta g) \cdot \exists x{:}\ \Delta g \cdot fx{=}y \wedge gx$
$f\ (\textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y) = \textbf{if}\ b\ \textbf{then}\ f\ x\ \textbf{else}\ f\ y$
$(\textbf{if}\ b\ \textbf{then}\ f\ \textbf{else}\ g)\ x = \textbf{if}\ b\ \textbf{then}\ f\ x\ \textbf{else}\ g\ x$

Arrow Laws
$f{:}\ null \rightarrow A$
$A \rightarrow B{:}\ (A‘C) \rightarrow (B,D)$
$f{:}\ A \rightarrow B \;\;=\;\; A{:}\ \Delta f \wedge \forall a{:}\ A \cdot fa{:}\ B$

End of Functions

11.4.8 Quantifiers

Let x be an element, let a , b and c be boolean, let n and m be numeric, let f and g be functions, and let P be a predicate.

$$\forall v: null\cdot b \;=\; \top$$
$$\forall v: x\cdot b \;=\; \langle v: x\to b\rangle\, x$$
$$\forall v: A,B\cdot b \;=\; (\forall v: A\cdot b) \wedge (\forall v: B\cdot b)$$
$$\forall v: (\S v: D\cdot b)\cdot c \;=\; \forall v: D\cdot b \Rightarrow c$$

$$\exists v: null\cdot b \;=\; \bot$$
$$\exists v: x\cdot b \;=\; \langle v: x\to b\rangle\, x$$
$$\exists v: A,B\cdot b \;=\; (\exists v: A\cdot b) \vee (\exists v: B\cdot b)$$
$$\exists v: (\S v: D\cdot b)\cdot c \;=\; \exists v: D\cdot b \wedge c$$

$$\Sigma v: null\cdot n \;=\; 0$$
$$\Sigma v: x\cdot n \;=\; \langle v: x\to n\rangle\, x$$
$$(\Sigma v: A,B\cdot n) + (\Sigma v: A`B\cdot n) \;=\; (\Sigma v: A\cdot n) + (\Sigma v: B\cdot n)$$
$$\Sigma v: (\S v: D\cdot b)\cdot n \;=\; \Sigma v: D\cdot \textbf{if } b \textbf{ then } n \textbf{ else } 0$$

$$\Pi v: null\cdot n \;=\; 1$$
$$\Pi v: x\cdot n \;=\; \langle v: x\to n\rangle\, x$$
$$(\Pi v: A,B\cdot n) \times (\Pi v: A`B\cdot n) \;=\; (\Pi v: A\cdot n) \times (\Pi v: B\cdot n)$$
$$\Pi v: (\S v: D\cdot b)\cdot n \;=\; \Pi v: D\cdot \textbf{if } b \textbf{ then } n \textbf{ else } 1$$

$$MIN\ v: null\cdot n \;=\; \infty$$
$$MIN\ v: x\cdot n \;=\; \langle v: x\to n\rangle\, x$$
$$MIN\ v: A,B\cdot n \;=\; min\ (MIN\ v: A\cdot n)\ (MIN\ v: B\cdot n)$$
$$MIN\ v: (\S v: D\cdot b)\cdot n \;=\; MIN\ v: D\cdot \textbf{if } b \textbf{ then } n \textbf{ else } \infty$$

$$MAX\ v: null\cdot n \;=\; -\infty$$
$$MAX\ v: x\cdot n \;=\; \langle v: x\to n\rangle\, x$$
$$MAX\ v: A,B\cdot n \;=\; max\ (MAX\ v: A\cdot n)\ (MAX\ v: B\cdot n)$$
$$MAX\ v: (\S v: D\cdot b)\cdot n \;=\; MAX\ v: D\cdot \textbf{if } b \textbf{ then } n \textbf{ else } -\infty$$

$$\S v: null\cdot b \;=\; null$$
$$\S v: x\cdot b \;=\; \textbf{if } \langle v: x\to b\rangle\, x \textbf{ then } x \textbf{ else } null$$
$$\S v: A,B\cdot b \;=\; (\S v: A\cdot b), (\S v: B\cdot b)$$
$$\S v: A`B\cdot b \;=\; (\S v: A\cdot b)\ `\ (\S v: B\cdot b)$$
$$\S v: (\S v: D\cdot b)\cdot c \;=\; \S v: D\cdot b \wedge c$$

Change of Variable Laws — if d does not appear in b

$$\forall r: fD\cdot b \;=\; \forall d: D\cdot \langle r: fD\to b\rangle\, (fd)$$
$$\exists r: fD\cdot b \;=\; \exists d: D\cdot \langle r: fD\to b\rangle\, (fd)$$
$$MIN\ r: fD\cdot n \;=\; MIN\ d: D\cdot \langle r: fD\to n\rangle\, (fd)$$
$$MAX\ r: fD\cdot n \;=\; MAX\ d: D\cdot \langle r: fD\to n\rangle\, (fd)$$

Identity Laws

$$\forall v\cdot \top$$
$$\neg \exists v\cdot \bot$$

Bunch-Element Conversion Laws

$$V: W \;=\; \forall v: V\cdot \exists w: W\cdot v=w$$
$$fV: gW \;=\; \forall v: V\cdot \exists w: W\cdot fv=gw$$

Idempotent Laws — if $D \neq null$ and v does not appear in b

$$\forall v: D\cdot b \;=\; b$$
$$\exists v: D\cdot b \;=\; b$$

Absorption Laws — if $x: D$

$$\langle v: D\to b\rangle\, x \wedge \exists v: D\cdot b \;=\; \langle v: D\to b\rangle\, x$$
$$\langle v: D\to b\rangle\, x \vee \forall v: D\cdot b \;=\; \langle v: D\to b\rangle\, x$$
$$\langle v: D\to b\rangle\, x \wedge \forall v: D\cdot b \;=\; \forall v: D\cdot b$$
$$\langle v: D\to b\rangle\, x \vee \exists v: D\cdot b \;=\; \exists v: D\cdot b$$

Distributive Laws — if $D \neq null$ and v does not appear in a

$$a \wedge \forall v: D\cdot b \;=\; \forall v: D\cdot a \wedge b$$
$$a \wedge \exists v: D\cdot b \;=\; \exists v: D\cdot a \wedge b$$
$$a \vee \forall v: D\cdot b \;=\; \forall v: D\cdot a \vee b$$
$$a \vee \exists v: D\cdot b \;=\; \exists v: D\cdot a \vee b$$
$$a \Rightarrow \forall v: D\cdot b \;=\; \forall v: D\cdot a \Rightarrow b$$
$$a \Rightarrow \exists v: D\cdot b \;=\; \exists v: D\cdot a \Rightarrow b$$

Antidistributive Laws — if $D \neq null$ and v does not appear in a

$$a \Leftarrow \exists v: D\cdot b \;=\; \forall v: D\cdot a \Leftarrow b$$
$$a \Leftarrow \forall v: D\cdot b \;=\; \exists v: D\cdot a \Leftarrow b$$

Specialization Law — if $x: D$

$\forall v: D\cdot b \;\Rightarrow\; \langle v: D\rightarrow b\rangle\, x$

Generalization Law — if $x: D$

$\langle v: D\rightarrow b\rangle\, x \;\Rightarrow\; \exists v: D\cdot b$

One-Point Laws — if $x: D$

and v does not appear in x

$\forall v: D\cdot v{=}x \Rightarrow b \;=\; \langle v: D\rightarrow b\rangle\, x$

