a Practical Theory of Programming

2025-7-2 edition

Eric C.R. Hehner

a Practical Theory of Programming

2025-7-2 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
hehner.ca/aPToP

An on-line course based on this book is at
hehner.ca/FMSD

The author's website is
hehner.ca

The cover picture is an inukshuk, which is a human-like figure made of piled stones. Inukshuks are found throughout arctic Canada. They are built by the Inuit people, who use them to mean “You are on the right path.”.

Contents

End of Contents

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 using some theory. Similarly a lot of mundane programming can be done without the aid of a theory, but the more difficult programming is 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 names “programming methodology”, “science of programming”, “logic of programming”, “theory of programming”, “formal methods of program development”, “programming from specifications”, 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.

In the first usable theory of programming, a specification is a pair of assertions: a precondition and postcondition (these and all technical terms will be defined in due course). A closely related theory uses the weakest precondition predicate transformer, which is a function from programs and postconditions to preconditions, further advanced in the Refinement Calculus. The Vienna Development Method has been used to advantage in some industries; in it, a specification is a pair, but the second member of the pair is a relation. There are theories that specialize in real-time programming, some in probabilistic programming, some in interactive programming, some in parallel programming.

The theory in this book (originally known as Predicative Programming, now called aPToP) is simpler than any of those just mentioned. In it, a specification is just a binary expression.

Refinement is just implication. This theory is also more comprehensive than those just mentioned, applying to both terminating and nonterminating computation, to both sequential and concurrent computation, to both stand-alone and interactive computation, to both imperative and functional programming, and to probabilistic programming, and it includes time and space bounds. All at the same time, we can have variables whose initial and final values are 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 binary expression whose (nonlocal) variables represent whatever is considered to be of interest.

Model-checking is an approach to program-proving that exhaustively tests all inputs. Its advantage over the theory in this book is that it is fully automated. With a clever representation of binary expressions (see Exercise 15), 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.

An on-line course based on this book is at hehner.ca/FMSD . Solutions to exercises are at hehner.ca/aPToP/solutions .

0.0 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 concurrent processes, there is now a choice: communication channels (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 150 pages; after that is just exercises and reference material.

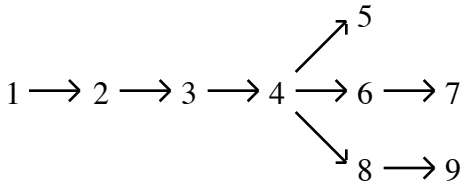
The log of changes to the book is at hehner.ca/aPToP/changelog.pdf .

End of Current Edition

0.1 Quick Tour

All technical terms used in this book are explained in this book. Each new term that you should learn is underlined. The terminology is descriptive rather than honorary. For example, the data type with two values is called “binary” rather than “boolean”; the former word describes the data type; the latter word honors George Boole. 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 binary values, 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 Section 10.4 "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 this material for the first time. If you find yourself asking such questions, don't hesitate to consult the justifications.

Chapter 11 also contains 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

0.2 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/Linklater, Peter Kanareitsev, Yannis Kassios, 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 Introduction

1 Basic Theories

The basic theories we need are Binary Theory, Number Theory, and Character Theory.

1.0 Binary Theory

Binary Theory, also known as boolean algebra, or logic, was designed as an aid to reasoning, and we will use it to reason about computation. The expressions of Binary Theory are called binary expressions. Some binary expressions are classified as theorems, and others as antitheorems.

Binary expressions 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 Binary Theory is perfectly suited is digital circuit design. In that application, binary expressions represent circuits; theorems represent circuits with high voltage output, and antitheorems represent circuits with low voltage output. In general, Binary Theory can be used for any application that has two values.

The two simplest binary expressions are $\top$ and $\bot$. The first one, $\top$, is a theorem, and the second one, $\bot$, is an antitheorem. When Binary Theory is being used for its original purpose, we pronounce $\top$ as "true" and $\bot$ as "false" because the former represents a true statement and the latter represents a false statement. When Binary 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 are the zero-operand binary operators because they have no operands.

There are four one-operand binary 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 value table (also known as a truth table, but truth is just one of many applications of Binary Theory).

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

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 binary 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", "binary addition"

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 called a reverse implication, x is called the consequent, and y is called the antecedent. An expression of the form $x = y$ is called an equation, x is called the left side, y is called 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 value table shows how the classification of binary expressions with two-operand operators can be obtained from the classification of the operands. Above the horizontal line, the pair $\top\top$ means that both operands are theorems; the pair $\top\bot$ means that the left operand is a theorem and the right operand is an antitheorem; and so on. Below the horizontal line, $\top$ means that the result is a theorem, and $\bot$ means that the result is an antitheorem.

	$\top\top$	$\top\bot$	$\bot\top$	$\bot\bot$
$\wedge$	$\top$	$\bot$	$\bot$	$\bot$
$\vee$	$\top$	$\top$	$\top$	$\bot$
$\Rightarrow$	$\top$	$\bot$	$\top$	$\top$
$\Leftarrow$	$\top$	$\top$	$\bot$	$\top$
$=$	$\top$	$\bot$	$\bot$	$\top$
$\neq$	$\bot$	$\top$	$\top$	$\bot$

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 brackets: either $(\bot \wedge \top) \vee \top$ or $\bot \wedge (\top \vee \top)$. To prevent a clutter of brackets, we employ a table of precedence levels (see Section 11.6). In the table, $\wedge$ can be found on level 9, and $\vee$ on level 10; that means, in the absence of brackets, 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 $\mathrel{=\!\!=} \Longrightarrow \Longleftarrow$ 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 bracket clutter still further. But a word of caution: a few well-chosen brackets, 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 **fi**. We call x the **if**-part, we call y the **then**-part, and we call z the **else**-part. Here is its value table.

	$\top\top\top$	$\top\top\bot$	$\top\bot\top$	$\top\bot\bot$	$\bot\top\top$	$\bot\top\bot$	$\bot\bot\top$	$\bot\bot\bot$
if then else fi	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$

For every natural number n, there are 2^{2^n} operators of n operands, but we now have 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 binary 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 brackets 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 brackets. We also replaced a conjunct y with a disjunction $\bot \vee \top$, so we had to enclose the disjunction in brackets.

- 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 binary expressions that make use of the expressions of those theories, and classify the new binary expressions. For example, when we present Number Theory we will introduce the number expressions $1+1$ and 2, and the binary expression $1+1 = 2$, and we will classify it as a theorem. We never intend to classify a binary 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 binary expression both ways, we have made a serious error. But we can leave a binary expression unclassified. For example, $1/0 = 5$ will be neither a theorem nor an antitheorem. An unclassified binary 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 binary expression is both a theorem and an antitheorem, and inconsistent if some binary expression is both a theorem and an antitheorem. A theory is called complete if every fully instantiated binary expression is either a theorem or an antitheorem, and incomplete if some fully instantiated binary 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 how they are classified as theorems and antitheorems. This classification is determined by the six rules of proof. We state the rules first, then discuss them after.

Axiom Rule	If a binary expression is an axiom, then it is a theorem. If a binary expression is an antiaxiom, then it is an antitheorem.
Evaluation Rule	If all the binary subexpressions of a binary expression are classified, then it is classified according to the value tables.
Completion Rule	If a binary expression contains unclassified binary subexpressions, and all ways of classifying them place it in the same class, then it is in that class.
Consistency Rule	If a classified binary expression contains binary subexpressions, and only one way of classifying them is consistent, then they are classified that way.
Transparency Rule	A binary expression does not gain, lose, or change classification when a classified subexpression is replaced by another expression in the same class.
Instance Rule	If a binary expression is classified, then all its instances have that same classification.

An axiom is a binary expression that is stated to be a theorem. An antiaxiom is similarly a binary expression stated to be an antitheorem. The only axiom of Binary 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 six 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 and antiaxioms may yield different theories, and different theories may have different applications. For a theory to be useful, it must be consistent; beyond that, our choice of axioms and antiaxioms is guided by our intended application(s).

The entry in the top left corner of the value table for the two-operand operators does not just 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 binary axiom (to prove that $\top$ is a theorem), the first entry on the $\wedge$ row of the value table (to prove that $\top\wedge\top$ is a theorem), and the first entry on the $=$ row of the value table (to prove that $\top\wedge\top = \top$ is a theorem).

The binary expression

$$\top\vee x$$

contains an unclassified binary subexpression x, 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 binary expression, if it would be inconsistent to place a binary subexpression in one class, then the Consistency Rule says it is in the other class. For example, suppose we know that x is a theorem, and that $x\Rightarrow y$ is also a theorem. Can we determine what class y is in? If y were an antitheorem, then by the Evaluation Rule $x\Rightarrow y$ would be an antitheorem, and that would be inconsistent. So, by the Consistency Rule, y is a theorem. This use of the Consistency Rule is traditionally called "detachment" or "modus ponens". As another example, if $\neg x$ is a theorem, then the Consistency Rule says that x 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 x is an antiaxiom or antitheorem, we can say that $\neg x$ is an axiom or theorem. But a word of caution: if a theory is incomplete, it is possible that neither x nor $\neg x$ 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 binary 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 binary expression, meaning we will prove it is a theorem.

End of Axioms and Proof Rules

We now replace the binary antiaxiom ($\perp$) with an axiom ($\neg\perp$). With our two binary axioms and six 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 Binary 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 $\Leftarrow$ by using the Mirror Law. All of them can be proved 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 proved old laws. In the next three subsections we see how.

1.0.1 Expression and Proof Format

The precedence table (Section 11.6) tells how to parse an expression in the absence of brackets. To help the eye group the symbols properly, it is a good idea to leave space for absent brackets. 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 brackets 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 brackets can be broken at its main connective, which is placed under the opening bracket. For example,

(*first part*
$\wedge$ *second part*)

A long expression without brackets can be broken at its main connective, which is placed under where the opening bracket 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 binary 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 binary expression

expression0 = *expression1*
$\wedge$ *expression1* = *expression2*
$\wedge$ *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

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

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

	$a \wedge b \Rightarrow c$	Material Implication
$=$	$\neg(a \wedge b) \vee c$	Duality
$=$	$\neg a \vee \neg b \vee c$	Material Implication
$=$	$a \Rightarrow \neg b \vee c$	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

$$a \wedge b \Rightarrow c \;=\; \neg(a \wedge b) \vee c$$

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

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

because $a \wedge b$ 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 silently use the Transparency Rule to apply it to part of the expression, replacing $\neg(a \wedge b)$ with $\neg a \vee \neg b$. By not using brackets 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" (and the Transparency Rule) 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.

	$(a \wedge b \Rightarrow c \;=\; a \Rightarrow (b \Rightarrow c))$	Material Implication, 3 times
$=$	$(\neg(a \wedge b) \vee c \;=\; \neg a \vee (\neg b \vee c))$	Duality
$=$	$(\neg a \vee \neg b \vee c \;=\; \neg a \vee \neg b \vee c)$	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

	$(a \Rightarrow b \;=\; a \wedge b) = a$	Associative Law for =
$=$	$(a \Rightarrow b \;=\; (a \wedge b \;=\; a))$	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 binary expression can be anything: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, $=$, $\neq$, or **if then else fi**.

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

$$(a \Rightarrow b) \wedge (c \Rightarrow d) \;\Longrightarrow\; a \wedge c \Rightarrow b \wedge d$$

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

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

From the mutual transitivity of $=$ and $\Leftarrow$, we have proved

$$a \wedge c \Rightarrow b \wedge d \;\Leftarrow\; (a \Rightarrow b) \wedge (c \Rightarrow d)$$

and then the Mirror Law gives the desired theorem.

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

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

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

Here are the monotonic and antimonotonic properties of binary expressions.

- $\neg a$ is antimonotonic in a
- $a \wedge b$ is monotonic in a and monotonic in b
- $a \vee b$ 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 **fi** is monotonic in b and monotonic in c

These properties are useful in proofs. For example, in Exercise 6(k), to prove $\neg(a \wedge \neg(a \vee b))$, we can employ the Law of Generalization $a \Rightarrow a \vee b$ to strengthen $a \vee b$ to a. That weakens $\neg(a \vee b)$ and that weakens $a \wedge \neg(a \vee b)$ and that strengthens $\neg(a \wedge \neg(a \vee b))$.

$$\begin{array}{lll} & \neg(a \wedge \neg(a \vee b)) & \text{use the Law of Generalization} \\ \Leftarrow & \neg(a \wedge \neg a) & \text{now use the Law of Noncontradiction} \\ = & \top & \end{array}$$

We thus prove that $\neg(a \wedge \neg(a \vee b)) \Leftarrow \top$, and by an identity law, that is the same as proving $\neg(a \wedge \neg(a \vee b))$. In other words, $\neg(a \wedge \neg(a \vee b))$ 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 $\Longleftarrow$ signs.

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

	$a \wedge \neg(a \vee b)$	use the Law of Generalization
$\Longrightarrow$	$a \wedge \neg a$	now use the Law of Noncontradiction
$=$	$\bot$	

This is called "proof by contradiction". It proves $a \wedge \neg(a \vee b) \Longrightarrow \bot$, which is the same as proving $\neg(a \wedge \neg(a \vee b))$. 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

Context rules are derived proof rules that can make some proofs easier. In these rules, *exp0* and *exp1* and *exp2* are (possibly lengthy) binary expressions. Here are the context rules.

In $exp0 \wedge exp1$, *exp0* is a local axiom within *exp1*.
In $exp0 \wedge exp1$, *exp1* is a local axiom within *exp0*.
In $exp0 \vee exp1$, *exp0* is a local antiaxiom within *exp1*.
In $exp0 \vee exp1$, *exp1* is a local antiaxiom within *exp0*.
In $exp0 \Longrightarrow exp1$, *exp0* is a local axiom within *exp1*.
In $exp0 \Longrightarrow exp1$, *exp1* is a local antiaxiom within *exp0*.
In $exp0 \Longleftarrow exp1$, *exp0* is a local antiaxiom within *exp1*.
In $exp0 \Longleftarrow exp1$, *exp1* is a local axiom within *exp0*.
In **if** *exp0* **then** *exp1* **else** *exp2* **fi**, *exp0* is a local axiom within *exp1*.
In **if** *exp0* **then** *exp1* **else** *exp2* **fi**, *exp0* is a local antiaxiom within *exp2*.
In **if** *exp0* **then** *exp1* **else** *exp2* **fi**, *exp1=exp2* is a local antiaxiom within *exp0*.

Suppose we have a conjunction.

$exp0 \wedge exp1$

The first context rule says that *exp0* is a local axiom within *exp1*, and therefore any occurrences of *exp0* within *exp1* can be replaced by $\top$ without changing the classification of the whole conjunction. For example, here is another proof of Exercise 6(k) using context.

	$\neg(a \wedge \neg(a \vee b))$	The subexpression $a \wedge \neg(a \vee b)$ is a conjunction. The left conjunct a occurs within the right conjunct $\neg(a \vee b)$, so it can be replaced by $\top$.
$=$	$\neg(a \wedge \neg(\top \vee b))$	Symmetry Law and Base Law for $\vee$
$=$	$\neg(a \wedge \neg\top)$	value table for $\neg$
$=$	$\neg(a \wedge \bot)$	Base Law for $\wedge$
$=$	$\neg\bot$	value table for $\neg$
$=$	$\top$	

Why does this work? If *exp0* happens to be a theorem, then by the Transparency Rule, replacing it with $\top$ within *exp1* doesn't change the conjunction $exp0 \wedge exp1$. If *exp0* happens to be an antitheorem, then the conjunction $exp0 \wedge exp1$ is an antitheorem, and it will remain an antitheorem no matter what happens to *exp1*.

Here is the simplest example of the first context rule.

$a \wedge a$ use left conjunct as context in right conjunct
$= \quad a \wedge \top$

Here is the simplest example of the second context rule.

$a \wedge a$ use right conjunct as context in left conjunct
$= \quad \top \wedge a$

But we cannot use both rules at the same time.

$a \wedge a$ this step is wrong
$= \quad \top \wedge \top$

Suppose we have a disjunction.

exp0 $\vee$ *exp1*

The first context rule for disjunction says that *exp0* is a local antiaxiom within *exp1*, and therefore any occurrences of *exp0* within *exp1* can be replaced by $\bot$ without changing the classification of the whole disjunction. Here is the simplest example.

$a \vee a$ use left disjunct as context in right disjunct
$= \quad a \vee \bot$

Here is another conjunction example.

$x=y \wedge x=z$ use left conjunct as context in right conjunct
$= \quad x=y \wedge y=z$

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

antecedent
$\Rightarrow$ (*expression0*
= *expression1*
= *expression2*
= *expression3*)

for example. The subproof within the consequent can make use of the *antecedent* as a local axiom. The whole proof is proving

antecedent $\Rightarrow$ (*expression0* = *expression3*)

If *expression3* is $\top$, then the whole proof is proving

antecedent $\Rightarrow$ *expression0*

If *expression3* is $\bot$, then the whole proof is proving

antecedent $\Rightarrow$ $\neg$*expression0*

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

if *possibility*
then subproof of *something* using *possibility* as a local axiom
else subproof of *anotherthing* using $\neg$*possibility* as a local axiom **fi**

The whole proof proves

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

If both subproofs prove the same thing,

if *possibility*
then subproof of *something* using *possibility* as a local axiom
else subproof of *something* using $\neg$*possibility* as a local axiom **fi**

then by the Case Idempotent Law, the whole proof proves *something*, and that is its most frequent use.

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 descriptions.

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 binary 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”, “although”, and “moreover” might all be translated as $\wedge$. Just putting things next to each other sometimes means $\wedge$. For example, “They're red, ripe, and juicy, but not sweet.” becomes

$$red \wedge ripe \wedge juicy \wedge \neg sweet$$

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

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 Binary 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, optionally including a decimal point	
∞	“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”
$x \uparrow y$	“x max y”
$x \downarrow y$	“x min y”
if a **then** x **else** y **fi**	“if a then x else y”

where x and y are any number expressions, and a is any binary 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 binary 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 laws 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 laws

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

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

End of Number Theory

1.2 Character Theory

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

End of Character Theory

All our theories use the operators $=$ $\neq$ **if then else fi** , 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 Binary Theory; for example, $x{=}x$ can be proved using the Completion and Evaluation rules. But in Number Theory and other theories, they are axioms; without them we cannot even prove $5{=}5$ or "a"="a" ; with them, we can.

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

End of Basic Theories

We have talked about binary 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 binary, 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 binary values, 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 binary (and later set, string of elements, list of elements) is an elementary bunch, or synonymously an element. For example, the number 2 is an elementary bunch, or element. Every expression is a bunch expression, though not all are elementary. Therefore the word "bunch" is usually superfluous. For example, we might say we have the bunch of numbers $0, 2, 5, 9$. Or we might just say we have the numbers $0, 2, 5, 9$.