$\exists v: D\cdot v{=}x \wedge b \;=\; \langle v: D\rightarrow b\rangle\, x$

Duality Laws

$\neg \forall v\cdot b \;=\; \exists v\cdot \neg b$ (deMorgan)

$\neg \exists v\cdot b \;=\; \forall v\cdot \neg b$ (deMorgan)

$-\,MAX\ v\cdot n \;=\; MIN\ v\cdot -n$

$-\,MIN\ v\cdot n \;=\; MAX\ v\cdot -n$

Solution Laws

$\S v: D\cdot \top \;=\; D$

$(\S v: D\cdot b):\ D$

$\S v: D\cdot \bot \;=\; null$

$(\S v\cdot b): (\S v\cdot c) \;=\; \forall v\cdot b{\Rightarrow}c$

$(\S v\cdot b), (\S v\cdot c) \;=\; \S v\cdot b \vee c$

$(\S v\cdot b)\text{ ‘ }(\S v\cdot c) \;=\; \S v\cdot b \wedge c$

$x: \S p \;=\; x: \Delta p \;\wedge\; px$

$\forall f \;=\; (\S f){=}(\Delta f)$

$\exists f \;=\; (\S f){\ne}null$

Bounding Laws

if v does not appear in n

$n > (MAX\ v: D\cdot m) \;\Rightarrow\; (\forall v: D\cdot n{>}m)$

$n < (MIN\ v: D\cdot m) \;\Rightarrow\; (\forall v: D\cdot n{<}m)$

$n \ge (MAX\ v: D\cdot m) \;=\; (\forall v: D\cdot n{\ge}m)$

$n \le (MIN\ v: D\cdot m) \;=\; (\forall v: D\cdot n{\le}m)$

$n \ge (MIN\ v: D\cdot m) \;\Leftarrow\; (\exists v: D\cdot n{\ge}m)$

$n \le (MAX\ v: D\cdot m) \;\Leftarrow\; (\exists v: D\cdot n{\le}m)$

$n > (MIN\ v: D\cdot m) \;=\; (\exists v: D\cdot n{>}m)$

$n < (MAX\ v: D\cdot m) \;=\; (\exists v: D\cdot n{<}m)$

Splitting Laws — for any fixed domain

$\forall v\cdot a \wedge b \;=\; (\forall v\cdot a) \;\wedge\; (\forall v\cdot b)$

$\exists v\cdot a \wedge b \;\Rightarrow\; (\exists v\cdot a) \;\wedge\; (\exists v\cdot b)$

$\forall v\cdot a \vee b \;\Leftarrow\; (\forall v\cdot a) \;\vee\; (\forall v\cdot b)$

$\exists v\cdot a \vee b \;=\; (\exists v\cdot a) \;\vee\; (\exists v\cdot b)$

$\forall v\cdot a \Rightarrow b \;\Rightarrow\; (\forall v\cdot a) \;\Rightarrow\; (\forall v\cdot b)$

$\forall v\cdot a \Rightarrow b \;\Rightarrow\; (\exists v\cdot a) \;\Rightarrow\; (\exists v\cdot b)$

$\forall v\cdot a = b \;\Rightarrow\; (\forall v\cdot a) \;=\; (\forall v\cdot b)$

$\forall v\cdot a = b \;\Rightarrow\; (\exists v\cdot a) \;=\; (\exists v\cdot b)$

Commutative Laws

$\forall v\cdot \forall w\cdot b \;=\; \forall w\cdot \forall v\cdot b$

$\exists v\cdot \exists w\cdot b \;=\; \exists w\cdot \exists v\cdot b$

Semicommutative Laws (Skolem)

$\exists v\cdot \forall w\cdot b \;\Rightarrow\; \forall w\cdot \exists v\cdot b$

$\forall x\cdot \exists y\cdot Pxy \;=\; \exists f\cdot \forall x\cdot Px(fx)$

Domain Change Laws

$A: B \;\Rightarrow\; (\forall v: A\cdot b) \;\Leftarrow\; (\forall v: B\cdot b)$

$A: B \;\Rightarrow\; (\exists v: A\cdot b) \;\Rightarrow\; (\exists v: B\cdot b)$

$\forall v: A\cdot v: B \Rightarrow p \;=\; \forall v: A\text{‘}B\cdot p$

$\exists v: A\cdot v: B \wedge p \;=\; \exists v: A\text{‘}B\cdot p$

Extreme Law

$(MIN\ v\cdot n) \le n \le (MAX\ v\cdot n)$

Connection Laws (Galois)

$n{\le}m \;=\; \forall k\cdot\ k{\le}n \Rightarrow k{\le}m$

$n{\le}m \;=\; \forall k\cdot\ k{<}n \Rightarrow k{<}m$

$n{\le}m \;=\; \forall k\cdot\ m{\le}k \Rightarrow n{\le}k$

$n{\le}m \;=\; \forall k\cdot\ m{<}k \Rightarrow n{<}k$

Distributive Laws — if $D{\ne}null$ and v does not appear in n

$max\ n\ (MAX\ v: D\cdot m) \;=\; (MAX\ v: D\cdot max\ n\ m)$

$max\ n\ (MIN\ v: D\cdot m) \;=\; (MIN\ v: D\cdot max\ n\ m)$

$min\ n\ (MAX\ v: D\cdot m) \;=\; (MAX\ v: D\cdot min\ n\ m)$

$min\ n\ (MIN\ v: D\cdot m) \;=\; (MIN\ v: D\cdot min\ n\ m)$

$n + (MAX\ v: D\cdot m) \;=\; (MAX\ v: D\cdot n{+}m)$

$n + (MIN\ v: D\cdot m) \;=\; (MIN\ v: D\cdot n{+}m)$

$n - (MAX\ v: D\cdot m) \;=\; (MIN\ v: D\cdot n{-}m)$

$n - (MIN\ v: D\cdot m) \;=\; (MAX\ v: D\cdot n{-}m)$

$(MAX\ v: D\cdot m) - n \;=\; (MAX\ v: D\cdot m{-}n)$

$(MIN\ v: D\cdot m) - n \;=\; (MIN\ v: D\cdot m{-}n)$

$n{\ge}0 \;\Rightarrow\; n \times (MAX\ v: D\cdot m) \;=\; (MAX\ v: D\cdot n{\times}m)$

$n{\ge}0 \;\Rightarrow\; n \times (MIN\ v: D\cdot m) \;=\; (MIN\ v: D\cdot n{\times}m)$

$n{\le}0 \;\Rightarrow\; n \times (MAX\ v: D\cdot m) \;=\; (MIN\ v: D\cdot n{\times}m)$

$n{\le}0 \;\Rightarrow\; n \times (MIN\ v: D\cdot m) \;=\; (MAX\ v: D\cdot n{\times}m)$

$n \times (\Sigma v: D\cdot m) \;=\; (\Sigma v: D\cdot n{\times}m)$

$(\Pi v: D\cdot m)^n \;=\; (\Pi v: D\cdot m^n)$

11.4.9 Limits

$(MAX\ m\cdot\ MIN\ n\cdot\ f(m+n)) \ \le\ (LIM\ f) \ \le\ (MIN\ m\cdot\ MAX\ n\cdot\ f(m+n))$
$\exists m\cdot\ \forall n\cdot\ p(m+n) \ \Longrightarrow\ LIM\ p \ \Longrightarrow\ \forall m\cdot\ \exists n\cdot\ p(m+n)$

End of Limits

11.4.10 Specifications and Programs

For specifications P , Q , R , and S , and boolean b ,

$ok \ = \ x'=x \ \wedge\ y'=y \ \wedge\ ...$
$x:= e \ = \ x'=e \ \wedge\ y'=y \ \wedge\ ...$
$P.\ Q \ = \ \exists x'', y'', ...\cdot\ \langle x', y', ...\rightarrow P\rangle\ x''\ y''\ ...\ \wedge\ \langle x, y, ...\rightarrow Q\rangle\ x''\ y''\ ...$
$P\|Q \ = \ \exists tP, tQ\cdot\ \langle t'\rightarrow P\rangle tP \ \wedge\ \langle t'\rightarrow Q\rangle tQ \ \wedge\ t' = max\ tP\ tQ$
if b **then** P **else** $Q \ = \ b \wedge P \vee \neg b \wedge Q$
var x: $T\cdot\ P \ = \ \exists x, x': T\cdot\ P$
frame $x\cdot\ P \ = \ P \ \wedge\ y'=y \ \wedge\ ...$
while b **do** $P \ = \ t'\ge t \wedge$ (**if** b **then** (P. $t:= t+1$. **while** b **do** P) **else** ok)
$\forall\sigma, \sigma'\cdot$ (**if** b **then** (P. W) **else** $ok \ \Leftarrow\ W$) $\ \Longrightarrow\ \forall\sigma, \sigma'\cdot$ (**while** b **do** $P \ \Leftarrow\ W$)
$(Fmn \ \Leftarrow\ m=n \wedge ok) \wedge (Fik \ \Leftarrow\ m\le i<j<k\le n \wedge (Fij.\ Fjk))$
$\Rightarrow$ ($Fmn \ \Leftarrow$ **for** $i:= m;..n$ **do** $m\le i<n \Rightarrow Fi(i+1)$)
$Im\Rightarrow I'n \ \Leftarrow$ **for** $i:= m;..n$ **do** $m\le i<n \wedge Ii \Rightarrow I'(i+1)$
wait until $w \ = \ t:= max\ t\ w$
assert $b \ =$ **if** b **then** ok **else** ($print$ "error". **wait until** ∞)
ensure $b \ = \ b \wedge ok$
$x' = (P$ **result** $e) \ = \ P.\ x' = e$
$c? \ = \ r:= r+1$
$c \ = \ \mathbb{M}c_{\ rc-1}$
$c!\ e \ = \ \mathbb{M}c_{\ wc} = e \ \wedge\ \mathbb{T}c_{\ wc} = t \ \wedge\ (wc:= wc+1)$
$\surd c \ = \ \mathbb{T}c_{\ rc}$ + (transit time) $\le t$
ivar x: $T\cdot\ S \ = \ \exists x: time\rightarrow T\cdot\ S$
chan c: $T\cdot\ P \ = \ \exists\mathbb{M}c: \infty{*}T\cdot\ \exists\mathbb{T}c: \infty{*}xreal\cdot$ **var** rc , wc: $xnat := 0\cdot\ P$

$ok.\ P \ = \ P.\ ok \ = \ P$	identity
$P.\ (Q.\ R) \ = \ (P.\ Q).\ R$	associativity
if b **then** P **else** $P \ = \ P$	idempotence
if b **then** P **else** $Q \ =$ **if** $\neg b$ **then** Q **else** P	case reversal
$P \ =$ **if** b **then** $b \Rightarrow P$ **else** $\neg b \Rightarrow P$	case creation
$P\vee Q.\ R\vee S \ = \ (P.\ R) \vee (P.\ S) \vee (Q.\ R) \vee (Q.\ S)$	distributivity
(**if** b **then** P **else** Q). $R \ =$ **if** b **then** ($P.\ R$) **else** ($Q.\ R$)	distributivity (unprimed b)
$ok \,\|\, P \ = \ P \,\|\, ok \ = \ P$	identity
$P \,\|\, Q \ = \ Q \,\|\, P$	symmetry
$P \,\|\, (Q \,\|\, R) \ = \ (P \,\|\, Q) \,\|\, R$	associativity
$P \,\|\, Q\vee R \ = \ (P \,\|\, Q) \vee (P \,\|\, R)$	distributivity
$P \,\|$ **if** b **then** Q **else** $R \ =$ **if** b **then** ($P \,\|\, Q$) **else** ($P \,\|\, R$)	distributivity
if b **then** ($P\|Q$) **else** ($R\|S$) $=$ **if** b **then** P **else** $R \,\|$ **if** b **then** Q **else** S	distributivity
$x:=$ **if** b **then** e **else** $f \ =$ **if** b **then** $x:= e$ **else** $x:= f$	functional-imperative

End of Specifications and Programs

11.4.11 Substitution

Let x and y be different boundary state variables, let e and f be expressions of the prestate, and let P be a specification.

$x{:=}\,e.\,P \;=\;$ (for x substitute e in P)

$(x{:=}\,e \,\|\, y{:=}\,f).\; P \;=\;$ (for x substitute e and independently for y substitute f in P)

End of Substitution

11.4.12 Conditions

Let P and Q be any specifications, and let C be a precondition, and let C' be the corresponding postcondition (in other words, C' is the same as C but with primes on all the state variables).

$C \wedge (P.\,Q) \;\Leftarrow\; C{\wedge}P.\,Q$

$C \Rightarrow (P.Q) \;\Leftarrow\; C{\Rightarrow}P.\,Q$

$(P.Q) \wedge C' \;\Leftarrow\; P.\,Q{\wedge}C'$

$(P.Q) \Leftarrow C' \;\Leftarrow\; P.\,Q{\Leftarrow}C'$

$P.\,C{\wedge}Q \;\Leftarrow\; P{\wedge}C'.\,Q$

$P.\,Q \;\Leftarrow\; P{\wedge}C'.\; C{\Rightarrow}Q$

C is a sufficient precondition for P to be refined by S
if and only if $C{\Rightarrow}P$ is refined by S .

C' is a sufficient postcondition for P to be refined by S
if and only if $C'{\Rightarrow}P$ is refined by S .

End of Conditions

11.4.13 Refinement

Refinement by Steps (Stepwise Refinement) (monotonicity, transitivity)

If $A \Leftarrow$ **if** b **then** C **else** D and $C \Leftarrow E$ and $D \Leftarrow F$ are theorems,
then $A \Leftarrow$ **if** b **then** E **else** F is a theorem.

If $A \Leftarrow B.C$ and $B \Leftarrow D$ and $C \Leftarrow E$ are theorems, then $A \Leftarrow D.E$ is a theorem.

If $A \Leftarrow B\|C$ and $B \Leftarrow D$ and $C \Leftarrow E$ are theorems, then $A \Leftarrow D\|E$ is a theorem.

If $A \Leftarrow B$ and $B \Leftarrow C$ are theorems, then $A \Leftarrow C$ is a theorem.

Refinement by Parts (monotonicity, conflation)

If $A \Leftarrow$ **if** b **then** C **else** D and $E \Leftarrow$ **if** b **then** F **else** G are theorems,
then $A{\wedge}E \Leftarrow$ **if** b **then** $C{\wedge}F$ **else** $D{\wedge}G$ is a theorem.

If $A \Leftarrow B.C$ and $D \Leftarrow E.F$ are theorems, then $A{\wedge}D \Leftarrow B{\wedge}E.\,C{\wedge}F$ is a theorem.