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 binaries

$A: B$	"A is in B", "A is included in B"
$A:: B$	"A includes 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
$x: A, B = x: A \vee x: B$	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
$A, (B`C) = (A, B)`(A, C)$	distributivity
$A`(B, C) = (A`B), (A`C)$	distributivity
$A: B`C = A: B \wedge A: C$	distributivity
$A, B: C = A: C \wedge B: C$	antidistributivity
$A: A, B$	generalization
$A`B: A$	specialization
$A: A$	reflexivity
$A: B \wedge B: A = A{=}B$	antisymmetry
$A: B \wedge B: C \Rightarrow A: C$	transitivity
$A:: B = B: A$	mirror
$\not\in x = 1$	size
$\not\in(A, B) + \not\in(A`B) = \not\in A + \not\in B$	size
$\neg\, x: A = \not\in(A`x) = 0$	size
$A: B \Rightarrow \not\in A \le \not\in B$	size

From these axioms, many laws can be proved. 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
$A: B \wedge C: D \Rightarrow A, C: B, D$	conflation
$A: B \wedge C: D \Rightarrow A`C: B`D$	conflation

Here are several bunches that we will find useful:

null			the empty bunch
bin	=	$\top, \bot$	the binary values
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 natural numbers
xint	=	$-\infty, ..., -2, -1, 0, 1, 2, ..., \infty$	the extended integer numbers
xrat	=	$-\infty, ..., -1, 0, 2/3, ..., \infty$	the extended rational numbers
xreal	=	$-\infty, ..., 2^{1/2}, ..., \infty$	the extended real numbers
char	=	..., “a”, “A”, ...	the character values

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 , ` $\not\in$: :: = ≠ **if then else fi** apply to bunch operands according to the laws 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,

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

This makes it easy to express the plural naturals $(nat+2)$, the even naturals $(nat\times 2)$, the square naturals (nat^2), the natural powers of two (2^{nat}), and many other things. (The operators that distribute over bunch union are listed in Section 11.7.)

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

$null: A$
$\not\in A = 0 \quad = \quad A = null$

This gives us three more laws:

$A, null = A$	identity
$A`null = null$	base
$\not\in null = 0$	size

The bunch *bin* is defined by the axiom

$bin = \top, \bot$

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 Section 3.4. 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* and *y* are extended integers and $x \le y$. Its axiom is

$i: x,..y \quad = \quad i: xint \wedge x \le i < y$

The notation ,.. is asymmetric as a reminder, roughly speaking, that the left end of the interval is included and the right end is excluded. For example,

$0,..3 = 0, 1, 2$
$0,..\infty = nat$
$5,..5 = null$
$x,..x+1 = x$
$\not\in(x,..y) = y-x$

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 $\{0, 1, 2\}$ or $\{0,..3\}$. 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 $\natural$ applies to a bunch and yields all sets that contain only elements of the bunch. Here is an example.

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

We "promote" the bunch operators to obtain the set operators $\$ \in \subseteq \natural \cup \cap = \neq$. Here are the axioms. Let S be a set, and let A and B be anything.

$\{A\} \neq A$	structure
$\{\sim S\} = S$	set formation
$\sim\{A\} = A$	"contents"
$\$\{A\} = \notin A$	"size", "cardinality"
$A \in \{B\} = A : B$	"elements"
$\{A\} \subseteq \{B\} = A : B$	"subset"
$\{A\} : \natural B = A : B$	"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

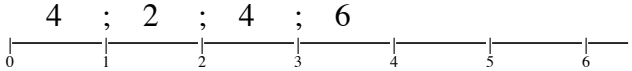
Any number, character, binary, set, (and later also list and function) is a one-item string, or item. For example, the number 2 is a one-item string, or item. A nonempty bunch of items is also an item. Strings are joined together by semicolons to make longer strings. For example,

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 at kilometer 0, a weigh scale at kilogram 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 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 select 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 joined together.

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

Without a left operand, $*S$ means all strings formed by joining 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 $S◅n▻i$ is a string like S except that the item at index n is i. For example,

$$3; 5; 9◅2▻8 = 3; 5; 8$$

Strings can be compared for equality $= \neq$. To be equal, strings must be of equal length, and have equal items at each index. If the items of the string can be compared for order $< \leq > \geq$, then so can the strings. 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 because 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 join T"
S_T	"S sub T"
$n*S$	"n copies of S"
$S◅n▻i$	"S but at n is i"

are strings,

$*S$	"copies of S"

is a bunch of strings, and

$↔S$	"length of S"

is an extended natural number.

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.

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

We also use the notation (note the semicolon)

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

where x and y are extended integers 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 text notation is an alternative way of writing a string of characters. A text begins with a left 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 right 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).

""""$_{0;0;(0;..29);28;28;(1;..28)}$""""$_{0;0;(0;..29);28;28;(1;..28)}$

Perform the indexing and see what you get.

String join distributes over bunch union:

$A; null; B = null$
$(A, B); (C, D) = A;C, A;D, B;C, B;D$
$¢ nil = 1$
$S;A;T : S;B;T = A: B$
$¢(A; B) \le ¢A \times ¢B$

So a string of bunches is equal to a bunch of strings. 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 both *nil* and 0;1;2 are elements, but *nat*; 1; (0,..10) is not.

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*

Both one- and two-operand * can apply to a bunch of strings in the right operand,

2*(0, 1) = (0, 1); (0, 1) = 0;0 , 0;1 , 1;0 , 1;1

but * does not distribute over bunch union in the right operand.

2*(0, 1) ≠ 2*0, 2*1 = 0;0 , 1;1

End of String Theory

2.3 List Theory

A list is a packaged string. For example,

[0; 1; 2]

is a list containing three items. Although the string 0; 1; 2 is not a single item, the list [0; 1; 2] is a single item. A list of elements is an element. 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. 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* join *M*"
n→*i* | *L*	"*n* maps to *i* otherwise *L*"

are lists,

~*L*	"contents of *L*"

is a string,

#*L*	"length of *L*"

is an extended natural number,

□*L*	"domain of *L*"

is a bunch of natural numbers, and

L n	"*L n*" or "*L* at index *n*"

is an item. Of course, brackets may be used around any expression, so we may write *L*(*n*) if we want. Brackets must be used when required by the precedence rules; for example, *L*(*n*+1) . But we cannot write *Ln* without a space between *L* and *n* because that would be a single multicharacter name.

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 placing its

index immediately after the list.

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

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 join is written with two semi-colons.

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

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

$2{\rightarrow}22 \mid [10;..15] = [10; 11; 22; 13; 14]$

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

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

$2{\rightarrow}L\ 3 \mid 3{\rightarrow}L\ 2 \mid L = [10; 11; 13; 12; 14]$

Lists can be compared for equality $= \neq$. To be equal, lists must be of equal length, and have equal items at each index. If the items of the list can be compared for order $< \leq > \geq$, then so can the lists; the order is lexicographic, just like string order.

Here are the axioms. Let L and M be a lists, let S and T be strings, let n be an index of S, let i be an item, and let A and B be bunches of strings.

$[S] \neq S$	structure
$[{\sim}L] = L$	list formation
${\sim}[S] = S$	contents
$\#[S] = {\leftrightarrow}S$	length
$\square L = 0,..\#L$	domain
$[S];;[T] = [S; T]$	join
$[S]\ n = S_n$	indexing
$[S]\ [T] = [S_T]$	composition
$n{\rightarrow}i \mid [S] = [S{\triangleleft}n{\triangleright}i]$	modification
$[S] = [T] \quad = \quad S = T$	equation
$[S] < [T] \quad = \quad S < T$	order
$[A]: [B] \quad = \quad A: B$	inclusion

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

$(L\ M)\ n = L\ (M\ n)$	composition
$(L\ M)\ N = L\ (M\ N)$	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 has 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

$$\begin{aligned} & S_{0,\,\{1,\,[2;\,1];\,0\}} \\ = \quad & S_0, \{S_1, [S_2; S_1]; S_0\} \\ = \quad & 10, \{11, [12; 11]; 10\} \end{aligned}$$

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

$$\begin{aligned} & L\,(0, \{1, [2; 1]; 0\}) \\ = \quad & L\,0, \{L\,1, [L\,2; L\,1]; L\,0\} \\ = \quad & 10, \{11, [12; 11]; 10\} \end{aligned}$$

2.3.0 Multidimensional Structures

A list is sometimes called an array, especially if it is multidimensional. For example, let

$$\begin{aligned} A \;=\; & [\;\; [6; 3; 7; 0]\;; \\ & \;\;\; [4; 9; 2; 5]\;; \\ & \;\;\; [1; 5; 8; 3]\;\;] \end{aligned}$$

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 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, 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 $S{\rightarrow}i \mid L$ to allow S to be a string of indexes. The axioms are

$$nil{\rightarrow}i \mid L \;=\; i$$

$$(S; T){\rightarrow}i \mid L \;=\; S{\rightarrow}(T{\rightarrow}i \mid L@S) \mid L$$

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

$$\begin{aligned} & (0;1){\rightarrow}6 \mid [\;[0; 1; 2]\;; \\ & \qquad\qquad [3; 4; 5]\;\;] \\ = \quad & [\;\; [0; 6; 2]\;; \\ & \;\;\; [3; 4; 5]\;\;] \end{aligned}$$

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 $x\times 1 = x$ 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 some axioms, and prove $3.14 < pi < 3.15$, 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

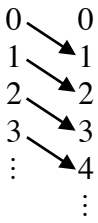
Let v be a name, let D be any expression (possibly using previously introduced names but not using v), and let b be any expression (possibly using previously introduced names and possibly using v). Then

$\langle v{:}\ D{\cdot}\ b\rangle$ "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,

$\langle n{:}\ nat{\cdot}\ n{+}1\rangle$

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



If f is a function, then $\Box f$ is the domain of f. The Domain Axiom is

$\Box\langle v{:}\ D{\cdot}\ b\rangle \ =\ D$

We say both that D is the domain of function $\langle v{:}\ D{\cdot}\ b\rangle$ and that within the body b, D is the domain of variable v. The size of function f is $\#f$, defined as

$\#f \ =\ ¢\Box f$

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

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

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, brackets may be used around any expression, so we may write $f\,(x)$ if we want. Brackets must be used when required by the precedence rules; for example, $f\,(x{+}1)$. But we cannot write fx without a space between f and x because that would be a single multicharacter name. As an example of application, if $suc = \langle n{:}\ nat\cdot\ n{+}1\rangle$, then

$$suc\ 3\ =\ \langle n{:}\ nat\cdot\ n{+}1\rangle\ 3\ =\ 3{+}1\ =\ 4$$

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

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

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. For example,

$$avg\ =\ \langle x{:}\ rat\cdot\ \langle y{:}\ rat\cdot\ (x{+}y)/2\rangle\rangle$$

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

$$avg\ 3\ =\ \langle y{:}\ rat\cdot\ (3{+}y)/2\rangle$$

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

$$avg\ 3\ 5\ =\ (3{+}5)/2\ =\ 4$$

This is not the same as $avg\ (3,\ 5)$, which applies avg to the bunch of arguments $(3,\ 5)$, and results in the bunch of one-parameter functions $(avg\ 3,\ avg\ 5)$.

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

$$even\ =\ \langle i{:}\ int\cdot\ i/2{:}\ int\rangle$$
$$odd\ =\ \langle i{:}\ int\cdot\ \neg\ i/2{:}\ int\rangle$$

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

$$divides\ =\ \langle n{:}\ nat{+}1\cdot\ \langle i{:}\ int\cdot\ i/n{:}\ int\rangle\rangle$$
$$divides\ 2\ =\ even$$
$$divides\ 2\ 3\ =\ \bot$$

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

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

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, as an abbreviation for the *avg* function seen earlier:

$\langle x, y\colon rat\cdot (x+y)/2\rangle$

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 $\langle n\cdot n+1\rangle$ in a context where it is understood that the domain is *nat* .

We may omit the variable when the body of a function does not use it. In this case, we write only the domain and body, with an arrow $\rightarrow$ between. For example, $2\rightarrow 3$ is a function that maps 2 to 3 , which we could have written $\langle n\colon 2\cdot 3\rangle$ 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,

$\langle x\colon int\cdot \langle y\colon int\cdot x+3\rangle\rangle$

which introduces two variables, would have the same abbreviation as

$\langle x\colon int\cdot x+3\rangle$

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 local to an expression if its introduction is inside the expression (and therefore formal). A variable is nonlocal 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.

$\langle x\cdot \quad x \quad \langle x\cdot \quad x \quad \rangle \quad x \quad \rangle 3$

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 . If 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

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

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

$$\langle y\cdot \quad x \quad y \quad \langle x\cdot \quad x \quad y \quad \rangle \quad x \quad y \quad \rangle x$$

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

$$= \quad \langle y\cdot \quad x \quad y \quad \langle z\cdot \quad z \quad y \quad \rangle \quad x \quad y \quad \rangle x$$

by renaming, and then substitution gives

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

The Application Axiom (for element x: D)

$\langle v{:}\ D\cdot\ b\rangle\ x\ =$ (substitute x for v 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 Subsection 5.0.1, 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:

$\langle v{:}\ D\cdot\ b\rangle\ =\ \langle w{:}\ D\cdot\ \langle v{:}\ D\cdot\ b\rangle w\rangle$

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

$f\ =\ \langle w{:}\ \square f\cdot\ f\ w\rangle$

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

$\langle v\cdot\ b\rangle\ x\ =\ \langle w\cdot\ \langle v\cdot\ b\rangle\ w\rangle\ x$

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 six examples: the operators $\wedge\ \vee\ +\ \times\ \uparrow\ \downarrow$ are used to define, respectively, the quantifiers $\forall\ \exists\ \Sigma\ \Pi\ \Uparrow\ \Downarrow$. Here are the informal intuitions (the formal definitions (axioms) will follow). If p is a predicate, then universal quantification $\forall p$ is the binary result of applying p to all its domain elements and conjoining all the results. Similarly, existential quantification $\exists p$ is the binary 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; Πf is the numeric result of applying f to all its domain elements and multiplying together all the results; $\Uparrow f$ is the numeric result of applying f to all its domain elements and finding the maximum (or least upper bound) of all the results; and $\Downarrow f$ is the numeric result of applying f to all its domain elements and finding the minimum (or greatest lower bound) of all the results. Here are six examples.

$\forall\langle r{:}\ rat\cdot\ r{\geq}0\rangle$	$=\ \bot$	"for all r in rat ..."
$\exists\langle n{:}\ nat\cdot\ n{=}0\rangle$	$=\ \top$	"there exists n in nat such that ..."
$\Sigma\langle n{:}\ nat{+}1\cdot\ 1/2^n\rangle$	$=\ 1$	"the sum, for n in $nat{+}1$, of ..."
$\Pi\langle n{:}\ nat{+}1\cdot\ n/(n{+}1)\rangle$	$=\ 0$	"the product, for n in $nat{+}1$, of ..."
$\Uparrow\langle x{:}\ rat\cdot\ x{\times}(4{-}x)\rangle$	$=\ 4$	"the maximum, for x in rat, of ..."
$\Downarrow\langle n{:}\ nat{+}1\cdot\ 1/n\rangle$	$=\ 0$	"the minimum, 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. For example we write

$\forall r\colon rat\cdot\ r{\geq}0$ abbreviates $\forall\langle r\colon rat\cdot\ r{\geq}0\rangle$

$\Sigma n\colon nat{+}1\cdot\ 1/2^n$ abbreviates $\Sigma\langle n\colon nat{+}1\cdot\ 1/2^n\rangle$

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

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

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 $\neg\forall lost$.

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 binary expression, n is a number expression, and x is an element).

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

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

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

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

$\Downarrow v\colon null\cdot\ n\ =\ \infty$
$\Downarrow v\colon x\cdot\ n\ =\ \langle v\colon x\cdot\ n\rangle\ x$
$\Downarrow v\colon A, B\cdot\ n\ =\ (\Downarrow v\colon A\cdot\ n)\ \downarrow\ (\Downarrow v\colon B\cdot\ n)$

$\Uparrow v\colon null\cdot\ n\ =\ {-}\infty$
$\Uparrow v\colon x\cdot\ n\ =\ \langle v\colon x\cdot\ n\rangle\ x$
$\Uparrow v\colon A, B\cdot\ n\ =\ (\Uparrow v\colon A\cdot\ n)\ \uparrow\ (\Uparrow v\colon B\cdot\ n)$

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.

Our final quantifier applies to predicates. The solution quantifier $\S$, which we pronounce "solution(s) of" or "those" (plural) or "that" (singular), gives the bunch of solutions of a predicate. For example, $\S\langle i: int\cdot i^2 = 4\rangle$, which we read as "those i in int such that i squared equals four", is the bunch $2, -2$. Equations are just a special case of binary expression; we can just as well talk about the solutions of any predicate. For example, omitting the scope brackets,

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

Here are the axioms.

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

By tradition, when the solution quantifier is used within a set, we can abbreviate by omitting the quantifier. For example, instead of writing $\{\S\langle n: nat\cdot n<3\rangle\}$, 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 $\S$ 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. The Specialization and Generalization laws at the back of the book say that if p is a predicate and element $x: \Box p$,

$$\forall p \;\Rightarrow\; p\, x \;\Rightarrow\; \exists p$$

If p results in $\top$ for all its domain elements, then p results in $\top$ for domain element x. And if p results in $\top$ for domain element x, then there is a domain element for which p results in $\top$. Similarly, if f is a numeric function and element $x: \Box f$

$$\Downarrow f \le f\, x \le \Uparrow f$$

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

$$\forall v: D\cdot v{=}x \Rightarrow b \;=\; \langle v: D\cdot b\rangle\, x \;=\; (\text{substitute } x \text{ for } v \text{ in } b)$$
$$\exists v: D\cdot v{=}x \wedge b \;=\; \langle v: D\cdot b\rangle\, x \;=\; (\text{substitute } x \text{ for } v \text{ in } b)$$

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:

$$\forall x: \Box p\cdot x{=}y \Rightarrow p\, x$$
$$= \quad \exists x: \Box p\cdot x{=}y \wedge p\, x$$
$$= \quad \langle x: \Box p\cdot p\, x\rangle\, y$$
$$= \quad p\, y$$

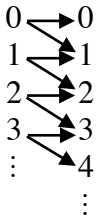
End of Quantifiers

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,

$\langle n\colon nat\cdot n, n{+}1\rangle$

maps each natural number to two natural numbers.



Application works as usual:

$\langle n\colon nat\cdot n, n{+}1\rangle\ 3 \ = \ 3, 4$

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 $\langle n\colon nat\cdot 0,..n\rangle$ is both partial and nondeterministic.

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

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

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

$f\ null \ = \ null$

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

$f\ (\S g) \ = \ \S y\colon f\ (\square g)\cdot \exists x\colon \square g\cdot f\ x = y \ \wedge \ g\ x$

So function application distributes over bunch union. The range of function f is $f\ (\square f)$.