If $A \Leftarrow B\|C$ and $D \Leftarrow E\|F$ are theorems, then $A{\wedge}D \Leftarrow B{\wedge}E \,\|\, C{\wedge}F$ is a theorem.

If $A \Leftarrow B$ and $C \Leftarrow D$ are theorems, then $A{\wedge}C \Leftarrow B{\wedge}D$ is a theorem.

Refinement by Cases

$P \Leftarrow$ **if** b **then** Q **else** R is a theorem if and only if
$P \Leftarrow b \wedge Q$ and $P \Leftarrow \neg b \wedge R$ are theorems.

End of Refinement

End of Laws

11.5 Names

Name	Law
abs: $xreal \rightarrow \S r: xreal \cdot r \ge 0$	$abs\ r\ =\ $ **if** $r \ge 0$ **then** r **else** $-r$
$bool$ (the booleans)	$bool\ =\ \top, \bot$
$ceil$: $real \rightarrow int$	$r \le ceil\ r < r+1$
$char$ (the characters)	$char\ =\ ...,$ "a", "A", ...
div: $real \rightarrow (\S r: real \cdot r>0) \rightarrow int$	$div\ x\ y\ =\ floor\ (x/y)$
$divides$: $(nat+1) \rightarrow int \rightarrow bool$	$divides\ n\ i\ =\ i/n: int$
$entro$: $prob \rightarrow \S r: xreal \cdot r \ge 0$	$entro\ p\ =\ p \times info\ p\ +\ (1-p) \times info\ (1-p)$
$even$: $int \rightarrow bool$	$even\ i\ =\ i/2: int$
	$even\ =\ divides\ 2$
$floor$: $real \rightarrow int$	$floor\ r \le r < floor\ r + 1$
$info$: $prob \rightarrow \S r: xreal \cdot r \ge 0$	$info\ p\ =\ -\ log\ p$
int (the integers)	$int\ =\ nat, -nat$
LIM (limit quantifier)	see Laws
log: $(\S r: xreal \cdot r \ge 0) \rightarrow xreal$	$log\ (2^x) = x$
	$log\ (x \times y)\ =\ log\ x + log\ y$
max: $xrat \rightarrow xrat \rightarrow xrat$	$max\ x\ y\ =\ $ **if** $x \ge y$ **then** x **else** y
	$-\ max\ a\ b\ =\ min\ (-a)\ (-b)$
MAX (maximum quantifier)	see Laws
min: $xrat \rightarrow xrat \rightarrow xrat$	$min\ x\ y\ =\ $ **if** $x \le y$ **then** x **else** y
	$-\ min\ a\ b\ =\ max\ (-a)\ (-b)$
MIN (minimum quantifier)	see Laws
mod: $real \rightarrow (\S r: real \cdot r>0) \rightarrow real$	$0 \le mod\ a\ d < d$
	$a\ =\ div\ a\ d \times d\ +\ mod\ a\ d$
nat (the naturals)	$0, nat+1: nat$
	$0, B+1: B\ \Rightarrow\ nat: B$
nil (the empty string)	$\leftrightarrow nil\ =\ 0$
	$nil; S\ =\ S\ =\ S; nil$
	$nil \le S$
$null$ (the empty bunch)	$¢null\ =\ 0$
	$null, A\ =\ A\ =\ A, null$
	$null: A$
odd: $int \rightarrow bool$	$odd\ i\ =\ \neg\ i/2: int$
	$odd\ =\ \neg even$
ok (the empty program)	$ok\ =\ \sigma'=\sigma$
	$ok.P\ =\ P.ok\ =\ ok \parallel P\ =\ P \parallel ok\ =\ P$
$prob$ (probability)	$prob\ =\ \S r: real \cdot 0 \le r \le 1$
$rand$ (random number)	$rand\ n: 0,..n$
rat (the rationals)	$rat\ =\ int/(nat+1)$
$real$ (the reals)	$r: real\ =\ r: xreal\ \wedge\ -\infty<r<\infty$
suc: $nat \rightarrow (nat+1)$	$suc\ n = n+1$
$xint$ (the extended integers)	$xint\ =\ -\infty, int, \infty$
$xnat$ (the extended naturals)	$xnat\ =\ nat, \infty$
$xrat$ (the extended rationals)	$xrat\ =\ -\infty, rat, \infty$
$xreal$ (the extended reals)	$x: xreal\ =\ \exists f: nat \rightarrow rat \cdot x = LIM\ f$

End of Names

11.6 Symbols

Symbol	Page	Meaning
⊤	3	true
⊥	3	false
¬	3	not
∧	3	and
∨	3	or
⇒	3	implies
⟹	3	implies
⇐	3	follows from, is implied by
⟸	3	follows from, is implied by
=	3	equals, if and only if
=	3	equals, if and only if
⧧	3	differs from, is unequal to
<	13	less than
>	13	greater than
≤	13	less than or equal to
≥	13	greater than or equal to
+	12	plus
+	20	list catenation
–	12	minus
×	12	times, multiplication
/	12	divided by
,	14	bunch union
,..	16	union from (incl) to (excl)
‘	14	bunch intersection
;	17	string catenation
;..	19	catenation from (incl) to (excl)
:	14	is in, are in, bunch inclusion
::	89	includes
:=	36	assignment
.	36	dep't (sequential) composition
·	26	quantifier abbreviation
!	133	output
?	133	input
√	133	input check

<table>
<tr><td>( )</td><td>4</td><td>parentheses for grouping</td></tr>
<tr><td>{ }</td><td>17</td><td>set brackets</td></tr>
<tr><td>[ ]</td><td>20</td><td>list brackets</td></tr>
<tr><td>⟨ ⟩</td><td>23</td><td>function (scope) brackets</td></tr>
<tr><td>ϟ</td><td>17</td><td>power</td></tr>
<tr><td>¢</td><td>14</td><td>bunch size, cardinality</td></tr>
<tr><td>$</td><td>17</td><td>set size, cardinality</td></tr>
<tr><td>↔</td><td>18</td><td>string size, length</td></tr>
<tr><td>#</td><td>20</td><td>list size, length</td></tr>
<tr><td>|</td><td>20,24</td><td>selective union, otherwise</td></tr>
<tr><td>‖</td><td>118</td><td>indep't (parallel) composition</td></tr>
<tr><td>~</td><td>17</td><td>contents of a set</td></tr>
<tr><td>~</td><td>20</td><td>contents of a list</td></tr>
<tr><td>*</td><td>18</td><td>repetition of a string</td></tr>
<tr><td>Δ</td><td>23</td><td>domain of a function</td></tr>
<tr><td>→</td><td>23</td><td>function arrow</td></tr>
<tr><td>∈</td><td>17</td><td>element of a set</td></tr>
<tr><td>⊆</td><td>17</td><td>subset</td></tr>
<tr><td>∪</td><td>17</td><td>set union</td></tr>
<tr><td>∩</td><td>17</td><td>set intersection</td></tr>
<tr><td>@</td><td>22</td><td>index with a pointer</td></tr>
<tr><td>∀</td><td>26</td><td>for all, universal quantifier</td></tr>
<tr><td>∃</td><td>26</td><td>there exists, existential quantifier</td></tr>
<tr><td>Σ</td><td>26</td><td>sum of, summation quantifier</td></tr>
<tr><td>Π</td><td>26</td><td>product of, product quantifier</td></tr>
<tr><td>§</td><td>28</td><td>those, solution quantifier</td></tr>
<tr><td>′</td><td>34</td><td>x' is final value of state var x</td></tr>
<tr><td>"</td><td>13,19</td><td>"hello" is a text or string of chars</td></tr>
<tr><td>a^b</td><td>12</td><td>exponentiation</td></tr>
<tr><td>a_b</td><td>18</td><td>string indexing</td></tr>
<tr><td>$a\ b$</td><td>20,31</td><td>indexing,application,composition</td></tr>
<tr><td>◃ ▹</td><td>18</td><td>string modification</td></tr>
<tr><td>∞</td><td>12</td><td>infinity</td></tr>
</table>