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

$double\ (2, 3)$ this step is right

$= \ double\ 2, double\ 3$

$= \ 4, 6$

but we cannot say

$double\ (2, 3)$ 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 at most 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,

$\langle n\colon nat\cdot n{\times}2\rangle\ (2, 3)$ this step is not really right, but it is harmless

$= \ (2, 3){\times}2$

$= \ 4, 6$

3.2.0 Function Inclusion and Equality optional

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

$$f{:}\ g \quad = \quad \Box f{::}\ \Box g \ \wedge\ \forall x{:}\ \Box g{\cdot}\ f\,x{:}\ g\,x$$

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

$$f = g \quad = \quad \Box f = \Box g \ \wedge\ \forall x{:}\ \Box f{\cdot}\ f\,x = g\,x$$

We now prove $suc{:}\ nat{\to}nat$. Function suc was defined earlier as $suc \ = \ \langle n{:}\ nat{\cdot}\ n{+}1\rangle$. Function $nat{\to}nat$ is an abbreviation of $\langle n{:}\ nat{\cdot}\ nat\rangle$, 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 .

$$\begin{array}{lll} & suc{:}\ nat{\to}nat & \text{use Function Inclusion Law} \\ = & \Box suc{::}\ nat \ \wedge\ \forall n{:}\ nat{\cdot}\ suc\ n{:}\ nat & \text{definition of } suc \text{ and } \Box \\ = & nat{::}\ nat \ \wedge\ \forall n{:}\ nat{\cdot}\ n{+}1{:}\ nat & \text{reflexivity, and } nat \text{ construction axiom} \\ = & \top & \end{array}$$

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

$$avg{:}\ rat{\to}rat{\to}rat$$
$$even{:}\ int{\to}bin$$
$$odd{:}\ int{\to}bin$$
$$divides{:}\ (nat{+}1){\to}int{\to}bin$$

And, more generally,

$$f{:}\ A{\to}B \quad = \quad \Box f{::}\ A \ \wedge\ f\,A{:}\ B$$

End of Function Inclusion and Equality

The definition of suc

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

is equivalent to

$$\Box suc = nat \quad \wedge \quad \forall n{:}\ nat{\cdot}\ suc\ n = n{+}1$$

We could have defined suc by the weaker axiom

$$\Box suc{::}\ nat \quad \wedge \quad \forall n{:}\ nat{\cdot}\ suc\ n = n{+}1$$

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 avg , $even$, 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. For example, define predicate $check$ as follows.

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

So $check$ applies to any function whose domain includes $0,..10$ ($check$ will be applying its argument to all elements in $0,..10$), and when applied to any element in $0,..10$ has a result in int (the result 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. Since

$$suc{:}\ nat{\to}nat{:}\ (0{,}..10){\to}int$$

we can apply $check$ to suc and the result is $\bot$.

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. Function application distributes over bunch union, so applying $check$ to suc applies $check$ to all functions included in suc . All functions included in suc have domains that include $0,..10$, and when applied to elements

of $0,..10$ they produce the same result as suc. So, in effect, we can apply $check$ to suc.

If we need a higher-order function for which application and distribution do not give the same result, we can package the parameter and argument as a set.

$\langle f{:}\, \natural(A{\to}B){\cdot}\ ...\ {\sim}f\ ...\ \rangle\ \{g\}$

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

End of Higher-Order Functions

3.2.2 Function Composition optional

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

$\Box(f\,g)\ =\ \S x{:}\ \Box g{\cdot}\ g\,x{:}\ \Box f$

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

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

$\Box(even\ suc)\ =\ \S x{:}\ \Box suc{\cdot}\ suc\,x{:}\ \Box even\ =\ \S x{:}\ nat{\cdot}\ x{+}1{:}\ int\ =\ nat$

$(even\ suc)\ 3\ =\ even\ (suc\ 3)\ =\ even\ 4\ =\ \top$

Suppose $x, y{:}\ int$ and $f, g{:}\ int{\to}int$ and $h{:}\ int{\to}int{\to}int$. Then

	$h\,f\,x\,g\,y$	adjacency 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 brackets
$=$	$h\,(f\,x)\,(g\,y)$	

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

Like application, composition distributes over bunch union.

$f\,(g, h)\ =\ f\,g, f\,h$

$(f, g)\,h\ =\ f\,h, g\,h$

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 suc to obtain a new function.

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

Similarly we can compose $\neg$ with $even$ to obtain odd.

$\neg even\ =\ odd$

We can write the Duality Laws this way:

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

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

$-\Downarrow f\ =\ \Uparrow -f$

$-\Uparrow f\ =\ \Downarrow -f$

End of Function Composition

End of Function Fine Points

3.3 List as Function

The list L has much in common with the function $\langle n{:}\ \square L{\cdot}\ L\ n\rangle$. List indexing is function application:

$$L\,m \;=\; \langle n{:}\ \square L{\cdot}\ L\ n\rangle\ m$$

List composition coincides with function composition:

$$L\,M\,m \;=\; \langle n{:}\ \square L{\cdot}\ L\ n\rangle\ \langle n{:}\ \square M{\cdot}\ M\ n\rangle\ m$$

List domain and size are function domain and size:

$$\square L \;=\; \square\langle n{:}\ \square L{\cdot}\ L\ n\rangle$$
$$\#L \;=\; \#\langle n{:}\ \square L{\cdot}\ L\ n\rangle$$

List equality is function equality:

$$L{=}M \;=\; \langle n{:}\ \square L{\cdot}\ L\ n\rangle{=}\langle n{:}\ \square M{\cdot}\ M\ n\rangle$$

Operators and functions can be composed with lists, just as they can be composed with functions. 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 $1{\to}21$ as left operand, and list $[10;\ 11;\ 12]$ as right operand, we get

$$1{\to}21\ |\ [10;\ 11;\ 12] \;=\; [10;\ 21;\ 12] \;=\; 0{\to}10\ |\ 1{\to}21\ |\ 2{\to}12$$

Functions can be applied to all types of elements, including sets, strings of elements, and lists of elements. Functions can also be applied to non-elementary bunches by distributing over bunch union. More generally, if the argument of a function is in its domain, then the function applies to the argument according to the Application Law. But if the argument to a function is a structure not in its domain, then application distributes over the structure. The distribution laws for functions are the same as the distribution laws for lists. Let f be a function. Then

$$f\,null \;=\; null$$
$$f\,(A, B) \;=\; f\,A, f\,B$$
$$f\,nil \;=\; nil$$
$$f\,(S;\ T) \;=\; f\,S;\ f\,T$$
$$f\,\{A\} \;=\; \{f\,A\}$$
$$f\,[S] \;=\; [f\,S]$$

For example,

$$\begin{aligned} & suc\ (0, \{1, [2;\ 1];\ 0\}) \\ = \quad & suc\ 0, \{suc\ 1, [suc\ 2;\ suc\ 1];\ suc\ 0\} \\ = \quad & 1, \{2, [3;\ 2];\ 1\} \end{aligned}$$

Quantifiers can be applied to lists. For example, ΣL gives the same result as $\Sigma n{:}\ \square L{\cdot}\ L\ n$. Thus ΣL conveniently expresses the sum of the items of the list.

In some respects, lists and functions differ. The join operator $L;;M$ applies to lists, but not to functions. The $@$ operator applies to lists, but not to functions. A list whose items are elements is an element; functions are not elements. List inclusion $L{:}\ M$ does not coincide with function inclusion $\langle n{:}\ \square L{\cdot}\ L\ n\rangle{:}\ \langle n{:}\ \square M{\cdot}\ M\ n\rangle$.

End of List as Function

3.4 Limits and Reals optional

Let f: $nat \to rat$ so that $f\,0; f\,1; f\,2;$ ... is a sequence of rationals. The limit of the function (limit of the sequence) is expressed as $⇳f$. For example,

$$⇳\langle n{:}\ nat{\cdot}\ (1 + 1/n)^n\rangle$$

or, with the usual abbreviation, and the understanding that the domain is nat,

$$⇳n{\cdot}\ (1 + 1/n)^n$$

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

$$(⇑m{\cdot}\ ⇓n{\cdot}\ f\,(m+n)) \le ⇳f \le (⇓m{\cdot}\ ⇑n{\cdot}\ f\,(m+n))$$

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

$$⇳n{\cdot}\ 1/(n+1) = 0$$

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

$$-1 \le (⇳n{\cdot}\ (-1)^n) \le 1$$

In general,

$$⇓f \le ⇳f \le ⇑f$$

For monotonic (nondecreasing) f, $⇳f = ⇑f$. For antimonotonic (nonincreasing) f, $⇳f = ⇓f$.

We define the extended real numbers as the limits of all functions with domain at least nat and range at most rat. And we define the reals as the extended reals without ∞ and $-\infty$.

$$x{:}\ xreal = ∃f{:}\ nat \to rat{\cdot}\ x = ⇳f$$
$$r{:}\ real = r{:}\ xreal \wedge -\infty < r < \infty$$

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

Let p: $nat \to bin$ so that p is a predicate and $p\,0; p\,1; p\,2;$ ... is a sequence of binary expressions. The limit of predicate p is defined by the axiom

$$∃m{\cdot}\ ∀n{\cdot}\ p\,(m+n) \implies ⇳p \implies ∀m{\cdot}\ ∃n{\cdot}\ p\,(m+n)$$

with all domains being nat. The limit axiom for predicates is 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.

$$∃m{\cdot}\ ∀i{\cdot}\ i \ge m \Rightarrow p\,i \implies ⇳p$$
$$∃m{\cdot}\ ∀i{\cdot}\ i \ge m \Rightarrow \neg\, p\,i \implies \neg\, ⇳p$$

For any particular assignment of values to (nonlocal) variables, the first implication says that $⇳p$ is $\top$ if there is an index m in the sequence $p\,0; p\,1; p\,2;$ ... past which $p\,i$ is always $\top$, and the second implication says that $⇳p$ is $\bot$ if there is an index m in the sequence past which $p\,i$ is always $\bot$. For example,

$$\neg\, ⇳n{\cdot}\ 1/(n+1) = 0$$

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 binary value, then the axiom does not determine the value of $⇳p$ for that assignment of values.

End of Limits and Reals

VERY IMPORTANT POINT: Any expression talks about its nonlocal variables. For example,

$$∃n{:}\ nat{\cdot}\ x = 2{\times}n$$

says that x is an even natural. The local variable n, which could just as well have been m or any other name except x, 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 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

σ = –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 Section 4.2 where we discuss execution time.

4.0 Specifications

A specification is a binary 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 computer then computes and delivers a poststate as output. To satisfy a specification, a computer must deliver a satisfactory poststate. In other words, the given prestate and the computed poststate must make the specification $\top$. 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: $\not\in(\S\sigma'\cdot S) < 1$

Specification S is satisfiable for prestate σ: $\not\in(\S\sigma'\cdot S) \geq 1$

Specification S is deterministic for prestate σ: $\not\in(\S\sigma'\cdot S) \leq 1$

Specification S is nondeterministic for prestate σ: $\not\in(\S\sigma'\cdot S) > 1$

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

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

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

$\forall x, y\cdot \exists x', y'\cdot x' = x+1 \ \wedge \ y'=y$ One-Point Law twice

$= \quad \forall x, y\cdot \top$ Identity Law twice

$= \quad \top$

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

In the same state variables, here is another implementable 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 $x' = x+1 \ \wedge \ y'=y$ 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 impossible to implement because it is not satisfied by any computer behavior. But $\bot$ is not the only unimplementable specification. Here is another.

$x\geq 0 \ \wedge \ y'=0$

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 value for x; in that case, the specifier should have written

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

For nonnegative initial x, this specification still requires variable y to be assigned 0. If we never provide a negative value for x then it doesn't matter what happens when x is initially negative. 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 $x\geq 0$. We can describe the acceptable inputs and the computer behavior together as $x\geq 0 \wedge (x\geq 0 \Rightarrow y'=0)$, which can be simplified to $x\geq 0 \wedge y'=0$. But $x\geq 0 \wedge y'=0$ 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" or "program variable", and a computing constant is also known as a "state constant" or "program 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.

$$\begin{aligned} ok &= \sigma'=\sigma \\ &= x'=x \ \wedge \ y'=y \ \wedge \ ... \end{aligned}$$

$$\begin{aligned} x:= e &= \sigma' \ = \ \sigma \triangleleft address\ \text{“}x\text{”} \triangleright e \\ &= x'=e \ \wedge \ y'=y \ \wedge \ ... \end{aligned}$$

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,

$$x:= x+y \ = \ x'=x+y \ \wedge \ y'=y$$

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 binary expressions, and they can be combined using any operators of Binary Theory. If S and R are specifications, then $S \wedge R$ is a specification that is satisfied by any computation that satisfies both S and R. Similarly, $S \vee R$ 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 **fi** where b is a binary 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 $\vee$ and **if then else fi** operators have the nice property that if their operands are implementable, so is the result; the operators $\wedge$ and $\neg$ do not have that property.

Specifications can also be combined by sequential composition, 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 sequential composition is pronounced "dot". This is not the same as the raised dot used in the function notation.) Sequential composition is defined as follows.

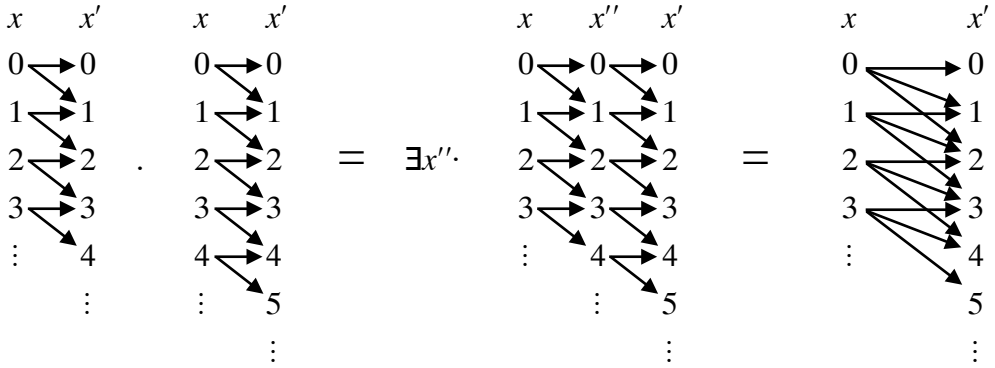
$$\begin{aligned} S.R \ &= \ \exists \sigma'' \cdot \ \langle \sigma' \cdot S \rangle \ \sigma'' \ \wedge \ \langle \sigma \cdot R \rangle \ \sigma'' \\ &= \ \exists x'', y'', ... \cdot \ \langle x', y', ... \cdot S \rangle \ x'' \ y'' \ ... \ \wedge \ \langle x, y, ... \cdot R \rangle \ x'' \ y'' \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

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

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

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 leave it alone, or 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 sequential 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 about what is meant by (substitute $x'', y'', \ldots$ for $x', y', \ldots$ in S) and (substitute $x'', y'', \ldots$ for $x, y, \ldots$ 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 are not 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 sequential composition, the inner substitutions must be made first. Here is an example, again in integer variables x and y.

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

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 binary b,

$$ok.P \;=\; P \;=\; P.ok \qquad \text{Identity Law}$$

$$P.(Q.R) \;=\; (P.Q).R \qquad \text{Associative Law}$$

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

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

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

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

x:= $e.\ P$ $=$ $\langle x\cdot P \rangle\ e$ Substitution Law
$=$ (substitute e for x in P)

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 121 illustrates all the difficult cases, so let us do the exercise. The state variables are integer 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 brackets around y+1 when we substitute it for x in $y' = 2\times x$.
$=$ $y' = 2\times(y+1)$

(d) x:= 1. $x \geq 1 \ \Rightarrow \ \exists x\cdot\ y' = 2\times x$
In $x \geq 1 \ \Rightarrow \ \exists x\cdot\ y' = 2\times x$, 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.
$=$ $1 \geq 1 \ \Rightarrow \ \exists x\cdot\ y' = 2\times x$
$=$ *even* y'

(e) x:= y. $x \geq 1 \ \Rightarrow \ \exists y\cdot\ y' = x\times y$
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. $x \geq 1 \ \Rightarrow \ \exists k\cdot\ y' = x\times k$
$=$ $y \geq 1 \ \Rightarrow \ \exists k\cdot\ y' = y\times k$

(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.

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

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

(It is questionable whether $x'=1 \wedge y'=2$ 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).

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

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

In part (g) we saw that $x:= 1.\ y:= 2 = x'=1 \wedge y'=2$. We can use that,

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

but now we are faced with a sequential composition for which the Substitution Law does not apply. We can proceed by expanding the final assignment

$= \quad x'=1 \wedge y'=2.\ x'=x+y \wedge y'=y$

and then applying the definition of sequential composition

$= \quad \exists x'', y''\cdot x''=1 \wedge y''=2 \wedge x'=x''+y'' \wedge y'=y''$

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

But there is an easier way: in a sequence of assignments, use the Substitution Law from right to left.

$x:= 1.\ y:= 2.\ x:= x+y$ expand final assignment

$= \quad x:= 1.\ y:= 2.\ x'=x+y \wedge y'=y$ use the Substitution Law on the last two parts

$= \quad x:= 1.\ x' = x+2 \ \wedge \ y'=2$ then use the Substitution Law again

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

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

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

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

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

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

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

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

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

$= \quad x'>2$

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

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 be equally 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.

$x'>x \Leftarrow x'=x+1 \wedge y'=y$

$x'=x+1 \wedge y'=y \Leftarrow x:= x+1$

$x' \leq x \Leftarrow$ **if** $x=0$ **then** $x'=x$ **else** $x'<x$ **fi**

$x'>y'>x \Leftarrow y:= x+1.\ x:= y+1$

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 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). 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 binary 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 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 binary expression of the initial values, and P and Q are programs, then **if** b **then** P **else** Q **fi** 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: binary, number, character, bunch, set, string, and list expressions, with all their operators. For now, in Chapter 4, we do not consider the expressions of Chapter 3, functions and quantifiers, to be programming notations, but they are still welcome in specifications. (We will consider functions to be implemented in Section 5.8.)

Part (e) states that any implementable specification P is a program if a program S that refines P is provided. 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.

$x{\geq}0 \Rightarrow x'{=}0 \quad \Leftarrow$ **if** $x{=}0$ **then** *ok* **else** $x{:=}\, x{-}1.\ x{\geq}0 \Rightarrow x'{=}0$ **fi**

The problem is $x{\geq}0 \Rightarrow x'{=}0$. The solution is **if** $x{=}0$ **then** *ok* **else** $x{:=}\, x{-}1.\ x{\geq}0 \Rightarrow x'{=}0$ **fi**. 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 $x{\geq}0 \Rightarrow x'{=}0$ is a program. By saying "recursion is allowed" we break the impasse and declare that $x{\geq}0 \Rightarrow x'{=}0$ is a program. We allow recursion because we know how to implement recursion. A computer executes $x{\geq}0 \Rightarrow x'{=}0$ 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** $x{:=}\, x{-}1.\ x{\geq}0 \Rightarrow x'{=}0$ **fi**	Replace *ok*; Substitution Law
$=$	**if** $x{=}0$ **then** $x'{=}x$ **else** $x{-}1{\geq}0 \Rightarrow x'{=}0$ **fi**	use context $x{=}0$ to modify the **then**-part and use context $x{\neq}0$ and x: *int* to modify the **else**-part
$=$	**if** $x{=}0$ **then** $x{\geq}0 \Rightarrow x'{=}0$ **else** $x{\geq}0 \Rightarrow x'{=}0$ **fi**	Case Idempotence
$=$	$x{\geq}0 \Rightarrow x'{=}0$	

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 **fi** and $C \Leftarrow E$ and $D \Leftarrow F$ are theorems,
then $A \Leftarrow$ **if** b **then** E **else** F **fi** 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 **fi** and $E \Leftarrow$ **if** b **then** F **else** G **fi** are theorems,
then $A \wedge E \Leftarrow$ **if** b **then** $C \wedge F$ **else** $D \wedge G$ **fi** 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. The last refinement law is

Refinement by Cases

$P \Leftarrow$ **if** b **then** Q **else** R **fi** 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

$x' \leq x \Leftarrow$ **if** $x=0$ **then** $x'=x$ **else** $x'<x$ **fi**

by proving both

$x' \leq x \Leftarrow x=0 \wedge x'=x$

and

$x' \leq x \Leftarrow x \neq 0 \wedge x'<x$

End of Refinement Laws

4.1.1 List Summation

As an example of program development, let us do Exercise 174: 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.

$s' = \Sigma L$

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.

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

(Remember: list indexes start at 0, and the list $[n;..\#L]$ includes n and excludes $\#L$.) This theorem is easily proved by two applications of the Substitution Law. 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. So we use the Case Creation law next.

$s' = s + \Sigma\, L\,[n;..\#L] \Leftarrow$ **if** $n=\#L$ **then** $n=\#L \Rightarrow s' = s + \Sigma\, L\,[n;..\#L]$
else $n \neq \#L \Rightarrow s' = s + \Sigma\, L\,[n;..\#L]$ **fi**

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

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

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.

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

This refinement is proved 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 $n{\leq}\#L$. We may therefore consider that $n{\leq}\#L$ 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

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

For those who were uncomfortable about the use of implicit information in the preceding paragraph, the first part of the antecedent $0{\leq}n{\leq}\#L$ 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 a name (identifier). For example, the summation program looks like

$A \;\Leftarrow\; s{:=}\,0.\;\; n{:=}\,0.\;\; B$

$B \;\Leftarrow\;$ **if** $n{=}\#L$ **then** C **else** D **fi**

$C \;\Leftarrow\; ok$

$D \;\Leftarrow\; s{:=}\, s + L\,n.\;\; n{:=}\, n{+}1.\;\; B$

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** $s{:=}\, s + L\,n.\;\; n{:=}\, n{+}1.\;\; B$ **fi**

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 goto, 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. Some textbooks use the words “iteration” and “loop” exclusively for a recursion that can be translated as a branch instruction.

End of List Summation

4.1.2 Binary Exponentiation

Now let's try Exercise 179: 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** $x>0 \Rightarrow y'=2^x$ **fi**
$x=0 \Rightarrow y'=2^x$ $\Leftarrow$ $y:= 1.\ x:= 3$
$x>0 \Rightarrow y'=2^x$ $\Leftarrow$ $x>0 \Rightarrow y'=2^{x-1}.\ y'=2\times y$
$x>0 \Rightarrow y'=2^{x-1}$ $\Leftarrow$ $x'=x-1.\ y'=2^x$
$y'=2\times y$ $\Leftarrow$ $y:= 2\times y.\ x:= 5$
$x'=x-1$ $\Leftarrow$ $x:= x-1.\ y:= 7$

The first refinement divides the problem into two cases; in the second case $x\neq 0$, 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 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 **fi**
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** $x:= x-1.\ y:= 7.\ A.\ y:= 2\times y.\ x:= 5$ **fi**

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

In Section 4.1.1 we solved list summation as follows.

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

$s' = s + \Sigma\ L\ [n;..\#L]$ $\Leftarrow$ **if** $n=\#L$ **then** ok
else $s:= s + L\ n.\ n:= n+1.\ s' = s + \Sigma\ L\ [n;..\#L]$ **fi**

The loop specification $s' = s + \Sigma\ L\ [n;..\#L]$ says, at any iteration of the loop, what is left to be

done: make the final value s' be the current sum s plus the sum of the remaining items $\Sigma\, L\,[n;..\#L]$. This specification "looks forward" to the rest of the computation.

Here is another solution.

$$s' = \Sigma L \quad \Leftarrow \quad s{:=}\,0.\ \ n{:=}\,0.\ \ s{=}(\Sigma\, L\,[0;..n]) \Rightarrow s'{=}\Sigma L$$

$$s{=}(\Sigma\, L\,[0;..n]) \Rightarrow s'{=}\Sigma L \quad \Leftarrow \quad \textbf{if } n{=}\#L \textbf{ then } ok$$
$$\textbf{else } s{:=}\,s + L\,n.\ \ n{:=}\,n{+}1.\ \ s{=}(\Sigma\, L\,[0;..n]) \Rightarrow s'{=}\Sigma L \textbf{ fi}$$

The loop specification $s{=}(\Sigma\, L\,[0;..n]) \Rightarrow s'{=}\Sigma L$ says, at any iteration of the loop, what has been done: given (as an antecedent) what is already computed $s{=}(\Sigma\, L\,[0;..n])$, complete the whole computation $s'{=}\Sigma L$. This specification "looks backward" at what has been computed.

Backward specifications are stronger than forward specifications. From a backward specification, which says both what has been done, and what the whole computation does, we can determine what remains to be done. But from a forward specification, which says only what remains to be done, we cannot determine what has been done. Backward specifications are typically longer than forward specifications, and harder to refine (because they are stronger). A specification may be partly backward and partly forward, with an antecedent that says something but not enough about the past computation, and a consequent that says more particularly what remains to be done than just saying complete the whole computation.

In the example, it was equally easy to say what has been done and what remains to be done. When that is the case, forward specifications are preferable. In some problems, one direction may be easy to say, and the other direction difficult to express. When that is the case, we might reasonably choose the easier direction.

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 nonnegative reals, whichever we prefer.

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

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

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 nonnegative extended real. 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 **fi** by
  $t := t +$(the time to evaluate b and branch). **if** b **then** P **else** Q **fi**
- 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. Assignments to the time variable are not executed; they are there for reasoning about time.

In Subsection 4.0.3 we considered an example of the form

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

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** $t := t+1.\ x := x-1.\ t := t+1.\ P$ **fi**

This refinement is a theorem when

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

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** $t := t+1.\ x := x-1.\ t := t+1.\ P$ **fi**

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

$P = x'=0$

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

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

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 proved 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

$$Q = t'{=}\infty$$
$$Q = x'{=}2 \wedge t'{=}\infty$$
$$Q = x'{=}3 \wedge t'{=}\infty$$

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 *xnat*, 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

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

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

$$P = \textbf{if } x{\geq}0 \textbf{ then } x'{=}0 \ \wedge\ t' = t{+}x \textbf{ else } t'{=}\infty \textbf{ fi}$$
$$P = x'{=}0 \ \wedge\ \textbf{if } x{\geq}0 \textbf{ then } t' = t{+}x \textbf{ else } t'{=}\infty \textbf{ fi}$$

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 148(b)):

$R \Leftarrow$ **if** $x{=}1$ **then** *ok* **else** $x{:=}\ div\ x\ 2.\ \ t{:=}\ t{+}1.\ \ R$ **fi**

where x is an integer variable, and

$R \ = \ x'{=}1 \ \wedge$ **if** $x{\geq}1$ **then** $t' \leq t + log\ x$ **else** $t'{=}\infty$ **fi**

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

$R \ = \ x'{=}1 \ \wedge \ (x{\geq}1 \ \Rightarrow \ t' \leq t + log\ x) \wedge (x{<}1 \ \Rightarrow \ t'{=}\infty)$

This exercise uses the functions *div* (divide and round down) and *log* (binary logarithm); they are defined in Section 11.4. 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:

$x'{=}1 \ \Leftarrow$ **if** $x{=}1$ **then** *ok* **else** $x{:=}\ div\ x\ 2.\ \ t{:=}\ t{+}1.\ \ x'{=}1$ **fi**

$x{\geq}1 \ \Rightarrow \ t' \leq t + log\ x \ \Leftarrow$ **if** $x{=}1$ **then** *ok*
else $x{:=}\ div\ x\ 2.\ \ t{:=}\ t{+}1.\ \ x{\geq}1 \Rightarrow t' \leq t + log\ x$ **fi**

$x{<}1 \ \Rightarrow \ t'{=}\infty \ \Leftarrow$ **if** $x{=}1$ **then** *ok* **else** $x{:=}\ div\ x\ 2.\ \ t{:=}\ t{+}1.\ \ x{<}1 \ \Rightarrow \ t'{=}\infty$ **fi**

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

$x'{=}1 \ \Leftarrow$ **if** $x{=}1$ **then** $x'{=}x \wedge t'{=}t$ **else** $x'{=}1$ **fi**
$x{\geq}1 \ \Rightarrow \ t' \leq t + log\ x \ \Leftarrow$ **if** $x{=}1$ **then** $x'{=}x \wedge t'{=}t$
else $div\ x\ 2 \geq 1 \ \Rightarrow \ t' \leq t + 1 + log\ (div\ x\ 2)$ **fi**
$x{<}1 \ \Rightarrow \ t'{=}\infty \ \Leftarrow$ **if** $x{=}1$ **then** $x'{=}x \wedge t'{=}t$ **else** $div\ x\ 2 < 1 \ \Rightarrow \ t'{=}\infty$ **fi**

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

$x'{=}1 \ \Leftarrow \ x{=}1 \wedge x'{=}x \wedge t'{=}t$
$x'{=}1 \ \Leftarrow \ x{\neq}1 \wedge x'{=}1$

$x{\geq}1 \ \Rightarrow \ t' \leq t + log\ x \ \Leftarrow \ x{=}1 \wedge x'{=}x \wedge t'{=}t$
$x{\geq}1 \ \Rightarrow \ t' \leq t + log\ x \ \Leftarrow \ x{\neq}1 \wedge (div\ x\ 2 \geq 1 \ \Rightarrow \ t' \leq t + 1 + log\ (div\ x\ 2))$

$x{<}1 \ \Rightarrow \ t'{=}\infty \ \Leftarrow \ x{=}1 \wedge x'{=}x \wedge t'{=}t$
$x{<}1 \ \Rightarrow \ t'{=}\infty \ \Leftarrow \ x{\neq}1 \wedge (div\ x\ 2 < 1 \ \Rightarrow t'{=}\infty)$

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,

$(x'{=}1 \ \Leftarrow \ x{\neq}1 \wedge x'{=}1)$ by specialization
$= \ \top$

Next,

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

Next comes the hardest one of the six.

$(x\geq 1 \;\Rightarrow\; t' \leq t + log\ x \;\Leftarrow\; x\neq 1 \wedge (div\ x\ 2 \geq 1 \;\Rightarrow\; t' \leq t + 1 + log\ (div\ x\ 2)))$

Again use the first Law of Portation to move the initial antecedent over to the solution side where it becomes a conjunct.

$=\quad t' \leq t + log\ x \;\Leftarrow\; x>1 \wedge (div\ x\ 2 \geq 1 \;\Rightarrow\; t' \leq t + 1 + log\ (div\ x\ 2))$

Since x is an integer, $x>1 \;=\; div\ x\ 2 \geq 1$, so by the first Law of Discharge,

$=\quad t' \leq t + log\ x \;\Leftarrow\; x>1 \;\wedge\; t' \leq t + 1 + log\ (div\ x\ 2)$

By the first Law of Portation, move $t' \leq t + 1 + log\ (div\ x\ 2)$ over to the left side.

$=\quad (t' \leq t + 1 + log\ (div\ x\ 2) \Rightarrow t' \leq t + log\ x) \;\Leftarrow\; x>1$

By a Connection Law, $(t'\leq a \Rightarrow t'\leq b) \;\Leftarrow\; a\leq b$.

$\Leftarrow\quad t + 1 + log\ (div\ x\ 2) \leq t + log\ x \;\Leftarrow\; x>1$ subtract 1 from each side

$=\quad t + log\ (div\ x\ 2) \leq t + log\ x - 1 \;\Leftarrow\; x>1$ law of logarithms

$=\quad t + log\ (div\ x\ 2) \leq t + log\ (x/2) \;\Leftarrow\; x>1$ *log* and + are monotonic for $x>0$

$\Leftarrow\quad div\ x\ 2 \leq x/2$ *div* is / and then round down

$=\quad \top$

The next one is easier.

$(x<1 \;\Rightarrow\; t'=\infty \;\Leftarrow\; x=1 \wedge x'=x \wedge t'=t)$ Law of Portation

$=\quad t'=\infty \;\Leftarrow\; x<1 \wedge x=1 \wedge x'=x \wedge t'=t$ Put $x<1 \wedge x=1$ together, and first Base Law

$=\quad t'=\infty \;\Leftarrow\; \bot$ last Base Law

$=\quad \top$

And finally,

$(x<1 \;\Rightarrow\; t'=\infty \;\Leftarrow\; x\neq 1 \wedge (div\ x\ 2 < 1 \;\Rightarrow t'=\infty))$ Law of Portation

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

$=\quad t'=\infty \;\Leftarrow\; x<1 \wedge t'=\infty$ 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) $x'=2 \;\wedge\; t'\leq t+1$

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

$x'=2 \;\Leftarrow\; t:= t+1.\ x'=2$

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 $x'\neq 2$, 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) $\wedge\ t'{\geq}t$ 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 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,

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

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 $x'{\neq}2$ 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 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

$$x'{=}2 \wedge t'{\leq}t{+}1 \quad \Longleftarrow \quad t{:=}\ t{+}1.\ x'{=}2 \wedge t'{\leq}t{+}1$$

is not a theorem. We refine

$$x'{=}2 \wedge t'{\leq}t{+}1 \quad \Longleftarrow \quad x{:=}\ 2$$

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

$$P \quad \Longleftarrow \quad \text{(something possibly involving recursive calls to } P\text{)}$$

then observations of the corresponding computation(s) (at finite times) will not 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

$$P \quad \Longleftarrow \quad \text{(something possibly involving recursive calls to } P\text{)}$$

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

$$P \quad \Longleftarrow \quad Q$$

$$Q \quad \Longleftarrow \quad \text{(something possibly involving recursive calls to } Q\text{)}$$

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 186: 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

$$\neg\, x{:}\, L\,(0,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L) \;\wedge\; t' \le t{+}\#L$$

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:

$$\neg\, x{:}\, L\,(0,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L) \;\Longleftarrow$$
$$h{:=}\,0.\;\; h{\le}\#L \;\Rightarrow\; \neg\, x{:}\, L\,(h,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L)$$

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 $L\,h' = x \vee h'{=}\#L$, we need to test either $L\,h = x$ or $h{=}\#L$. To test $L\,h = x$ we need to know $h{<}\#L$, so we have to test $h{=}\#L$ first.

$$h{\le}\#L \;\Rightarrow\; \neg\, x{:}\, L\,(h,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L) \;\Longleftarrow$$
$$\textbf{if } h{=}\#L \textbf{ then } ok \textbf{ else } h{<}\#L \;\Rightarrow\; \neg\, x{:}\, L\,(h,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L) \textbf{ fi}$$

In the remaining problem we are able to test $L\,h = x$.

$$h{<}\#L \;\Rightarrow\; \neg\, x{:}\, L\,(h,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L) \;\Longleftarrow$$
$$\textbf{if } L\,h = x \textbf{ then } ok \textbf{ else } h{:=}\,h{+}1.\;\; h{\le}\#L \;\Rightarrow\; \neg\, x{:}\, L\,(h,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L) \textbf{ fi}$$

Now for the timing:

$$t' \le t{+}\#L \;\Longleftarrow\; h{:=}\,0.\;\; h{\le}\#L \Rightarrow t' \le t{+}\#L{-}h$$
$$h{\le}\#L \Rightarrow t' \le t{+}\#L{-}h \;\Longleftarrow\; \textbf{if } h{=}\#L \textbf{ then } ok \textbf{ else } h{<}\#L \Rightarrow t' \le t{+}\#L{-}h \textbf{ fi}$$
$$h{<}\#L \Rightarrow t' \le t{+}\#L{-}h \;\Longleftarrow\; \textbf{if } L\,h = x \textbf{ then } ok$$
$$\textbf{else } h{:=}\,h{+}1.\;\; t{:=}\,t{+}1.\;\; h{\le}\#L \Rightarrow t' \le t{+}\#L{-}h \textbf{ fi}$$

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 $\neg\, x{:}\, L\,(0,..h') \wedge (L\,h' = x \vee h'{=}\#L)$ and $t'{\le}t{+}\#L$. So the same refinement works for their conjunction, and that solves the original problem. We could have divided the specification $\neg\, x{:}\, L\,(0,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L)$ into parts also. And we should prove our refinements.

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

$$\neg\, x{:}\, L\,(0,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L) \;\wedge\; t' \le t{+}\#L \;\Longleftarrow$$
$$h{:=}\,0.\;\; h{\le}\#L \;\Rightarrow\; \neg\, x{:}\, L\,(h,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L) \;\wedge\; t' \le t{+}\#L{-}h$$
$$h{\le}\#L \;\Rightarrow\; \neg\, x{:}\, L\,(h,..h') \;\wedge\; (L\,h' = x \vee h'{=}\#L) \;\wedge\; t' \le t{+}\#L{-}h \;\Longleftarrow$$
$$\textbf{if } h = \#L \textbf{ then } ok$$
$$\textbf{else if } L\,h = x \textbf{ then } ok$$
$$\textbf{else } h{:=}\,h{+}1.\;\; t{:=}\,t{+}1.$$
$$h{\le}\#L \;\Rightarrow\; \neg\, x{:}\, L\,(h,..h') \wedge (L\,h' = x \vee h'{=}\#L) \wedge t' \le t{+}\#L{-}h \textbf{ fi fi}$$

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

$\neg\ x{:}\ L\,(0,..h')\ \wedge\ (L\,h' = x \vee h'{=}\#L)\ \wedge\ t' \le t{+}\#L\ \Longleftarrow$
$\quad h{:=}\,0.\ h{<}\#L\ \Rightarrow\ \neg\ x{:}\ L\,(h,..h')\ \wedge\ (L\,h' = x \vee h'{=}\#L)\ \wedge\ t' \le t{+}\#L{-}h$

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. 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 join one value to the end of it. If we can do so cheaply enough, we should begin by joining x. Then the search is sure to find x, and we can skip the test $h{=}\#L$ each iteration. The program, ignoring time, becomes

$\neg\ x{:}\ L\,(0,..h')\ \wedge\ (L\,h' = x \vee h'{=}\#L)\ \Longleftarrow\ L{:=}\,L;;[x].\ h{:=}\,0.\ Q$
$Q\ \Longleftarrow$ **if** $L\,h = x$ **then** *ok* **else** $h{:=}\,h{+}1.\ Q$ **fi**

where $Q\ =\ L\,(\#L{-}1) = x \wedge h{<}\#L\ \Rightarrow\ L'{=}L\ \wedge\ \neg\ x{:}\ L\,(h,..h')\ \wedge\ L\,h' = x$.

End of Linear Search

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; one can sometimes 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.