Keyword	Page
assert	77
chan	138
ensure	77
exit when	71
for do	74
frame	67
go to	76
if then else	4
ivar	126
loop end	71
or	77
result	78
var	66
wait until	76
while do	69

End of Symbols

11.7 Precedence

0	⊤ ⊥ ( ) { } [ ] ⟨ ⟩ **loop end** numbers texts names
1	@ juxtaposition
2	prefix– ¢ $ ↔ # * ~ ~ ⚡ Δ → √ superscript subscript
3	× / ∩
4	+ infix– + ∪
5	; ;.. ‘
6	, ,.. ǀ ◃ ▹
7	= ⧧ < > ≤ ≥ : :: ∈ ⊆
8	¬
9	∧
10	∨
11	⇒ ⇐
12	:= ! ?
13	**if then else while do exit when for do go to wait until assert ensure or**
14	. ǁ **result**
15	∀· ∃· Σ· Π· §· *LIM*· *MAX*· *MIN*· **var**· **ivar**· **chan**· **frame**·
16	= ⇒ ⇐

On level 2, superscripting and subscripting serve to bracket all operations within them.

Juxtaposition associates from left to right, so $a\ b\ c$ means the same as $(a\ b)\ c$. The infix operators @ / – associate from left to right. The infix operators * → associate from right to left. The infix operators × ∩ + + ∪ ; ‘ , ǀ ∧ ∨ . ǁ are associative (they associate in both directions).

On levels 7, 11, and 16 the operators are continuing. For example, $a = b = c$ neither associates to the left nor associates to the right, but means the same as $a = b \ \wedge\ b = c$. On any one of these levels, a mixture of continuing operators can be used. For example, $a \le b < c$ means the same as $a \le b \ \wedge\ b < c$.

On levels 13 and 15, the precedence applies to the final operand (and to both operands of **or**), not to operands that are surrounded by the operator.

The operators = ⇒ ⇐ are identical to = ⇒ ⇐ except for precedence.

End of Precedence

11.8 Distribution

The operators in the following expressions distribute over bunch union in any operand:

[*A*] *A*@*B* *A B* –*A* $*A* ↔*A* #*A* ~*A* ~*A*

A^B A_B $A{\times}B$ A/B $A{\cap}B$ $A{+}B$ $A{-}B$ A^+B $A{\cup}B$ $A;B$ *A*‘*B*

$\neg A$ $A{\wedge}B$ $A{\vee}B$

The operator in $A{*}B$ distributes over bunch union in its left operand only.

End of Distribution

End of Reference

End of a Practical Theory of Programming

Formal Proof

Yannis Kassios

February 20, 2009

A *formal proof* is *not* a natural language argument. It is a *calculation* that follows *precise rules*. This is the whole point of Formal Methods. Instead of using natural (informal) language to reason about program correctness, we use formal notation and proof. Formal notation and proof are rigorous, unambiguous and can be checked mechanically.

In this course, *all proofs will be formal*, unless otherwise specified. Therefore, it is essential to learn the basics of formal proofs. We start by proving small boring theorems of boolean theory. Later on, we will move to more interesting theories, but it is essential that we get this formal proof thing right first.

After discussing the basics (Sect. 1), we will move on to the proofs of special forms of expressions (Sect. 2). In Sect. 3 we discuss the advanced techniques of *monotonicity* and *context*.

1 Basics

So how do we prove a boolean expression, say A? A common way to go is to write a big series of equations

$$A = B = C = ... = \top$$

such that, each individual equation (i.e. $A = B$, $B = C$ and so forth) is an "obvious" theorem (more on what "obvious" means, later on in Sect. 1.1). By the transitivity of equality, this proves that $A = \top$ which classifies A as a theorem. Of course, we could also go the other way, beginning with $\top$ and working our way to A.

More generally, we can prove that $\top \Rightarrow A$ (or $A \Leftarrow \top$). This proves that A is *at least as true as* $\top$ (or $\top$ is at least as false as A). Since $\top$ is always true, this makes A always true, which classifies it again as a theorem.

We are thus allowed to use not only equations, but also implications in our big proving formula, e.g.:

$$A = B \Leftarrow C = D \Leftarrow ... \Leftarrow \top$$

The transitivity properties of both equality and implication mean that the above proves $A \Leftarrow \top$, which proves A. Again, each individual step must be an "obvious" theorem.

The solution to Exercise 2(m), as given in the Self-Check solutions for Week 1 is a good example of this type of proof.

To prove A, we also have the (dual) option of disproving $\neg A$. This can be done by proving $\bot \Rightarrow \neg A$ in a similar manner.

1.1 What is "obvious"

So what is an "obvious" step in a proof? In this exercise, it basically means that it is a direct instantiation of one of the boolean laws at the back of the book (see the examples in the lecture, the book and the self-check solutions). You may also prove your own lemmas and then use them in a proof as "obvious" steps.

Later on, when we start tackling real programming problems, this will turn out to be an overkill. In those later exercises, several very obvious steps may be skipped, as long as you convince the marker that you know what you are doing. But in this exercise we are talking basic stuff, so you must be very detailed in your proofs.

You are allowed to use the transparency property of equality and substitute a sub-expression with an equal sub-expression, citing the relevant law. In that case, it helps if you underline the substituted expression, as in the following example:

$$\begin{array}{ll} & \underline{a \vee a} \Rightarrow b \quad \text{idempotence} \\ = & a \Rightarrow b \end{array}$$

Of course, you don't have to do that, if the sub-expression is obvious. But in complicated expressions, it helps the marker understand what you are doing without frustrating him or her; we don't want that, do we?

1.2 Common Pitfalls

It is noteworthy to remember not to fall into any of the following traps:

- Starting from $\top$ and strengthening instead of weakening, i.e.

  $\top \Leftarrow Z \Leftarrow Y ... \Leftarrow A$ **WRONG!**

  or starting from A and weakening instead of strengthening, i.e.

  $A \Rightarrow Z \Rightarrow Y ... \Rightarrow \top$ **WRONG!**

  This proves that A implies $\top$, which we already know is a theorem, but does not prove anything about A. Remember, if you want to prove A, you should weaken $\top$ to A or strengthen A to $\top$.

- Mixing $\Rightarrow$ and $\Leftarrow$. Don't do that. All implications must go the same way. Otherwise, you don't prove much:

  $A \Leftarrow B \Rightarrow C$

means that B implies A and that B implies C, but it says nothing about the relation between A and C.

- A lot of people think of proofs as a sequence of expressions which are not connected. This is not correct. A proof is *one long boolean expression.* The connectors between the various parts of the proof *must* be there. They are very important.

  If you forget the connectors, not only you are making a seemingly insignificant formal mistake in your syntax, you also run the more serious risk of producing a wrong proof. This is because the connectors actually tell you which direction your implications are going. As I've noted above, the direction is very important.