4.2.5 Binary Search

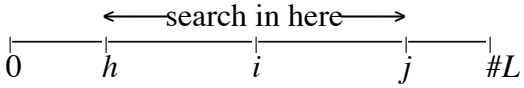
Exercise 187: Write a program to find a given item in a given nonempty sorted list. The execution time must be logarithmic in the length of the list. The strategy is to identify which half of the list contains the item if it occurs at all, then which quarter, then which eighth, and so on.

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 binary variable p the value $\top$ if it is and $\bot$ if not. Ignoring time for the moment, the problem is

$x{:}\ L\,(\Box L)\ =\ p'\ \Longrightarrow\ L\,h' = x$

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 $h,..j$.

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



We can now solve the problem.

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

$h{<}j \Rightarrow R\ \Longleftarrow$ **if** $j{-}h = 1$ **then** $p{:=}\,L\,h = x$ **else** $j{-}h \ge 2 \Rightarrow R$ **fi**

$j{-}h \ge 2 \Rightarrow R\ \Longleftarrow\ \ j{-}h \ge 2\ \Rightarrow\ h'{=}h{<}i'{<}j{=}j'.$
$\quad$ **if** $L\,i \le x$ **then** $h{:=}\,i$ **else** $j{:=}\,i$ **fi**.
$\quad h{<}j \Rightarrow R$

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.

$$j-h \geq 2 \;\Rightarrow\; h' = h < i' < j = j' \;\Leftarrow\; i := \mathit{div}\ (h+j)\ 2$$

After finding the mid-point i of the segment $h;..j$, it is tempting to test whether $L\ i = x$; if $L\ i$ 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. According to the real time measure, both the worst case and average execution times are 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

$$\begin{aligned} T &\Leftarrow h := 0.\ j := \#L.\ U \\ U &\Leftarrow \textbf{if}\ j-h = 1\ \textbf{then}\ p := L\ h = x\ \textbf{else}\ V\ \textbf{fi} \\ V &\Leftarrow i := \mathit{div}\ (h+j)\ 2. \\ &\quad\ \textbf{if}\ L\ i \leq x\ \textbf{then}\ h := i\ \textbf{else}\ j := i\ \textbf{fi}. \\ &\quad\ t := t+1.\ U \end{aligned}$$

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

$$\begin{aligned} T &= t' \leq t + \mathit{ceil}\ (\mathit{log}\ (\#L)) \\ U &= h<j \Rightarrow t' \leq t + \mathit{ceil}\ (\mathit{log}\ (j-h)) \\ V &= j-h \geq 2 \Rightarrow t' \leq t + \mathit{ceil}\ (\mathit{log}\ (j-h)) \end{aligned}$$

where ceil is the function that rounds up.

Here are the proofs of the seven refinements in this subsection. For the first refinement

$$(x{:}\ L\ (\Box L) = p' \;\Rightarrow\; L\ h' = x) \;\Leftarrow\; h := 0.\ j := \#L.\ h<j \Rightarrow R$$

we start with the right side.

$$\begin{aligned} & h := 0.\ j := \#L.\ h<j \Rightarrow R && \text{replace } R \text{ and then use Substitution Law twice} \\ = \quad & 0<\#L \Rightarrow (x{:}\ L\ (\Box L) = p' \;\Rightarrow\; L\ h' = x) && \text{we are given that } L \text{ is nonempty} \\ = \quad & (x{:}\ L\ (\Box L) = p' \;\Rightarrow\; L\ h' = x) \end{aligned}$$

The second refinement

$$h<j \Rightarrow R \;\Leftarrow\; \textbf{if}\ j-h = 1\ \textbf{then}\ p := L\ h = x\ \textbf{else}\ j-h \geq 2 \Rightarrow R\ \textbf{fi}$$

can be proved by cases. And its first case is

$$\begin{aligned} & (h<j \Rightarrow R \;\Leftarrow\; j-h = 1 \;\wedge\; (p := L\ h = x)) && \text{portation} \\ = \quad & j-h = 1 \;\wedge\; (p := L\ h = x) \;\Rightarrow\; R && \text{expand assignment and } R \\ = \quad & j-h = 1 \wedge p' = (L\ h = x) \wedge h' = h \wedge i' = i \wedge j' = j \;\Rightarrow\; (x{:}\ L\ (h,..j) = p' \;\Rightarrow\; L\ h' = x) \\ & && \text{use the antecedent as context to simplify the consequent} \\ = \quad & j-h = 1 \wedge p' = (Lh = x) \wedge h' = h \wedge i' = i \wedge j' = j \;\Rightarrow\; (x = L\ h = L\ h = x \;\Rightarrow\; L\ h = x) \\ & && \text{Symmetry and Base and Reflexive Laws} \\ = \quad & \top \end{aligned}$$

The second case of the second refinement is

$(h<j \Rightarrow R \Leftarrow j-h \neq 1 \wedge (j-h \geq 2 \Rightarrow R))$ portation
$= \quad j-h \neq 1 \wedge (j-h \geq 2 \Rightarrow R) \wedge h<j \Rightarrow R$ combine $j-h \neq 1$ and $h<j$
$= \quad j-h \geq 2 \wedge (j-h \geq 2 \Rightarrow R) \Rightarrow R$ discharge
$= \quad j-h \geq 2 \wedge R \Rightarrow R$ specialization
$= \quad \top$

The next refinement

$j-h \geq 2 \Rightarrow R \Leftarrow j-h \geq 2 \Rightarrow h'=h<i'<j=j'.$ **if** $L\, i \leq x$ **then** $h:= i$ **else** $j:= i$ **fi**. $h<j \Rightarrow R$

can be proved by cases. Using the distributive laws of sequential composition, its first case is

$(j-h \geq 2 \Rightarrow R \Leftarrow j-h \geq 2 \Rightarrow h'=h<i'<j=j'.\ L\, i \leq x \wedge (h:= i.\ h<j \Rightarrow R))$ Assertion
$\Leftarrow \quad (j-h \geq 2 \Rightarrow R \Leftarrow j-h \geq 2 \Rightarrow (h'=h<i'<j=j'.\ L\, i \leq x \wedge (h:= i.\ h<j \Rightarrow R)))$ Portation
$= \quad j-h \geq 2 \wedge (j-h \geq 2 \Rightarrow (h'=h<i'<j=j'.\ L\, i \leq x \wedge (h:= i.\ h<j \Rightarrow R))) \Rightarrow R$
discharge $j-h \geq 2$ and specialize
$\Leftarrow \quad (h'=h<i'<j=j'.\ L\, i \leq x \wedge (h:= i.\ h<j \Rightarrow R)) \Rightarrow R$ expand first R and use Substitution
$= \quad (h'=h<i'<j=j'.\ L\, i \leq x \wedge (i<j \Rightarrow (x{:}\ L\,(i,..j) = p' \Longrightarrow L\, h' = x))) \Rightarrow R$
sequential composition
$= \quad (\exists h'', i'', j'', p''\cdot \quad h''=h<i''<j=j'' \wedge L\, i'' \leq x$
$\wedge (i''<j'' \Rightarrow (x{:}\ L\,(i'',..j'') = p' \Longrightarrow L\, h' = x)))$
$\Rightarrow R$ eliminate p'', h'', and j'' by one-point, and rename i'' to i
$= \quad (\exists i\cdot\ h<i<j \wedge L\, i \leq x \wedge (i<j \Rightarrow (x{:}\ L\,(i,..j) = p' \Longrightarrow L\, h' = x))) \Rightarrow R$
use context $i<j$ to discharge
$= \quad (\exists i\cdot\ h<i<j \wedge L\, i \leq x \wedge (x{:}\ L\,(i,..j) = p' \Longrightarrow L\, h' = x)) \Rightarrow R$
If $h<i$ and $L\, i \leq x$ and L is sorted, then $x{:}\ L\,(i,..j) = x{:}\ L\,(h,..j)$
$= \quad (\exists i\cdot\ h<i<j \wedge L\, i \leq x \wedge (x{:}\ L\,(h,..j) = p' \Longrightarrow L\, h' = x)) \Rightarrow R$
note that $x{:}\ L\,(h,..j) = p' \Longrightarrow L\, h' = x$ is R
since it doesn't use i, bring it outside the scope of the quantifier
$= \quad (\exists i\cdot\ h<i<j \ \wedge\ L\, i \leq x) \wedge R \Rightarrow R$ specialize
$= \quad \top$

Its second case

$j-h \geq 2 \Rightarrow R \Leftarrow j-h \geq 2 \Rightarrow h'=h<i'<j=j'.\ L\, i > x \wedge (j:= i.\ h<j \Rightarrow R)$

is proved just like its first case.

The next refinement is

$(j-h \geq 2 \Rightarrow h'=h<i'<j=j' \Leftarrow i:= div\ (h+j)\ 2)$ expand assignment
$= \quad (j-h \geq 2 \Rightarrow h'=h<i'<j=j' \Leftarrow i' = div\ (h+j)\ 2 \wedge p'=p \wedge h'=h \wedge j'=j)$
use the equations in the antecedent as context to simplify the consequent
$= \quad (j-h \geq 2 \Rightarrow h = h < div\ (h+j)\ 2 < j = j \Leftarrow i' = div\ (h+j)\ 2 \wedge p'=p \wedge h'=h \wedge j'=j)$
simplify $h=h$ and $j=j$ and use the properties of div
$= \quad (j-h \geq 2 \Rightarrow \top \Leftarrow i' = div\ (h+j)\ 2 \wedge p'=p \wedge h'=h \wedge j'=j)$ base law twice
$= \quad \top$

The next refinement is

$(T \Leftarrow h:= 0.\ j:= \#L.\ U)$ replace T and U
$= \quad (t' \leq t + ceil\ (log\ (\#L)) \Leftarrow h:= 0.\ j:= \#L.\ h<j \Rightarrow t' \leq t + ceil\ (log\ (j-h)))$
Substitution Law twice
$= \quad (t' \leq t + ceil\ (log\ (\#L)) \Leftarrow 0<\#L \Rightarrow t' \leq t + ceil\ (log\ (\#L-0)))$
$= \quad \top$

The next refinement

$U \Leftarrow$ **if** $j–h = 1$ **then** $p := L\ h = x$ **else** V **fi**

can be proved by cases. And its first case is

$(U \Leftarrow j–h = 1 \wedge (p := L\ h = x))$ expand U and the assignment

$= (h<j \Rightarrow t' \le t + ceil(log(j–h)) \Leftarrow j–h = 1 \wedge p'=(L\ h = x) \wedge h'=h \wedge i'=i \wedge j'=j \wedge t'=t)$

use main antecedent as context in main consequent

$= (h<j \Rightarrow t \le t + ceil\ (log\ 1) \Leftarrow j–h = 1 \wedge p'=(L\ h = x) \wedge h'=h \wedge i'=i \wedge j'=j \wedge t'=t)$

Use $log\ 1 = 0$

$= (h<j \Rightarrow \top \Leftarrow j–h=1 \wedge p'=(L\ h = x) \wedge h'=h \wedge i'=i \wedge j'=j \wedge t'=t)$ base law twice

$= \top$

Its second case is

$(U \Leftarrow j–h \neq 1 \wedge V)$ expand U and V

$= (h<j \Rightarrow t' \le t + ceil\ (log\ (j–h)) \Leftarrow j–h \neq 1 \wedge (j–h \ge 2 \Rightarrow t' \le t + ceil\ (log\ (j–h))))$

portation

$= h<j \wedge j–h \neq 1 \wedge (j–h \ge 2 \Rightarrow t' \le t + ceil\ (log\ (j–h))) \Rightarrow t' \le t + ceil\ (log\ (j–h))$

simplify

$= j–h \ge 2 \wedge (j–h \ge 2 \Rightarrow t' \le t + ceil\ (log\ (j–h))) \Rightarrow t' \le t + ceil\ (log\ (j–h))$ discharge

$= j–h \ge 2 \wedge t' \le t + ceil\ (log\ (j–h)) \Rightarrow t' \le t + ceil\ (log\ (j–h))$ 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 $div\ (h+j)\ 2 < j$

$= (h+j–1)/2 < j$

$= j–h > –1$

$\Leftarrow j–h \ge 2$ **fi**

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

$= ceil\ (log\ (j–h))$ **fi**

Finally, the last refinement

$V \Leftarrow i := div\ (h+j)\ 2.$ **if** $L\ i \le x$ **then** $h := i$ **else** $j := i$ **fi**. $t := t+1.\ U$

can be proved in two cases. First case:

$(V \Leftarrow i := div\ (h+j)\ 2.\ L\ i \le x \wedge (h := i.\ t := t+1.\ U))$ drop $L\ i \le x$ 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 last case

$V \Leftarrow i := div\ (h+j)\ 2.\ L\ i > x\ \wedge\ (j := i.\ t := t+1.\ U)$

is proved just like the first.

End of Binary Search

4.2.6 Fast Exponentiation

Exercise 180: 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** $y>0 \Rightarrow P$ **fi**

$y>0 \Rightarrow P \Leftarrow z := z \times x.\ y := y-1.\ P$

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** $odd\ y \Rightarrow P$ **fi**

$even\ y \wedge y>0 \Rightarrow P \Leftarrow x := x \times x.\ y := y/2.\ P$

$odd\ y \Rightarrow P \Leftarrow z := z \times x.\ y := y-1.\ P$

Each of these refinements is easily proved.

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

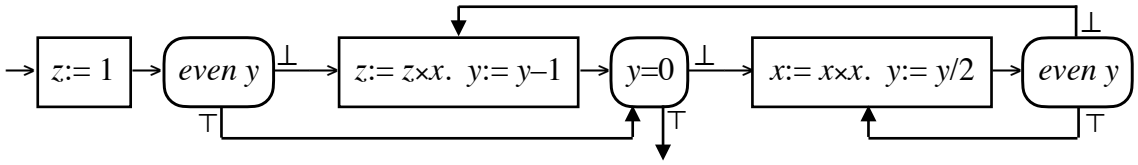
$odd\ y \Rightarrow P \Leftarrow z := z \times x.\ y := y-1.\ even\ y \Rightarrow P$

$even\ y \Rightarrow P \Leftarrow$ **if** $y = 0$ **then** ok **else** $even\ y \wedge y>0 \Rightarrow P$ **fi**

And one more very minor improvement: if the program is used to calculate x^0 less often than x to a positive 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** $odd\ y \Rightarrow P$ **fi**

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** $odd\ y \Rightarrow P$ **fi**
$even\ y \Rightarrow P \Leftarrow$ **if** $y=0$ **then** *ok* **else** $even\ y \wedge y>0 \Rightarrow P$ **fi**
$odd\ y \Rightarrow P \Leftarrow z:= z\times x.\ y:= y-1.\ even\ y \Rightarrow P$
$even\ y \wedge y>0 \Rightarrow P \Leftarrow x:= x\times x.\ y:= y/2.\ y>0 \Rightarrow P$
$y>0 \Rightarrow P \Leftarrow$ **if** *even y* **then** $even\ y \wedge y>0 \Rightarrow P$ **else** $odd\ y \Rightarrow P$ **fi**

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** $t' = t + floor\ (log\ y)$ **fi**

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** $t' \le t + log\ y$ **fi**

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** $even\ y \Rightarrow T$ **else** $odd\ y \Rightarrow T$ **fi**
$even\ y \Rightarrow T \Leftarrow$ **if** $y=0$ **then** *ok* **else** $even\ y \wedge y>0 \Rightarrow T$ **fi**
$odd\ y \Rightarrow T \Leftarrow z:= z\times x.\ y:= y-1.\ even\ y \Rightarrow T$
$even\ y \wedge y>0 \Rightarrow T \Leftarrow x:= x\times x.\ y:= y/2.\ t:= t+1.\ y>0 \Rightarrow T$
$y>0 \Rightarrow T \Leftarrow$ **if** $even\ y$ **then** $even\ y \wedge y>0 \Rightarrow T$ **else** $odd\ y \Rightarrow T$ **fi**

It does not matter that specification T is refined more than once. When we conjoin it 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 256. 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

$fib = 0{\to}0 \mid 1{\to}1 \mid \langle n{:}\ nat{+}2{\cdot}\ fib\ (n{-}2) + fib\ (n{-}1)\rangle$
$= \langle n{:}\ nat{\cdot}$ **if** $n{<}2$ **then** n **else** $fib\ (n{-}2) + fib\ (n{-}1)$ **fi**$\rangle$

We did not include functions in our programming language, so we still have some work to do. Also, the functional solution we have just given has exponential execution time, and we can do much better.

For $n{\geq}2$, 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 $n{\geq}1$, 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.

$P \Leftarrow$ **if** $n{=}0$ **then** $x{:=}\ 0.\ y{:=}\ 1$ **else** $n{:=}\ n{-}1.\ P.\ x'{=}y \wedge y' = x{+}y$ **fi**

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 $t'{=}t$.

$t' = t{+}n \Leftarrow$ **if** $n{=}0$ **then** $x{:=}\ 0.\ y{:=}\ 1$ **else** $n{:=}\ n{-}1.\ t{:=}\ t{+}1.\ t' = t{+}n.\ t'{=}t$ **fi**
$t'{=}t \Leftarrow n{:=}\ x.\ x{:=}\ y.\ y{:=}\ n{+}y$

Linear time is a lot better than exponential time, but we can do even better. Exercise 256 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

$P \Leftarrow$ **if** $n{=}0$ **then** $x{:=}\ 0.\ y{:=}\ 1$
else if $even\ n$ **then** $even\ n \wedge n{>}0 \Rightarrow P$
else $odd\ n \Rightarrow P$ **fi fi**

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 $fib(2{\times}k + 2)$.

$$even\ n \wedge n{>}0 \Rightarrow P \quad \Longleftarrow \quad n{:=}\ n/2-1.\ P.\ x' = 2{\times}x{\times}y + y^2 \ \wedge\ y' = x^2 + y^2 + x'$$

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.

$$x' = x^2 + y^2 \ \wedge\ y' = 2{\times}x{\times}y + y^2 \quad \Longleftarrow \quad n{:=}\ x.\ x{:=}\ x^2 + y^2.\ y{:=}\ 2{\times}n{\times}y + y^2$$
$$x' = 2{\times}x{\times}y + y^2 \ \wedge\ y' = x^2 + y^2 + x' \quad \Longleftarrow \quad n{:=}\ x.\ x{:=}\ 2{\times}x{\times}y + y^2.\ y{:=}\ n^2 + y^2 + x$$

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

$$T \Longleftarrow \textbf{if } n{=}0 \textbf{ then } x{:=}\ 0.\ y{:=}\ 1 \textbf{ else if } even\ n \textbf{ then } even\ n \wedge n{>}0 \Rightarrow T \textbf{ else } odd\ n \Rightarrow T \textbf{ fi fi}$$
$$odd\ n \Rightarrow T \quad \Longleftarrow \quad n{:=}\ (n{-}1)/2.\ t{:=}\ t{+}1.\ T.\ t'{=}t$$
$$even\ n \wedge n{>}0 \Rightarrow T \quad \Longleftarrow \quad n{:=}\ n/2-1.\ t{:=}\ t{+}1.\ T.\ t'{=}t$$
$$t'{=}t \quad \Longleftarrow \quad n{:=}\ x.\ x{:=}\ x^2 + y^2.\ y{:=}\ 2{\times}n{\times}y + y^2$$
$$t'{=}t \quad \Longleftarrow \quad n{:=}\ x.\ x{:=}\ 2{\times}x{\times}y + y^2.\ y{:=}\ n^2 + y^2 + x$$

The first one and last two are easy. Here are the other two.

$$(odd\ n \Rightarrow t' \le t + log\ (n{+}1)) \quad \Longleftarrow \quad (n{:=}\ (n{-}1)/2.\ t{:=}\ t{+}1.\ t' \le t + log\ (n{+}1).\ t'{=}t)$$
$$= \quad (odd\ n \Rightarrow t' \le t + log\ (n{+}1)) \quad \Longleftarrow \quad t' \le t + 1 + log\ ((n{-}1)/2{+}1)$$
$$\text{note that } (a \Rightarrow b) \Longleftarrow c \ =\ a \Rightarrow (b \Longleftarrow c)$$
$$= \quad odd\ n \Rightarrow (t' \le t + log\ (n{+}1) \ \Longleftarrow \ t' \le t + 1 + log\ ((n{-}1)/2{+}1)) \qquad \text{connection law}$$
$$\Longleftarrow \quad odd\ n \ \Rightarrow \ 1 + log\ ((n{-}1)/2{+}1) \le log\ (n{+}1) \qquad \text{logarithm law}$$
$$= \quad odd\ n \ \Rightarrow \ log\ (n{-}1{+}2) \le log\ (n{+}1) \qquad \text{arithmetic}$$
$$= \quad odd\ n \ \Rightarrow \ log\ (n{+}1) \le log\ (n{+}1) \qquad \text{reflexivity and base}$$
$$= \quad \top$$

$$(even\ n \wedge n{>}0 \Rightarrow t' \le t + log\ (n{+}1)) \Longleftarrow (n{:=}\ n/2-1.\ t{:=}\ t{+}1.\ t' \le t + log\ (n{+}1).\ t'{=}t)$$
$$\text{by the same steps}$$
$$= \quad even\ n \wedge n{>}0 \ \Rightarrow \ 1 + log\ (n/2-1{+}1) \le log\ (n{+}1)$$
$$= \quad even\ n \wedge n{>}0 \ \Rightarrow \ log\ n \le log\ (n{+}1)$$
$$= \quad \top$$

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 255 (Collatz), which is a famous program whose execution time is considered to be unknown. Let n be a natural variable. Then, including recursive time,

$$n'{=}1 \quad \Longleftarrow \quad \textbf{if } n{=}1 \textbf{ then } ok$$
$$\textbf{else if } even\ n \textbf{ then } n{:=}\ n/2.\ t{:=}\ t{+}1.\ n'{=}1$$
$$\textbf{else } n{:=}\ 3{\times}n + 1.\ t{:=}\ t{+}1.\ n'{=}1 \textbf{ fi fi}$$

We can express the execution time as $f\,n$, where function f must satisfy

$$t' = t + f\,n \quad \Longleftarrow \quad \textbf{if } n{=}1 \textbf{ then } ok$$
$$\textbf{else if } even\ n \textbf{ then } n{:=}\ n/2.\ t{:=}\ t{+}1.\ t' = t + f\,n$$
$$\textbf{else } n{:=}\ 3{\times}n + 1.\ t{:=}\ t{+}1.\ t' = t + f\,n \textbf{ fi fi}$$

which can be simplified to

$$f\,n \quad = \quad \textbf{if } n{=}1 \textbf{ then } 0$$
$$\textbf{else if } even\ n \textbf{ then } 1 + f\,(n/2)$$
$$\textbf{else } 1 + f\,(3{\times}n + 1) \textbf{ fi fi}$$

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 change of value of 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 clock 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. It is not even known whether f is finite for all $n>0$. 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

$$\neg\ x{:}\ L\ (0,..h') \ \wedge\ L\ h' = x$$

and the program can be simplified to

$$\neg\ x{:}\ L\ (0,..h') \ \wedge\ L\ h' = x \ \Longleftarrow\ h{:=}\ 0.\ \neg\ x{:}\ L\ (h,..h') \ \wedge\ L\ h' = x$$

$$\neg\ x{:}\ L\ (h,..h') \ \wedge\ L\ h' = x \ \Longleftarrow\ \textbf{if}\ L\ h = x\ \textbf{then}\ ok$$
$$\textbf{else}\ h{:=}\ h{+}1.\ \neg\ x{:}\ L\ (h,..h') \ \wedge\ L\ h' = x\ \textbf{fi}$$

Adding recursive time, we can prove

$$t' = t{+}h' \ \Longleftarrow\ h{:=}\ 0.\ t'{=}t{+}h'{-}h$$

$$t' = t{+}h'{-}h \ \Longleftarrow\ \textbf{if}\ L\ h = x\ \textbf{then}\ ok\ \textbf{else}\ h{:=}\ h{+}1.\ t{:=}\ t{+}1.\ t' = t{+}h'{-}h\ \textbf{fi}$$

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 293: 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, proceeding 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 $\Longleftarrow$ **if** $n{=}0$ **then** *ok*
else $n{:=}\ n{-}1.$
MovePile from using to.
MoveDisk from to.
MovePile using to from.
$n{:=}\ n{+}1$ **fi**

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 **fi**

by cases. First case, starting with its right side:

n=0 ∧ *ok* — expand *ok*
$=$ n=0 ∧ n'=n ∧ t'=t — arithmetic
$\Longrightarrow$ t:= $t + 2^n - 1$

Second case, starting with its right side:

n>0 ∧ (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
$\Longrightarrow$ n:= n–1. t:= $t + 2^n - 1$. t:= t+1. t:= $t + 2^n - 1$. n'=n+1 ∧ t'=t
— use substitution law repeatedly from right to left
$=$ n'=n–1+1 ∧ t' = t+2^{n-1}–1+1+2^{n-1}–1 — simplify
$=$ n'=n ∧ t' = t+2^n–1
$=$ 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 the initial space occupied may not be 0 , just

as the computation may not begin 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

$$
\begin{array}{ll}
s'=s \Longleftarrow & \textbf{if } n=0 \textbf{ then } ok \\
& \textbf{else } n := n-1. \\
& \quad s := s+1.\ s'=s.\ s := s-1. \\
& \quad ok. \\
& \quad s := s+1.\ s'=s.\ s := s-1. \\
& \quad n := n+1 \textbf{ fi}
\end{array}
$$

which says that the space occupied is the same at the end as at the start.

It may be 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. Implementability requires $m' \geq m$. Wherever space is being increased, we insert $m := m \uparrow s$ to keep m current. 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 $s \leq m$, since m is supposed to be the maximum value of s. We also assume $m \leq s+n$ so that m does not start larger than the maximum we are trying to prove. The refinement becomes

$$
\begin{array}{ll}
s \leq m \leq s+n \ \Rightarrow\ (m := s+n) \Longleftarrow & \\
\quad \textbf{if } n=0 \textbf{ then } ok & \\
\quad \textbf{else } n := n-1. & \\
\quad\quad s := s+1.\ m := m \uparrow s.\ s \leq m \leq s+n \ \Rightarrow\ (m := s+n).\ s := s-1. & \\
\quad\quad ok. & \\
\quad\quad s := s+1.\ m := m \uparrow s.\ s \leq m \leq s+n \ \Rightarrow\ (m := s+n).\ s := s-1. & \\
\quad\quad n := n+1 \textbf{ fi} &
\end{array}
$$

The proof of the refinement proceeds in the usual two cases. First case:

$$
\begin{array}{lll}
 & n=0 \wedge ok \ \Rightarrow\ (s \leq m \leq s+n \ \Rightarrow\ (m := s+n)) & \text{portation} \\
= & n=0 \ \wedge\ ok \ \wedge\ s \leq m \leq s+n \ \Rightarrow\ (m := s+n) & \text{expand } ok \text{ and assignment} \\
= & n=0 \wedge s'=s \wedge m'=m \wedge n'=n \wedge s \leq m \leq s+n \ \Rightarrow\ s'=s \wedge m'=s+n \wedge n'=n & \text{context } n=0 \\
= & n=0 \wedge s'=s \wedge m'=m \wedge n'=n \wedge s \leq m \leq s \ \Rightarrow\ s'=s \wedge m'=s+n \wedge n'=n & \\
= & n=0 \wedge s'=s \wedge m'=m \wedge n'=n \wedge s=m \ \Rightarrow\ s'=s \wedge m'=s+n \wedge n'=n & \text{context } n=0 \wedge s=m \\
= & n=0 \wedge s'=s \wedge m'=m \wedge n'=n \wedge s=m \ \Rightarrow\ s'=s \wedge m'=m \wedge n'=n & \text{specialization} \\
= & \top &
\end{array}
$$

Before proving the last case, let's simplify the long line that occurs twice.

$s{:=}\,s{+}1.\ m{:=}\,m{\uparrow}s.\ s \le m \le s{+}n \Rightarrow (m{:=}\,s{+}n).\ s{:=}\,s{-}1$ Use an Assertion Law

$\Longrightarrow$ $s{:=}\,s{+}1.\ m{:=}\,m{\uparrow}s.\ s \le m \le s{+}n \Rightarrow (m{:=}\,s{+}n.\ s{:=}\,s{-}1)$ Expand final assignment

$=$ $s{:=}\,s{+}1.\ m{:=}\,m{\uparrow}s.\ s \le m \le s{+}n \Rightarrow (m{:=}\,s{+}n.\ s'{=}s{-}1 \wedge m'{=}m \wedge n'{=}n)$

Use Substitution Law inside brackets

$=$ $s{:=}\,s{+}1.\ m{:=}\,m{\uparrow}s.\ s \le m \le s{+}n \Rightarrow s'{=}s{-}1 \wedge m' = s{+}n \wedge n'{=}n$

Use Substitution Law

$=$ $s{:=}\,s{+}1.\ s \le m{\uparrow}s \le s{+}n \Rightarrow s'{=}s{-}1 \wedge m' = s{+}n \wedge n'{=}n$ Use Substitution Law

$=$ $s{+}1 \le m{\uparrow}(s{+}1) \le s{+}1{+}n \Rightarrow s'{=}s \wedge m' = s{+}1{+}n \wedge n'{=}n$ Simplify antecedent and rewrite consequent

$=$ $m \le s{+}1{+}n \Rightarrow (m{:=}\,s{+}1{+}n)$

Now the last case:

$n{>}0 \wedge (\ n{:=}\,n{-}1.$
$\quad s{:=}\,s{+}1.\ m{:=}\,m{\uparrow}s.\ s \le m \le s{+}n \Rightarrow (m{:=}\,s{+}n).\ s{:=}\,s{-}1.$
$\quad ok.$
$\quad s{:=}\,s{+}1.\ m{:=}\,m{\uparrow}s.\ s \le m \le s{+}n \Rightarrow (m{:=}\,s{+}n).\ s{:=}\,s{-}1.$
$\quad n{:=}\,n{+}1\ )$ Drop $n{>}0$ and ok. Simplify long lines.

$\Longrightarrow$ $n{:=}\,n{-}1.\ m \le s{+}1{+}n \Rightarrow (m{:=}\,s{+}1{+}n).\ m \le s{+}1{+}n \Rightarrow (m{:=}\,s{+}1{+}n).\ n{:=}\,n{+}1$

Assertion Law twice

$\Longrightarrow$ $n{:=}\,n{-}1.\ m \le s{+}1{+}n \Rightarrow (m{:=}\,s{+}1{+}n.\ m \le s{+}1{+}n \Rightarrow (m{:=}\,s{+}1{+}n.\ n{:=}\,n{+}1))$

Expand final assignment and substitution law

$=$ $n{:=}\,n{-}1.\ m \le s{+}1{+}n \Rightarrow (m{:=}\,s{+}1{+}n.\ m \le s{+}1{+}n \Rightarrow s'{=}s \wedge m'{=}s{+}1{+}n \wedge n'{=}n{+}1)$

substitution Law twice more

$=$ $m \le s{+}1{+}n{-}1 \Rightarrow (s{+}1{+}n{-}1 \le s{+}1{+}n{-}1 \Rightarrow s'{=}s \wedge m'{=}s{+}1{+}n{-}1 \wedge n'{=}n{-}1{+}1)$

simplify

$=$ $m \le s{+}n \Rightarrow (s{+}n \le s{+}n \Rightarrow s'{=}s \wedge m'{=}s{+}n \wedge n'{=}n)$ strengthen antecedent $s{\le}m$

$\Longrightarrow$ $s \le m \le s{+}n \Rightarrow (m{:=}\,s{+}n)$

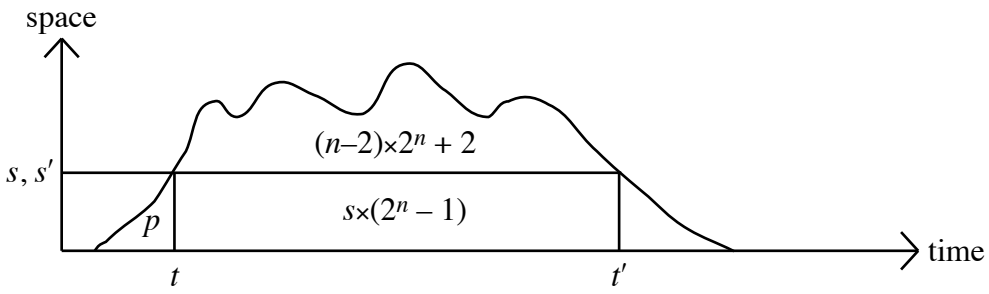
End of Maximum Space

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, p is increased by $s{\times}1$. We prove

$p{:=}\,p + s{\times}(2^n - 1) + (n{-}2){\times}2^n + 2 \Longleftarrow$
$\quad$ **if** $n{=}0$ **then** ok
$\quad$ **else** $n{:=}\,n{-}1.$
$\quad\quad s{:=}\,s{+}1.\ p{:=}\,p + s{\times}(2^n - 1) + (n{-}2){\times}2^n + 2.\ s{:=}\,s{-}1.$
$\quad\quad p{:=}\,p + s{\times}1.$
$\quad\quad s{:=}\,s{+}1.\ p{:=}\,p + s{\times}(2^n - 1) + (n{-}2){\times}2^n + 2.\ s{:=}\,s{-}1.$
$\quad\quad n{:=}\,n{+}1$ **fi**

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. First part:

$n=0 \wedge ok$ expand ok

$=$ $n=0 \wedge n'=n \wedge s'=s \wedge p'=p$ arithmetic

$\Longrightarrow$ $n'=n \ \wedge \ s'=s \ \wedge \ p' = p + s\times(2^n – 1) + (n–2)\times 2^n + 2$

$=$ $p := p + s\times(2^n – 1) + (n–2)\times 2^n + 2$

Last part:

$n>0 \wedge$ ($n := n–1.\ \ s := s+1.\ \ p := p + s\times(2^n – 1) + (n–2)\times 2^n + 2.\ \ s := s–1.\ \ n := n+1.$
$p := p + s\times 1.$
$n := n–1.\ \ s := s+1.\ \ p := p + s\times(2^n – 1) + (n–2)\times 2^n + 2.\ \ s := s–1.\ \ n := n+1$)

drop conjunct $n>0$; expand final assignment

$\Longrightarrow$ $n := n–1.\ \ s := s+1.\ \ p := p + s\times(2^n – 1) + (n–2)\times 2^n + 2.\ \ s := s–1.\ \ n := n+1.\ p := p+s.$
$n := n–1.\ s := s+1.\ p := p + s\times(2^n–1) + (n–2)\times 2^n + 2.\ s := s–1.\ n'=n+1 \wedge s'=s \wedge p'=p$

use substitution law 10 times from right to left

$=$ $n'=n \wedge s'=s$
$\wedge$ $p' = p + (s+1)\times(2^{n–1}–1) + (n–3)\times 2^{n–1} + 2 + s + (s+1)\times(2^{n–1}–1) + (n–3)\times 2^{n–1} + 2$

simplify

$=$ $n'=n \ \wedge \ s'=s \ \wedge \ p' = p + s\times(2^n – 1) + (n–2)\times 2^n + 2$

$=$ $p := p + s\times(2^n – 1) + (n–2)\times 2^n + 2$

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 293(f).

End of Average Space

Putting together all the proofs for the Towers of Hanoi problem, we have

MovePile $\Longleftarrow$ **if** $n=0$ **then** ok
else $n := n–1.$
$s := s+1.\ \ m := m\uparrow s.\ \ MovePile.\ \ s := s–1.$
$t := t+1.\ \ p := p+s.\ \ ok.$
$s := s+1.\ \ m := m\uparrow s.\ \ MovePile.\ \ s := s–1.$
$n := n+1$ **fi**

where *MovePile* is the specification

$n'=n$
$\wedge$ $t' = t + 2^n – 1$
$\wedge$ $s'=s$
$\wedge$ $(s \le m \le s+n \ \Rightarrow \ m' = s+n)$
$\wedge$ $p' = p + s\times(2^n – 1) + (n–2)\times 2^n + 2$

The methods used to find the maximum and average space can be used to find the maximum and average and minimum value of any number variable.

End of Space

4.4 Old Program Theory optional

The original method of proving properties of a computation was to place assertions at strategic points within a program to describe the state of the computation at those points. An assertion is a binary expression that talks about the state where it is located in a program (a program may be as small as ok, or as large as thousands of lines, and it may be within a larger program). An assertion situated at the start of a program is called a precondition for that program; an assertion situated at the end of a program is called a postcondition for that program. An assertion may be both a postcondition for the preceding program and a precondition for the following program. An assertion situated at the start and end of a loop is called an invariant for that loop. There were proof rules that relate the program to its embedded assertions.

We do not present the old theory because it is completely superseded by the theory in this book. The old theory is harder to use, and applies to only some forms of program; the theory in this book is easier to use, and applies to all forms of program. Assertions are situated in a program and have meaning only in their situation; specifications (as defined in this book) are written before and during program development, they guide program development, and they enable proof of each refinement before reaching the program stage. A programmer who is starting to write a program thinks about the purpose of that program, which is what a specification says, not about what will be true at strategic points in the not-yet-written program.

Loops with intermediate and deep exits, and those with intermediate and deep entry points, are difficult or impossible to reason about with assertions. The **go to** statement is famously difficult to reason about with assertions. Specifications (as defined in this book) and recursive refinement make reasoning about all kinds of loops and control constructs easier and more straightforward (see Section 5.2). Even the **go to** can be handled (see Subsection 5.2.4).

If we have a precondition P and postcondition R for a program, we can form a specification $P \Rightarrow R'$ for the program. But specifications are not necessarily implications with only unprimed variables in the antecedent and only primed variables in the consequent, and they are not necessarily decomposable into a precondition and postcondition. A precondition-postcondition pair can always be rewritten as a specification, but a specification cannot always be rewritten as a precondition-postcondition pair. For example, the specification

$$(P \Rightarrow R') \wedge (Q \Rightarrow S')$$

cannot be written as a precondition-postcondition pair.

Specifications (as defined in this book) have every advantage over assertions, but the old theory is still taught and is still in use. So we now present the old terminology in the context of the modern theory.