  Here is an example of a "proof" gone bad:

$(a \wedge \top) \ \vee \ b$	specialization	**WRONG!**
$\top \ \vee \ b$	base	**WRONG!**
$\top$		**WRONG!**

  The connectors are missing, which means that the proof is wrong anyway. But it is not a wrong proof of a theorem. It is a wrong proof of a non-theorem, which is a serious mistake. What went wrong? Here is what we see when we use connectors:

	$(a \wedge \top) \ \vee \ b$	specialization
$\Rightarrow$	$\top \ \vee \ b$	base
$=$	$\top$	

  We see that this is not a proof for $(a \wedge \top) \ \vee \ b$, because it uses the wrong direction in the implication (see the point above).

2 Proving special forms

If A is of some special form, we don't have to reduce it to $\top$.

If A is an equality, we can prove it by starting from the left side of the equality and working our way to the right side (or we can start from the right and move to the left). Remember, in this case, we are only permitted to use equality in each step; implication won't do.

If A is an implication, we can start from one of the operands and move to the other. We can use equality and implication in each step, but the direction of the implication must be the same as the direction of the implication in A. For example, the solution of 2(m) can be done as follows:

	$a \Rightarrow \neg a$	material implication
$=$	$\neg a \vee \neg a$	idempotence
$=$	$\neg a$	reflexivity
$\Rightarrow$	$\neg a$	

If A is a conjunction, we can prove each conjunct separately.

3 Monotonicity and Context

And, finally, we will talk about the advanced techniques of *(anti-)monotonicity* and *context*. Please learn these techniques very well. Not only they are very helpful in proving theorems, they are also a source of common errors if not properly applied.

3.1 (Anti-)monotonicity

If a sub-expression is in a *monotonic* context, you are allowed to substitute it for a weaker (stronger) sub-expression and weaken (strengthen) the whole expression. If a sub-expression is in an *anti-monotonic* context, you are allowed to substitute it for a weaker (stronger) sub-expression and strengthen (weaken) the whole expression. Again, it helps if you underline the relevant expression.

For example, let us do 2(q) using anti-monotonicity.

$$\begin{array}{lll} & \neg(a \Rightarrow \underline{b}) & \text{antimonoticity, base} \\ \Rightarrow & \neg(\underline{a \Rightarrow \bot}) & \text{indirect proof} \\ = & \neg\neg a & \text{double negation} \\ = & a & \end{array}$$

What we did in the first step is substitute b with $\bot$. By base law, we know that b is weaker than $\bot$ (i.e. $b \Leftarrow \bot$ is a theorem) and because b is in an anti-monotonic position, this substitution strengthens the whole expression (the direction of the implication is now $\Rightarrow$).

3.1.1 Common Pitfalls

The commonest problems with the use of monotonicity are as follows:

- Ignoring whether a sub-expression is in a monotonic or anti-monotonic context. Monotonic contexts preserve the direction of the implication, while anti-monotonic contexts reverse it. If you forget that, then you might end up making a mistake such as this:

  $$\begin{array}{lll} & \underline{a} \Rightarrow b & \text{base: } a \Rightarrow \top \text{ - } \textbf{WRONG!} \\ \Rightarrow & \top \Rightarrow b & \text{identity} \\ = & b & \end{array}$$

  Here, we used $a \Rightarrow \top$ to substitute a with $\top$. But we disregarded the context of a and the overall direction of the implication is wrong. Since a is in an anti-monotonic context, the direction of the overall implication should have been reversed, like this:

  $$\begin{array}{lll} & \underline{a} \Rightarrow b & \text{base: } a \Rightarrow \top \\ \Leftarrow & \top \Rightarrow b & \text{identity} \\ = & b & \end{array}$$

- Disregarding the fact that some expressions are in neither monotonic nor anti-monotonic context. This means that no substitution can be made based on an implication law. For example, the following doesn't work:

$$\begin{array}{lll} & \underline{a} = b & \text{base: } a \Rightarrow \top \text{ - } \textbf{WRONG!} \\ \Rightarrow & \top = b & \text{identity} \\ = & b & \end{array}$$

  Here, the sub-expression a is neither in a monotonic context nor in an antimonotonic one. So we are *not* allowed to use an implication law such as $a \Rightarrow \top$ to substitute it in the expression.

Both problems mentioned here are very likely to occur if one forgets the connectors between the lines of a proof, as mentioned in Sect. 1.2.

3.2 The Context Rule

The second advanced rule that you can use is the context rule. There are many context rules listed on p.11 of the book. Here let us use a context rule for implication and redo 2(m):

$$\begin{array}{lll} & a \Rightarrow \underline{\neg a} & \text{context (assume } a \text{ is a theorem in the underlined expression)} \\ = & a \Rightarrow \neg\top & \text{boolean axiom} \\ = & a \Rightarrow \bot & \text{indirect proof} \\ = & \neg a & \end{array}$$

In the first step, we assumed that a is a theorem while transforming the underlined sub-expression. The context of the underlined sub-expression allows us to do that. The context rule is very powerful and therefore very useful later on in proofs of program correctness.

3.2.1 Common Pitfalls

Don't forget one basic thing about the context rule: you are allowed to transform *one of the operands* at a time. Not both of them. Transforming both operands using the context rule is *wrong*. For example, in $a \wedge a$, you are allowed to use the context rule to transform the expression to $a \wedge \top$ or to $\top \wedge a$. But you *cannot* use it simultaneously to both operands, transforming the expression to $\top \wedge \top$.

Some people use the context rule totally inappropriately in situations in which no context rule can apply. There are context rules for conjunctions, disjunctions and implications but there is no context rule for equality. Thus, the following is wrong:

$$\begin{array}{lll} & a = \underline{a} & \text{context rule (assume } a \text{ in the underlined expression) - } \textbf{WRONG!} \\ = & a = \top & \text{identity} \\ = & a & \end{array}$$

There is no context rule for $a = b$ that justifies changing b using a as a theorem, so we cannot do that.

4 Key Points

- **A Formal Proof is not a natural language argument but a mathematical calculation**. This calculation takes the form of a (usually big) boolean expression.
- **To prove an expression A, we weaken $\top$ to A or strengthen A to $\top$.** This means that the proof is of the form $A \Leftarrow ... \Leftarrow \top$ or $\top \Rightarrow ... \Rightarrow A$. The direction of the implication is **very important** and cannot be reversed.
- **A proof is a single boolean expression and not a sequence of expressions**. Connectors between the seperate lines of the proof are important and should never be forgotten.
- **Special Forms of Expressions can be proved in different ways**. To prove an equality $A = B$ you may start from A and end in B (or from B to A) following only equation steps. To prove an implication $A \Rightarrow B$, you may start from A and end in B following implication steps (or from B to A following reverse implication steps). To prove a disjunction $A \wedge B$, you may prove A and B seperately.
- **Monotonicity is important when substituting sub-expressions**. Substituting a sub-expression a for a weaker one makes the whole expression weaker if a is in a monotonic context and stronger if a is in an anti-monotonic context.
- **Some sub-expressions are in neither a monotonic nor an anti-monotonic context**. A typical example is the operands of an equality. We are not allowed to substitute such sub-expressions with weaker or stronger ones in a proof.
- **The most powerful proof technique is the context rule**. It involves transforming part of an expression assuming another part as a theorem. Various context rules are included in the textbook. Care must be taken to ensure that a **valid context rule applies** when we are substituting a sub-expression using context.