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 $\forall \sigma \cdot P \Leftarrow S$.

These are the same as refinement except that the quantification is over only one state. Both P and S may talk about the prestate σ and the poststate σ', but the quantification $\forall \sigma'$ makes σ' local, so the exact precondition is an assertion about its nonlocal variables σ. The quantification $\forall \sigma$ makes σ local, so the exact postcondition is an assertion about its nonlocal variables σ'.

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 $x:= x+1$)

$= \quad \forall x' \cdot x'>5 \Leftarrow (x:= x+1)$

$= \quad \forall x' \cdot x'>5 \Leftarrow x'=x+1$ One-Point Law

$= \quad x+1 > 5$

$= \quad x > 4$

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 specification 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 $x:= x+1$)

$= \quad \forall x \cdot x>4 \Leftarrow (x:= x+1)$

$= \quad \forall x \cdot x>4 \Leftarrow x'=x+1$ One-Point Law

$= \quad x'-1 > 4$

$= \quad x' > 5$

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 specification 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 precondition and exact postcondition for P to be refined by S.

Any assertion that implies the exact precondition is called a sufficient precondition.

Any assertion implied by the exact precondition is called a necessary precondition.

Any assertion that implies the exact postcondition is called a sufficient postcondition.

Any assertion implied by the exact postcondition is called a necessary postcondition.

The exact precondition is the necessary and sufficient precondition, and the exact postcondition is the necessary and sufficient postcondition. For example,

$x>2$ is a necessary (but not sufficient) precondition for $x:= x+1$ to refine $x'>5$

$x>6$ is a sufficient (but not necessary) precondition for $x:= x+1$ to refine $x'>5$

$x>4$ is the exact (necessary and sufficient) precondition for $x:= x+1$ to refine $x'>5$

for $x>4$ to be refined by $x:= x+1$, a necessary (but not sufficient) postcondition is $x'>3$

for $x>4$ to be refined by $x:= x+1$, a sufficient (but not necessary) postcondition is $x'>7$

for $x>4$ to be refined by $x:= x+1$, the exact (necessary and sufficient) postcondition is $x'>5$

In English, the word "precondition" means "something that is necessary beforehand". The old theory mistakenly used the word "precondition" to mean "something that is sufficient beforehand". It therefore used the words "weakest precondition" to mean "exact precondition", and the words "strongest postcondition" to mean "exact postcondition".

Exercise 301(c) asks for the exact precondition and exact 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 Section 11.4). 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

$=$ $x \neq -1 \wedge x \neq 0 \wedge x \neq 1$

If x starts anywhere but -1, 0, or 1, it will move farther from zero.

(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

$=$ $x' \neq 0 \wedge x' \neq 1$

If x ends anywhere but 0 or 1, it did move farther from zero.

Let S be a specification, let I be an assertion with all nonlocal variables unprimed, and let I' be the same as I but with primes on all nonlocal variables. Then I is an invariant for S if $I \Rightarrow I'$ is refined by S.

$\forall \sigma, \sigma' \cdot (I \Rightarrow I') \Leftarrow S$

Here are two equivalent definitions.

$\forall \sigma, \sigma' \cdot I \wedge S \Rightarrow I'$

$\forall \sigma, \sigma' \cdot I \Rightarrow (S \Rightarrow I')$

Executing S in a state where I is true creates a state in which I is again true.

Exercise 304(f) asks us to prove that $y = x^2$ is an invariant for $(x := x+1.\ y := y + 2 \times x - 1)$ where the variables are x and y. Working within the quantifications,

$(I \Rightarrow I') \Leftarrow S$ replace I and S

$=$ $(y = x^2 \Rightarrow y' = x'^2) \Leftarrow (x := x+1.\ y := y + 2 \times x - 1)$ replace last assignment

$=$ $(y = x^2 \Rightarrow y' = x'^2) \Leftarrow (x := x+1.\ x' = x \wedge y' = y + 2 \times x - 1)$ substitution

$=$ $(y = x^2 \Rightarrow y' = x'^2) \Leftarrow x' = x+1 \wedge y' = y + 2 \times (x+1) - 1$ arithmetic

$=$ $(y = x^2 \Rightarrow y' = x'^2) \Leftarrow x' = x+1 \wedge y' = y + 2 \times x + 1$ context

$=$ $(y = x^2 \Rightarrow (y + 2 \times x + 1) = (x+1)^2) \Leftarrow x' = x+1 \wedge y' = y + 2 \times x + 1$ arithmetic and cancellation

$=$ $(y = x^2 \Rightarrow y = x^2) \Leftarrow x' = x+1 \wedge y' = y + 2 \times x + 1$ reflexive, base

$=$ $\top$

There are correct loops whose correctness cannot be proved using an invariant (see Exercise 326 and Subsection 5.2.3), but all correct loops can be proved correct using the specifications and theory in this book (see Subsection 4.2.3).

In addition to the invariant associated with a loop, the old theory had a variant (or bound function, or well-founded relation) for the purpose of proving termination. A variant is a natural-valued expression whose value decreases each iteration. A variant is really just a time bound, using the recursive measure, with a clock that runs backward. It enables proof that a computation terminates, but it does not enable proof that a computation does not terminate. The theory in this book enables us to prove both termination and nontermination.

This section has given meaning to the old terminology (assertion, precondition, postcondition, invariant, variant) within the modern theory, but the old terms are of minor importance within the modern theory. Preconditions, postconditions, and variants are not used. Invariants are used only in a special case of **for**-loops (Subsection 5.2.3). The only remaining use for assertions is to add error-checking redundancy to a program (Section 5.4).

End of Old Program Theory

End of Program Theory

5 Programming Language

We have been using a very simple programming language consisting of only *ok* , assignment, **if then else fi** , 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 (parallelism) 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 with a local scope is so useful that it is provided by every decent programming language. A declaration may look something like this:

new 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 in Section 11.6. 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 binary expression in the initial and final state.

new x: T· P = $\exists x, x'$: T· 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 **new** x: T· 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

new x: *int*· x:= 2. y:= x+z
= $\exists x, x'$: *int*· x'=2 ∧ y' = 2+z ∧ z'=z
= y' = 2+z ∧ z'=z

According to our definition of variable declaration, the initial value of the local variable is an arbitrary value of its type.

new x: *int*· y:= x
= $\exists x, x'$: *int*· x'=x ∧ y'=x ∧ z'=z
= 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

new x: *int*· y:= x–x
= y'=0 ∧ 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 binary expressions, we introduce the expression *undefined* but we do not give any axioms about it (that's what makes it undefined), so nothing can be proved about it. Then

new x: T· P = $\exists x$: *undefined*· $\exists x'$: T, *undefined*· 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. If e is of type T,

$$\textbf{new}\ x{:}\ T := e\cdot\ P \quad = \quad \exists x{:}\ e\cdot\ \exists x'{:}\ T\cdot\ P$$

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.

We can declare several variables in one declaration. For example,

$$\textbf{new}\ x, y, z{:}\ T\cdot\ P \quad = \quad \exists x, x', y, y', z, z'{:}\ T\cdot\ P$$

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 in Section 11.6, just like **new**. 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

$$\textbf{frame}\ x, y\cdot\ P \quad = \quad P \wedge w'{=}w \wedge z'{=}z$$

Within P the state variables are x and y; we can refer to w and z, but only as 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

Assignment and ok were defined formally at the low level (using σ and σ'), but at the high level they were defined informally

$$ok \quad = \quad x'{=}x \wedge y'{=}y \wedge \ldots$$
$$x{:=}\ e \quad = \quad x'{=}e \wedge y'{=}y \wedge \ldots$$

using three dots to say "and other conjuncts for other state variables". If we had defined **frame** first, we could have defined them formally at the high level 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 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.

$$s{:=}\ \Sigma L \quad = \quad \textbf{frame}\ s\cdot\ \ \textbf{new}\ n{:}\ nat\cdot\ \ s' = \Sigma L$$

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 refine $s' = \Sigma L$ as before.

End of Scope

5.1 Data Structures

5.1.0 Array

In most popular programming languages there is the notion of indexed variable, usually called an "array" (same word as a multidimensional list, but different meaning). Each part of an array is called an "element" (same word as an element of a bunch or set, but different meaning), and each element is an assignable 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 unless they are needed for precedence. We can write an array element assignment as a binary 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

$$A\,i := e \quad = \quad A'i{=}e \ \wedge\ (\forall j{\cdot}\ j{\neq}i \Rightarrow A'j = A\,j) \ \wedge\ x'{=}x \ \wedge\ y'{=}y \ \wedge\ \ldots$$

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

$$\begin{array}{ll} & A(A\,2) := 3 \\ = & A'(A\,2) = 3 \ \wedge\ (\forall j{\cdot}\ j \neq A\,2 \ \Rightarrow\ A'j = A\,j) \ \wedge\ x'{=}x \ \wedge\ y'{=}y \ \wedge\ \ldots \end{array}$$

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,

$$A\,2 := 3.\ \ i := 2.\ \ A\,i := 4.\ \ A\,i = A\,2$$

should equal $\top$, because $i{=}2$ just before the final binary expression, and $A\,2 = A\,2$ certainly equals $\top$. If we try to apply the Substitution Law, we get

$$\begin{array}{lll} & A\,2 := 3.\ \ i := 2.\ \ A\,i := 4.\ \ A\,i = A\,2 & \text{invalid use of substitution law} \\ = & A\,2 := 3.\ \ i := 2.\ \ 4 = A\,2 & \text{valid use of substitution law} \\ = & A\,2 := 3.\ \ 4 = A\,2 & \text{invalid use of substitution law} \\ = & 4{=}3 & \\ = & \bot & \end{array}$$

Here is a second example of the failure of the Substitution Law for array elements.

$$A\,2 := 2.\ \ A(A\,2) := 3.\ \ A\,2 = 2$$

This should equal $\bot$ because $A\,2 = 3$ just before the final binary expression. But the Substitution Law says

$$\begin{array}{lll} & A\,2 := 2.\ \ A(A\,2) := 3.\ \ A\,2 = 2 & \text{invalid use of substitution law} \\ = & A\,2 := 2.\ \ A\,2 = 2 & \text{invalid use of substitution law} \\ = & 2{=}2 & \\ = & \top & \end{array}$$

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.

$$\begin{array}{ll} & A\,i := e \\ = & A'i{=}e \ \wedge\ (\forall j{\cdot}\ j{\neq}i \Rightarrow A'j = A\,j) \ \wedge\ x'{=}x \ \wedge\ y'{=}y \ \wedge\ \ldots \\ = & A' = i{\rightarrow}e \,|\, A \ \wedge\ x'{=}x \ \wedge\ y'{=}y \ \wedge\ \ldots \\ = & A := i{\rightarrow}e \,|\, A \end{array}$$

Let us look again at the examples for which the Substitution Law did not work, this time using the notation $A{:=}\ i{\rightarrow}e \mid A$.

$$
\begin{array}{ll}
 & A{:=}\ 2{\rightarrow}3 \mid A.\ \ i{:=}\ 2.\ \ A{:=}\ i{\rightarrow}4 \mid A.\ \ A\ i\ = A\ 2 \\
= & A{:=}\ 2{\rightarrow}3 \mid A.\ \ i{:=}\ 2.\ \ (i{\rightarrow}4 \mid A)i = (i{\rightarrow}4 \mid A)2 \\
= & A{:=}\ 2{\rightarrow}3 \mid A.\ \ (2{\rightarrow}4 \mid A)2 = (2{\rightarrow}4 \mid A)2 \\
= & A{:=}\ 2{\rightarrow}3 \mid A.\ \ \top \\
= & \top
\end{array}
$$

$$
\begin{array}{ll}
 & A{:=}\ 2{\rightarrow}2 \mid A.\ \ A{:=}\ A\ 2 \rightarrow 3 \mid A.\ \ A\ 2 = 2 \\
= & A{:=}\ 2{\rightarrow}2 \mid A.\ \ (A\ 2 \rightarrow 3 \mid A)2 = 2 \\
= & ((2{\rightarrow}2 \mid A)2{\rightarrow}3 \mid 2{\rightarrow}2 \mid A)\ 2 = 2 \\
= & (2{\rightarrow}3 \mid 2{\rightarrow}2 \mid A)\ 2 = 2 \\
= & 3 = 2 \\
= & \bot
\end{array}
$$

This time the results are correct. The only thing to remember about array element assignment is this: change $A\ i{:=}\ e$ to $A{:=}i{\rightarrow}e \mid A$ before applying any programming theory. A two-dimensional array element assignment $A\ i\ j{:=}\ e$ must be changed to $A{:=}\ (i;\ j){\rightarrow}e \mid A$, and similarly for more dimensions. In program theory, an array is a list variable, and array element assignment assigns the list variable to a new list that is like the old list but differs in one item.

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.

$$
\begin{array}{lll}
person & = & \text{“name”} \rightarrow text \\
 & \mid & \text{“age”} \rightarrow nat
\end{array}
$$

We declare

new *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* **od**

where *b* is binary and *P* is a specification. To execute it, evaluate *b* , and if its value is ⊥ then you're done, but if its value is ⊤ 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 the refinement

W ⟸ **while** b **do** P **od**

to be an alternative notation for the refinement

W ⟸ **if** b **then** P. W **else** ok **fi**

For example, to prove

$s' = s + \Sigma\, L\, [n;..\#L] \;\wedge\; t' = t + \#L - n$ ⟸
while $n{\neq}\#L$ **do** $s{:=}\, s + L\, n.\;\; n{:=}\, n{+}1.\;\; t{:=}\, t{+}1$ **od**

prove instead

$s' = s + \Sigma\, L\, [n;..\#L] \;\wedge\; t' = t + \#L - n$ ⟸
if $n{\neq}\#L$ **then** $s{:=}\, s + L\, n.\;\; n{:=}\, n{+}1.\;\; t{:=}\, t{+}1.\;\; s' = s + \Sigma\, L\, [n;..\#L] \;\wedge\; t' = t{+}\#L{-}n$
else ok **fi**

During programming, we may happen to refine a specification W by **if** b **then** P. W **else** ok **fi** . If so, we may use the alternative refinement notation with a **while**-loop, although there is no good reason to do so.

Exercise 320 (unbounded bound): Consider the following program in natural variables x and y .

while ¬ $x{=}y{=}0$
do **if** $y{>}0$ **then** $y{:=}\, y{-}1$
else $x{:=}\, x{-}1.$ **new** n: $nat{\cdot}\; y{:=}\, n$ **fi**
od

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 refinement.

P ⟸ **if** $x{=}y{=}0$ **then** ok
else if $y{>}0$ **then** $y{:=}\, y{-}1$ **else** $x{:=}\, x{-}1.$ $(\exists n{\cdot}\; y{:=}\, n)$ **fi**.
$t{:=}\, t{+}1.\;\; P$ **fi**

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

$t' = t{+}x{+}y{+}s$ ⟸ **if** $x{=}y{=}0$ **then** ok
else **if** $y{>}0$ **then** $y{:=}\, y{-}1.\;\; t{:=}\, t{+}1.\;\; t' = t{+}x{+}y{+}s$
else $x{:=}\, x{-}1.\;\; y{:=}\, f\, x.\;\; t{:=}\, t{+}1.\;\; t' = t{+}x{+}y{+}s$ **fi** **fi**

The proof is in three cases.

$x{=}y{=}0 \;\wedge\; ok$
⟹ $x{=}y{=}s{=}0 \;\wedge\; t'{=}t$
⟹ $t' = t{+}x{+}y{+}s$

$y{>}0 \wedge (y{:=}\, y{-}1.\ \ t{:=}\, t{+}1.\ \ t' = t{+}x{+}y{+}s)$ substitution law twice
$=$ $y{>}0 \ \wedge\ t' = t{+}1{+}x{+}y{-}1{+}s$
$\Longrightarrow$ $t' = t{+}x{+}y{+}s$

$x{>}0 \wedge y{=}0 \wedge (x{:=}\, x{-}1.\ \ y{:=}\, f\, x.\ \ t{:=}\, t{+}1.\ \ t' = t{+}x{+}y{+}s)$ substitution law 3 times
$=$ $x{>}0 \ \wedge\ y{=}0 \ \wedge\ t' = t{+}1{+}x{-}1{+}f\,(x{-}1){+}\Sigma\, f\,[0;..x{-}1]$
$\Longrightarrow$ $t' = t{+}x{+}y{+}s$

The execution time of the program is $x + y +$ (the sum of x arbitrary natural numbers) .

A different (least fixed-point) account of **while**-loops is given in Subsection 6.1.1.

End of While-Loop

5.2.1 Exit-Loop

Some languages provide a command to jump out of the middle of a loop. Suppose the loop syntax is

do *P* **od**

with the additional syntax

exit when *b*

allowed within P, where b is binary. Some programming languages say "break" 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 ⇐ do A.
       exit when b.
       C
     od
```

is an alternative notation for

$L \Leftarrow A.$ **if** b **then** ok **else** $C.\ L$ **fi**

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 binary 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 ⇐ do A.
       do B.
          exit 2 when c.
          D
       od.
       E
     od
```

The refinement structure corresponding to this loop is

$P \Leftarrow A.\ Q$
$Q \Leftarrow B.$ **if** c **then** ok **else** $D.\ Q$ **fi**

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 ⟸  do  A.
         exit 1 when b.
         C.
         do  D.
             exit 2 when e.
             F.
             exit 1 when g.
             H
         od.
         I
     od
```

The refinement structure corresponding to this loop is

P ⟸ A. **if** b **then** ok **else** C. Q **fi**

Q ⟸ D. **if** e **then** ok **else** F. **if** g **then** I. P **else** H. Q **fi fi**

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 191: 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 $i'=n$. The problem, except for time, is P:

P = **if** x: $A\ (0,..n)\ (0,..m)$ **then** $x = A\ i'\ j'$ **else** $i'=n$ **fi**

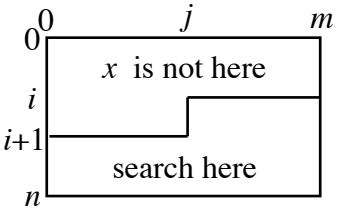
We may as well search row 0 first, then row 1, and so on. Accordingly, we define Q to specify the search from row i onward:

Q = **if** x: $A\ (i,..n)\ (0,..m)$ **then** $x = A\ i'\ j'$ **else** $i'=n$ **fi**

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: $A\ i\ (j,..m), A\ (i+1,..n)\ (0,..m)$ **then** $x = A\ i'\ j'$ **else** $i'=n$ **fi**

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** ok **else** $i{<}n \Rightarrow Q$ **fi**

$i{<}n \Rightarrow Q \Leftarrow j{:=}0.\ i{<}n \wedge j{\le}m \Rightarrow R$

$i{<}n \wedge j{\le}m \Rightarrow R \Leftarrow$ **if** $j{=}m$ **then** $i{:=}i{+}1.\ i{\le}n \Rightarrow Q$ **else** $i{<}n \wedge j{<}m \Rightarrow R$ **fi**

$i{<}n \wedge j{<}m \Rightarrow R \Leftarrow$ **if** $A\ i\ j = x$ **then** ok **else** $j{:=}j{+}1.\ i{<}n \wedge j{\le}m \Rightarrow R$ **fi**

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, after two uses of the Stepwise Refinement Law, the program appears as follows:

$P \Leftarrow i{:=}0.\ L0$
$L0 \Leftarrow$ **if** $i{=}n$ **then** ok **else** $j{:=}0.\ L1$ **fi**
$L1 \Leftarrow$ **if** $j{=}m$ **then** $i{:=}i{+}1.\ L0$
else if $A\ i\ j = x$ **then** ok
else $j{:=}j{+}1.\ L1$ **fi fi**

In C, this is

```
      i = 0;
L0:  if (i==n) ;
     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** ok **else** $i{<}n \Rightarrow t' \le t + (n{-}i){\times}m$ **fi**

$i{<}n \Rightarrow t' \le t + (n{-}i){\times}m \Leftarrow j{:=}0.\ i{<}n \wedge j{\le}m \Rightarrow t' \le t + (n{-}i){\times}m - j$

$i{<}n \wedge j{\le}m \Rightarrow t' \le t + (n{-}i){\times}m - j \Leftarrow$
$t{:=}t{+}1.$
if $j{=}m$ **then** $i{:=}i{+}1.\ i{\le}n \Rightarrow t' \le t + (n{-}i){\times}m$
else $i{<}n \wedge j{<}m \Rightarrow t' \le t + (n{-}i){\times}m - j$ **fi**

$i{<}n \wedge j{<}m \Rightarrow t' \le t + (n{-}i){\times}m - j \Leftarrow$
if $A\ i\ j = x$ **then** ok
else $j{:=}j{+}1.\ i{<}n \wedge j{\le}m \Rightarrow t' \le t + (n{-}i){\times}m - j$ **fi**

End of Two-Dimensional Search

5.2.3 For-Loop

Let us use the syntax

for i:= m;..n **do** P **od**

where i is a fresh name called the **for**-loop index, 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. The **for**-loop is indexed, and it refines a specification that is also indexed (a predicate). Specification $F\,i$ describes the computation from index i to the end. If the index runs from m to n then $F\,m$ describes the whole computation from the beginning to the end. To prove

$F\,m \Leftarrow$ **for** i:= m;..n **do** P **od**

prove

$F\,i \Leftarrow i{:}\ m,..n \ \wedge\ (P.\ F(i{+}1))$

$F\,n \Leftarrow ok$

The iterations from i to the end ($F\,i$) are implemented by doing one iteration (P) and then the iterations from $i{+}1$ to the end ($F(i{+}1)$). At the end ($F\,n$) there is nothing left to do (ok).