Quantifier Use

Yannis Kassios

February 20, 2009

To illustrate how we use quantifiers in formalization problems, here is the solution to Q.72(a) of the textbook. We are asked to formalize:

> natural n is the largest proper (neither 1 nor m) factor of natural m

This is a sentence that talks about natural numbers n and m. Therefore, these are the only free (not bound by quantifiers) variables that we are allowed to use in the formal expression.

Formalization usually needs a lot of *rephrasing* steps. The purpose of rephrasing is to bring the informal sentence closer to the formal notation. We do that by making the informal sentence *more precise*.

In our problem, the first step is to rephrase the sentence so that the logical connectives in it become apparent:

> n is not m and n is not 1 and n is a factor of m and n is greater than or equal to any other natural number that has these properties

Formalization of most of this sentence is immediate:

$n \neq m \;\wedge\; n \neq 1 \;\wedge\;$ (n is a factor of m)
$\wedge\;$ (n is greater than or equal to any other natural number that has these properties)

We can further simplify as follows:

$\neg n : 1, m \;\wedge\;$ (n is a factor of m)
$\wedge\;$ (n is greater than or equal to any other natural number that has these properties)

We are closer to the formal language. Let us focus on the last conjunct, which is still informal:

> n is greater than or equal to any other natural number that has these properties

How do we formalize "these properties"? We have to be more specific:

> n is greater than or equal to any other natural number that is not m and is not 1 and is a factor of m

To formalize "n is greater than or equal to any natural number that has property p"? We can rephrase by saying "all natural numbers that have property p are less than or equal to n". Or, "for any natural number x, if x has property p, then x is less than or equal to n". So the last conjunct is formalized as follows:

$$\forall x : nat \cdot \ \neg x : 1, m \ \wedge \ (x \text{ is a factor of } m) \ \ \Rightarrow \ x \leq n$$

Now, we know that a factor of m cannot be greater than m itself and it cannot be 0. So, the conjunct (x is a factor of m) allows us to restrict the domain of our formalization:

$$\forall x : 1, ..m + 1 \cdot \ \neg x : 1, m \ \wedge \ (x \text{ is a factor of } m) \ \ \Rightarrow \ x \leq n$$

Also, the conjunct $x : 1, m$ excludes values 1 and m from the domain:

$$\forall x : 2, ..m \cdot \ (x \text{ is a factor of } m) \ \ \Rightarrow \ x \leq n$$

Finally, we must formalize the part "is a factor", which remains informal. We find this part in two different places in our formalization. Furthermore, in these two places, we talk about different numbers. Finally, the formalization of "is a factor" is by itself non-trivial. For all these reasons, we will define "is a factor" as a separate *predicate* of two variables:

$$factor = \langle x : nat \rightarrow \langle y : nat \rightarrow x \text{ is a factor of } y \rangle \rangle$$

The formalization of the original sentence is complete if we use that predicate:

$$\neg n : 1, m \ \wedge \ factor\ n\ m \ \wedge \ \forall x : 2, ..m \cdot \ factor\ x\ m \ \ \Rightarrow \ x \leq n$$

The *factor* predicate is formalized as follows: x is a factor of y means:

there is a natural number z such that $y = x \times z$

Formally:

$$factor = \langle x : nat \rightarrow \langle y : nat \rightarrow \exists z : nat \cdot \ y = x \times z \rangle \rangle$$

Programming Theory Basics

Yannis Kassios

January 14, 2009

In this course, there are three basic principles related to programming and specifications:

(a) A specification is a boolean expression (typically one that talks about the initial and the final values of the program variables).

(b) A program is a special kind of specification.

(c) Refinement (i.e. a specification satisfies another one) is expressed by implication $S \Rightarrow P$. Program correctness (i.e. a program A satisfies its specification B) is a special kind of refinement ($A \Rightarrow B$).

The hardest part to grasp is (b): a program is a special kind of specification. In other words, a program is a *boolean expression*. In particular, our programming theory equates a program with a boolean expression that describes what happens if we execute that program.

Now, a real world program does not look at all like a boolean expression. In it, there are constructs like assignments, sequential compositions, conditionals and loops. To achieve (b), our theory has to provide *definitions* for all these constructs. In fact, our theory defines programming notations as *abbreviations* for logical expressions.

Assignment

For example, *assignment* $x := E$ is defined as an *abbreviation*:

$$x := E \;=\; x' = E \;\wedge\; y' = y \;\wedge\; z' = z \;\wedge ...$$

where $x, y, z, ...$ are all the program variables. The definition says that $x := E$ changes the value of x, leaving everything else alone.

The important point is that, to treat assignment in our logic, we have to expand it into its logical equivalent. For example, suppose that our program variables are x and y and they are both integer. Suppose that our specification says "increase x". Formally, the specification is $x' > x$, i.e. the final value of x is greater than its initial value. How about assigning $x + 2$ to x? This seems to be a valid way to refine this specification. The refinement we need to prove is:

$$x := x + 2 \;\Longrightarrow\; x' > x$$

To prove this, we have to realize that $x := x + 2$ is an abbreviation for a boolean expression. We can't use logic, unless we expand the assignment, getting rid of the abbreviation. Because we said that our variables are x and y, the expansion is:

$$x' = x + 2 \underline{\wedge y' = y}$$

Notice the underlined part. Even though y does not appear in the abbreviation $x := x + 2$, it appears in the unabbreviated logical expression. The refinement proof goes as follows:

$$\begin{array}{ll} & (x := x + 2) \Rightarrow x' > x \qquad \text{expand assignment} \\ = & x' = x + 2 \ \wedge \ y' = y \ \Rightarrow \ \underline{x' > x} \\ & \quad \text{context (assume } x' = x + 2 \text{ in the underlined expression)} \\ = & x' = x + 2 \ \wedge \ y' = y \ \Rightarrow \ x + 2 > x \qquad \text{arithmetic} \\ = & x' = x + 2 \ \wedge \ y' = y \ \Rightarrow \ \top \qquad \text{base} \\ = & \top \end{array}$$

Now, you might think that $y' = y$ is not really that important. In this particular example, it isn't. But in general it is *very* important. You rely on the fact that the assignment changes the value of only one variable all the time when you are writing your programs (if your programming language supports pointers, then at least you hope that it is true). You will rely on it in your refinement formal proofs later on in this course too.

For the time being, let us look at an example that does rely on it. Suppose that the program variables are x, y. We would like to see the exact precondition under which $x := x + 1$ refines $y' > 2$. As you might guess, the precondition should be $y > 2$. The reason is that $x := x + 1$ does not change y, so to ensure that y is greater than 2 in the final state, we must assure it is true in the initial state. Our informal reasoning relies on the fact that $x := x + 1$ does not change y. So must our formal reasoning.

Now for the formal proof. Remember that the formula for the exact precondition is: $\forall \sigma' \cdot \ P \Leftarrow S$. In this example, our state is x, y, the specification S is $y' > 2$ and the specification P is $x := x + 1$. Let us put them all together:

$$\begin{array}{ll} & \forall x', y' \cdot \ y' > 2 \ \Leftarrow \ (x := x + 1) \qquad \text{expand assignment} \\ = & \forall x', y' \cdot \ y' > 2 \ \Leftarrow \ x' = x + 1 \ \wedge \ y' = y \\ & \qquad \text{one-point law twice} \\ & \qquad \text{i.e. substitute } (x + 1 \text{ for } x' \text{ and } y \text{ for } y') \text{ in } y' > 2 \\ = & y > 2 \end{array}$$

The common pitfalls regarding assignment have to do with not understanding that assignment *stands for* a boolean expression:

- (a) Some people go freely from $x := E$ to $x' = E$ or from $x' = E$ to $x := E$ disregarding all other program variables.
- (b) Some people confuse $=$ with $:=$. I have seen people write things like $x' := E$ which is completely wrong. I have also seen things like

if $x := 1$ **then** ... **else** ... which is not wrong grammatically (remember, $x := 1$ is a boolean expression!) but it is probably not what was intented. Remember: The assignment operator := and the equality operator = are *different*. Make sure you understand the differences between them.

The *ok* Statement

You find it in most languages as the empty statement. It is a statement that performs no change to the state. In our theory, it is an abbreviation of $x' = x \wedge y' = y \wedge ...$ where $x, y, ...$ are the program variables. Everything said about assignment above is also useful for the treatment of *ok*. Notice that $x := x$ is equal to *ok*.

Dependent Composition

You find it in most languages as *sequential composition* and usually it is denoted by semicolon. In our language, it is denoted by a dot and it connects not just programs, but specifications in general. Our theory defines it again as an abbreviation: when you see a sequential composition $P.Q$ you replace it with an existential quantification:

$$\begin{array}{lll} \exists x'', y'', ... \cdot & & (\text{substitute } x', y', ... \text{ with } x'', y'' \text{ in } P) \\ & \wedge & (\text{substitute } x, y, ... \text{ with } x'', y'' \text{ in } Q) \end{array}$$

As with assignment, to treat sequential composition in logic, we should replace it with the boolean expression it stands for. For example, in integer program variables x, y:

$$\begin{array}{ll} & x' > x + y \;.\; y' > x + y \\ = & \exists x'', y'' \cdot\; x'' > x + y \;\wedge\; y' > x'' + y'' \\ = & \exists x'', y'' \cdot\; x'' > x + y \;\wedge\; y' > x'' + y'' \;\wedge\; y' > 2 \times x + y \\ & \text{distribution: notice that } x'', y'' \text{ do \textit{not} appear in } y' > 2 \times x + y \\ & \hfill \text{otherwise the distribution step would be illegal} \\ = & (\exists x'', y'' \cdot\; x'' > x + y \;\wedge\; y' > x'' + y'') \;\wedge\; y' > 2 \times x + y \\ & \hfill \text{various steps left as an exercise on quantifiers} \\ = & \top \;\wedge\; y' > 2 \times x + y \\ = & y' > 2 \times x + y \end{array}$$

If one (or both) of the operands of dependent composition is an abbreviation, then *we have to expand them first before expanding the dependent composition.* Remember that the substitutions in the definition of depended composition work on the boolean expressions to its left and right operands, *not their abbreviations.* So the following examples don't make sense:

$$x := x+1 \;.\; y := y+1 \;=\; \exists x'', y'' \cdot\; (x := x+1) \wedge (y'' := y''+1) \quad \textbf{WRONG!}$$

$$ok \;.\; x' > y \;=\; \exists x'', y'' \cdot\; ok' \;\wedge\; x' > y'' \quad \textbf{WRONG!}$$

The correct way to expand these definitions (left as an exercise) gives respectively:

$$x' = x + 1 \ \wedge \ y' = y + 1$$

and

$$x' > y$$

Another mistake that I see quite often is to relace dependent composition with conjunction and vice versa. This often appears in interesting combinations with the common mistakes about assignments that I mentioned above. For example, it is not rare to see things like:

$$x := x + 2 \ . \ y := x + 1 \ \ = \ \ x' = x + 2 \ \wedge \ y' = x + 1 \quad \textbf{WRONG!}$$

(Notice in this example that the final value of y should be $x + 3$ and not $x + 1$. The final value of x is accidentally correct.)

See the solution to Ex. 97 in pages 38-39 of the book for more common traps regarding dependent composition.

Substitution Law

The substitution law plays a special role in our theory. We will be using it a lot, so it is a good idea to learn how to apply it well. The reason the substitution law is important is that the situation in which it is useful (a sequence of assignments followed by a specification) happens all the time.

The substitution law applies when we have an assignment connected with a specification by dependent composition

$$x := E \ . \ P$$

It does not apply in other cases. It does not apply for example in:

$$x' = E \ . \ P$$

or in

$$x' = E \ \wedge \ P$$

(but try the context rule for the last example).

Whenever we have the correct situation:

$$x := E \ . \ P$$

we can apply the substitution law and replace the whole thing with

substitute x with E in P

The reason why this is important is because it provides a quick way to make two expansions (one for the assignment and one for the dependent composition). Skipping the expansion of the dependent composition is especially important, because this expansion is tedious and error-prone.

Again, if P is abbreviated, e.g. an assignment, we have to remember to expand it first, before applying the substitution law. The following is wrong:

$$\begin{array}{ll} & x := x+1\ .\ ok \quad \text{substitution law } \textbf{WRONG!} \\ = & ok \end{array}$$

The correct use of the substitution law would be to expand *ok* first. So, if we had three variables, say x, a, b, that would be:

$$\begin{array}{ll} & x := x+1\ .\ ok \\ = & x := x+1\ .\ x' = x \wedge a' = a \wedge b' = b \\ = & x' = x+1 \wedge a' = a \wedge b' = b \end{array}$$

Notice also that in the substitution we leave x' alone. It is only x that is being replaced by $x+1$.

A final thing to note about the substitution law is that in a long series of assignments it is better to start *from the end* and work our way to the begining. For example, in the following series of assignments (assume we only have two integer program variables x, y), we apply the substitution law first to the underlined dependent composition:

$$\begin{array}{ll} & x := x+1\ .\ \underline{y := x \times 2\ .\ x' < y} \\ = & x := x+1\ .\ x' < x \times 2 \end{array}$$

The reason is that we can now re-appy the substitution law:

$$= \quad x' < (x+1) \times 2$$

If we had worked from the beginning to the end, our work would be more tedious:

$$\begin{array}{ll} & x := x+1\ .\ y := x \times 2\ .\ x' < y \\ = & x := x+1\ .\ x' = x\ \wedge\ y' = x \times 2\ .\ x' < y \\ = & x' = x+1\ \wedge y' = (x+1) \times 2\ .\ x' < y \end{array}$$

and now the substitution law does not apply any more. We have to expand the dependent composition. The result will be $x' < (x+1) \times 2$ again, but this time we get there the hard way.

Key Points

- In our theory, programs and specifications are seen as boolean expressions and the common programming operators are defined as abbreviations for boolean operators. The assignment, the empty statement and the dependent composition are examples of operators that are defined as abbreviations for logic.

- **Before using boolean theory in our programs, we must expand their programming constructs into their equivalent boolean expressions**.
- **The assignment** $x := E$ **is not equal to** $x' = E$. Their difference is that the assignment also ensures that any programming variables other than x retain their values.
- When expanding a dependent composition $P.Q$, we have to make sure that P and Q **are already expanded into boolean expressions**
- **Dependent composition and conjunction are not the same**. Dependent composition talks about several events that happen in sequence. Conjunction ensures that all its operands are true and does not say anything about sequencing.
- **The Substitution Law is a very useful tool for expanding programs**. It allows us to do two expansions simultaneously.
- The Substitution Law applies when we have **an assignment followed by a specification**, like this: $x := E\ .\ P$. It does **not** apply if there is no assignment, as in $x' = E\ .\ P$ or no dependent composition as in $x' = E\ \wedge\ P$.
- When we have two or more assignments in sequence, it is better to start applying the Substitution Law from right to left.