Let x be a natural variable and n be a natural constant. The binary exponentiation problem, Exercise 179, can be solved by some initialization and then a **for**-loop.

$x'{=}2^n \Leftarrow x{:=}\ 1.$ **for** i:= 0;..n **do** $x{:=}\ 2{\times}x$ **od**

To prove it, we need to find an indexed specification $F\,i$ such that

$x'{=}2^n \Leftarrow x{:=}\ 1.\ F\,0$

$F\,i \Leftarrow i{:}\ 0,..n \ \wedge\ (x{:=}\ 2{\times}x.\ F(i{+}1))$

$F\,n \Leftarrow ok$

The specification we want is

$F\,i = x' = x{\times}2^{n-i}$

which says that the final product is the product so far times the remaining factors. The three proofs are easy.

The time taken by the body of a **for**-loop may be a function f of the iteration i. To prove

$t' = t + \Sigma i{:}\ m,..n{\cdot}\ f\,i \Leftarrow$ **for** i:= m;..n **do** $t' = t + f\,i$ **od**

define

$F\,i = t' = t + \Sigma j{:}\ i,..n{\cdot}\ f\,j$

and prove

$t' = t + \Sigma i{:}\ m,..n{\cdot}\ f\,i \Leftarrow F\,m$

$F\,i \Leftarrow i{:}\ m,..n \ \wedge\ (t' = t + f\,i.\ F(i{+}1))$

$F\,n \Leftarrow ok$

all of which are easy. When the body takes constant time c, this simplifies to

$t' = t + (n–m){\times}c \Leftarrow$ **for** i:= m;..n **do** $t' = t{+}c$ **od**

A typical use of the **for**-loop rule is to do something to each item in a list. For example, Exercise 326 asks us to add 1 to each item in a list. The specification S is defined as

$S = \#L'{=}\#L \ \wedge\ \forall n{:}\ \Box L{\cdot}\ L'n = L\,n + 1$

Now we need a specification $F\,i$ that describes the iterations from i to the end: adding 1 to

each item from index i to (not including) $\#L$.

$$F\,i \;=\; \#L'{=}\#L \wedge (\forall n{:}\ 0,..i{\cdot}\ L'n = L\,n) \wedge (\forall n{:}\ i,..\#L{\cdot}\ L'n = L\,n+1)$$

Then $S = F\,0$. To prove

$$F\,0 \;\Longleftarrow\; \textbf{for}\ i{:=}\,0;..\#L\ \textbf{do}\ L{:=}\,i \rightarrow L\,i+1 \mid L\ \textbf{od}$$

we must prove two theorems:

$$F\,i \;\Longleftarrow\; i{:}\ 0,..\#L \ \wedge\ (L{:=}\,i \rightarrow L\,i+1 \mid L.\ F(i{+}1))$$

$$F\,(\#L) \;\Longleftarrow\; ok$$

Proof: First refinement:

$i{:}\ 0,..\#L \ \wedge\ (L{:=}\,i \rightarrow L\,i+1 \mid L.\ F(i{+}1))$ expand $F(i{+}1)$

$=$ $i{:}\ 0,..\#L \ \wedge\ (L{:=}\,i \rightarrow L\,i+1 \mid L.$

$\#L'{=}\#L \wedge (\forall n{:}\ 0,..i{+}1{\cdot}\ L'n = L\,n) \wedge (\forall n{:}\ i{+}1,..\#L{\cdot}\ L'n = L\,n+1))$

substitution law

$=$ $i{:}\ 0,..\#L \ \wedge\ \#L'{=}\#(i \rightarrow L\,i+1 \mid L) \wedge (\forall n{:}\ 0,..i{+}1{\cdot}\ L'n = (i \rightarrow L\,i+1 \mid L)\,n)$

$\wedge\ (\forall n{:}\ i{+}1,..\#(i \rightarrow L\,i+1 \mid L){\cdot}\ L'n = (i \rightarrow L\,i+1 \mid L)\,n+1)$

For $i{:}\ 0,..\#L$, $\#(i \rightarrow L\,i+1 \mid L){=}\#L$. Divide domain $0,..i{+}1$ into $0,..i$ and i .

For $n{:}\ 0,..i$, $(i \rightarrow L\,i+1 \mid L)\,n = L\,n$. For $n{:}\ i$, $(i \rightarrow L\,i+1 \mid L)\,n = L\,i+1$.

For $n{:}\ i{+}1,..\#L$, $(i \rightarrow L\,i+1 \mid L)\,n = L\,n$.

$=$ $i{:}\ 0,..\#L \ \wedge\ \#L'{=}\#L \wedge (\forall n{:}\ 0,..i{\cdot}\ L'n = L\,n) \wedge L'i = L\,i+1$

$\wedge\ (\forall n{:}\ i{+}1,..\#L{\cdot}\ L'n = L\,n+1)$ Combine i and $i{+}1,..\#L$.

$=$ $i{:}\ 0,..\#L \ \wedge\ \#L'{=}\#L \wedge (\forall n{:}\ 0,..i{\cdot}\ L'n = L\,n) \wedge (\forall n{:}\ i,..\#L{\cdot}\ L'n = L\,n+1)$

$=$ $F\,i$

Last refinement:

$F\,(\#L)$

$=$ $\#L'{=}\#L \wedge (\forall n{:}\ 0,..\#L{\cdot}\ L'n = L\,n) \wedge (\forall n{:}\ \#L,..\#L{\cdot}\ L'n = L\,n+1)$ *null* domain

$=$ $\#L'{=}\#L \wedge (\forall n{:}\ 0,..\#L{\cdot}\ L'n = L\,n)$

$\Longleftarrow$ ok

Sometimes the indexed specification for the **for**-loop has the form $A\,m \Rightarrow A'n$, where $A\,i$ is a binary expression in unprimed variables called an invariant, and $A'i$ is the same binary expression but with primed variables. This gives us a special case of the **for**-loop rule.

$$A\,m \Rightarrow A'n \;\Longleftarrow\; \textbf{for}\ i{:=}\,m;..n\ \textbf{do}\ i{:}\ m,..n \wedge A\,i \;\Rightarrow\; A'(i{+}1)\ \textbf{od}$$

A specification cannot always be put in this form; neither "do something to each element of a list" (Exercise 326) nor the loop timing examples can be. But when a specification can be put in this form, the benefit is that there is nothing to prove. As an example of the invariant **for**-loop rule, we do Exercise 226(a): Given a list of integers, possibly including negatives, write a program to find the minimum sum of any nonempty segment (sublist of consecutive items). Let constant L be the list, and let s be an integer variable. Formally, the problem P is defined as

$$P \;=\; s' = \Downarrow i{:}\ 0,..\#L{\cdot}\ \Downarrow j{:}\ i{+}1,..\#L{+}1{\cdot}\ \Sigma\,L\,[i;..j]$$

We take the minimum of all nonempty segment sums for all starting points i from 0 to but not including the end of L , and all ending points j from $i{+}1$ to and including the end of L . We will be using a **for**-loop indexed from 0 to $\#L$. At index k , s is the minimum sum of any nonempty segment we have seen so far, so that's any nonempty segment that ends at or before k . Define

$$A\,k \;=\; s = (\Downarrow i{:}\ 0,..k{\cdot}\ \Downarrow j{:}\ i{+}1,..k{+}1{\cdot}\ \Sigma\,L\,[i;..j])$$

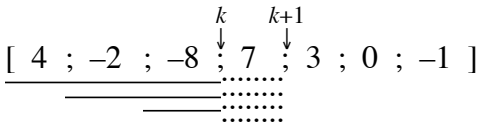
which is identical to P except that $\#L$ is replaced by k . When $k{=}0$, there are no nonempty segments ending at or before index 0 , and the minimum of an empty bunch is ∞ . So that is required initialization.

$$P \;\Longleftarrow\; s{:=}\,\infty.\ A\,0 \Rightarrow A'(\#L)$$

is easily proved. Now, for free, with no proof required, we have

$$A\,0 \Rightarrow A'(\#L) \;\Longleftarrow\; \textbf{for}\ k{:=}\,0;..\#L\ \textbf{do}\ k{:}\ 0,..\#L \wedge A\,k \;\Rightarrow\; A'(k{+}1)\ \textbf{od}$$

All that remains is to refine the loop body $k{:}\ 0,..\#L \wedge A\ k \Rightarrow A'(k{+}1)$, which takes us from index k to $k{+}1$. Here's the picture.



Variable s is the minimum sum of all nonempty segments ending at or before index k. As we move to index $k{+}1$, the new segments are one-item extensions of old segments, plus the new segment that consists of just the one new item $L\ k$. So we need to know the minimum sum of all nonempty segments ending exactly at index k. Call it c and add it to $A\ k$.

$$A\ k \ =\ s{=}(\Downarrow i{:}\ 0,..k{\cdot}\ \Downarrow j{:}\ i{+}1,..k{+}1{\cdot}\ \Sigma\ L\ [i;..j]) \wedge\ c{=}(\Downarrow i{:}\ 0,..k{\cdot}\ \Sigma\ L\ [i;..k])$$

When $k{=}0$, there are no nonempty segments ending at index 0, and the minimum of an empty bunch is ∞, so $c{:=}\ \infty$ must be added to the initialization.

$$P \ \Leftarrow\ s{:=}\ \infty.\ c{:=}\ \infty.\ A\ 0 \Rightarrow A'(\#L)$$

Now we can write the body of the loop. If we take the nonempty segments ending at k and extend them with item $L\ k$, the new minimum sum is $c + L\ k$. And there's one new segment consisting of just $L\ k$. So $c{:=}\ (c + L\ k){\downarrow}(L\ k)$. We can simplify: $(c + L\ k){\downarrow}(L\ k) = c{\downarrow}0 + L\ k$. Now c is the minimum sum of all new segments, so $s{:=}\ s{\downarrow}c$. The loop body is refined as

$$k{:}\ 0,..\#L \wedge A\ k \ \Rightarrow\ A'(k{+}1) \ \Leftarrow\ \ c{:=}\ c{\downarrow}0 + L\ k.\ \ s{:=}\ s{\downarrow}c$$

Altogether,

$$P \ \Leftarrow\ \ s{:=}\ \infty.\ \ c{:=}\ \infty.\ \ \textbf{for}\ k{:=}\ 0;..\#L\ \textbf{do}\ c{:=}\ c{\downarrow}0 + L\ k.\ \ s{:=}\ s{\downarrow}c\ \textbf{od}$$

End of For-Loop

5.2.4 Go To

Suppose the fast exponentiation program $z'{=}x^y$ of Subsection 4.2.6 Exercise 180 were written as follows, using $\wr$ for labeling the target of a **go to**:

$A\wr\ z{:=}\ 1.\ \ \textbf{if}\ \textit{even}\ y\ \textbf{then go to}\ C$
$\quad\textbf{else}\ B\wr\ z{:=}\ z{\times}x.\ \ y{:=}\ y{-}1.$
$\quad\quad C\wr\ \textbf{if}\ y{=}0\ \textbf{then go to}\ E$
$\quad\quad\quad\textbf{else}\ D\wr\ x{:=}\ x{\times}x.\ \ y{:=}\ y/2.$
$\quad\quad\quad\quad\textbf{if}\ \textit{even}\ y\ \textbf{then go to}\ D$
$\quad\quad\quad\quad\textbf{else go to}\ B\ \textbf{fi fi fi}.$
$E\wr\ \textit{ok}$

Straight from the program, what needs to be proved is the following:

$$A \ \Leftarrow\ \ z{:=}\ 1.\ \ \textbf{if}\ \textit{even}\ y\ \textbf{then}\ C\ \textbf{else}\ B\ \textbf{fi}$$
$$B \ \Leftarrow\ \ z{:=}\ z{\times}x.\ \ y{:=}\ y{-}1.\ \ C$$
$$C \ \Leftarrow\ \ \textbf{if}\ y{=}0\ \textbf{then}\ E\ \textbf{else}\ D\ \textbf{fi}$$
$$D \ \Leftarrow\ \ x{:=}\ x{\times}x.\ \ y{:=}\ y/2.\ \ \textbf{if}\ \textit{even}\ y\ \textbf{then}\ D\ \textbf{else}\ B\ \textbf{fi}$$
$$E \ \Leftarrow\ \ \textit{ok}$$

for appropriately defined A, B, C, D, and E. The difficulty with **go to**, as with loop constructs, is inventing specifications that were not recorded during program construction. In this example, the appropriate specifications are:

$$A \ =\ z'{=}x^y$$
$$B \ =\ \textit{odd}\ y \Rightarrow z'{=}z{\times}x^y$$
$$C \ =\ \textit{even}\ y \Rightarrow z'{=}z{\times}x^y$$
$$D \ =\ \textit{even}\ y \wedge y{>}0 \Rightarrow z'{=}z{\times}x^y$$
$$E \ =\ \textit{ok}$$

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 t 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**...**fi** . But the assignment $t := 5$ is not allowed. We can look at the clock, but not reset it arbitrarily; all assignments to t must correspond to the passage of time (according to some measure); otherwise t would not represent the time. (A computer's clock may need to be set 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 $=$ $t := t \uparrow w$

Because we are not allowed to reset the clock, $t := t \uparrow w$ is not acceptable as a program until we refine it by a program. Letting time be an extended natural and using recursive time,

wait until w $\Leftarrow$ **if** $t \ge w$ **then** ok **else** $t := t+1$. **wait until** w **fi**

and we obtain a busy-wait loop. We can prove this refinement by cases. First,

$t \ge w \wedge ok$
$=$ $t \ge w \wedge (t := t)$
$\Longrightarrow$ $t := t \uparrow w$

And second,

$t < w \wedge (t := t+1.\ t := t \uparrow w)$ expand assignment
$=$ $t < w \wedge (t := t+1.\ t' = t \uparrow w \wedge$ (other variables unchanged))
In the left conjunct, use $t : xnat$. In the right conjunct, use the Substitution Law.
$=$ $t+1 \le w \wedge (t' = (t+1) \uparrow w \wedge$ (other variables unchanged))
$=$ $t+1 \le w \wedge (t := (t+1) \uparrow w)$
$=$ $t+1 \le w \wedge (t := w)$
$=$ $t < w \wedge (t := t \uparrow w)$
$\Longrightarrow$ $t := t \uparrow w$

In programs that depend on 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 333(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

As a safety check, some programming languages include the notation

assert b

where b is binary, to mean something like "I believe b is true". If it comes at the beginning of a procedure or method, it may use the keyword **precondition** ; if it comes at the end, it may use the keyword **postcondition** ; if it comes at the start or end of a loop, it may use the keyword

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 can help to find errors and prevent subsequent damage to the state variables. But it's not free; it costs execution time. Assertions are a way to make programs more robust.

Assertions are defined as follows.

assert b = **if** b **then** ok **else** *screen*! "error". **wait until** ∞ **fi**

If b is true, **assert** b is the same as ok. If b is false, an error message is printed, and execution cannot proceed in finite time to any following actions.

5.4.0 Backtracking

If P and Q are implementable specifications, so is $P \vee Q$. The disjunction can be implemented by choosing one of P or Q and satisfying it. Normally this choice is made as a refinement, either $P \vee Q \Leftarrow P$ or $P \vee Q \Leftarrow Q$. 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 binary, to mean something like "make b true without changing anything". We define it as follows.

ensure b = **if** b **then** ok **else** $b' \wedge ok$ **fi** = $b' \wedge ok$

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 $\top$). 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** $x=1$

$= \quad \exists x'', y'' \cdot (x''=0 \wedge y''=y \ \vee \ x''=1 \wedge y''=y) \ \wedge \ x'=1 \wedge x'=x'' \wedge y'=y''$

$= \quad x'=1 \wedge y'=y$

$= \quad x := 1$

Although an implementation is given a choice between $x := 0$ and $x := 1$, it must choose the right one to satisfy a later binary expression. It can do so by making either choice (as usual), and when faced with a later **ensure** whose binary expression is $\bot$, it must backtrack and make another choice. Since choices can be nested within choices, a lot of bookkeeping is necessary.

Some 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. 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 Value Expression

Let P be a specification and e be an expression in unprimed variables. Then

P **value** e

is an expression of the initial state. It expresses the value that would be obtained by executing P and then evaluating e. The base of the natural logarithms can be expressed as follows.

new *term*, *sum*: *rat* := 1·
for *i*:= 1;..15 **do** *term*:= *term*/*i*. *sum*:= *sum*+*term* **od**
value *sum*

The scope of local variables *term* and *sum* extends to the end of the **value** expression.

The **value** expression axiom is

P. (P **value** e)=e

except that (P **value** e) is not subject to double-priming in sequential composition, nor to substitution when using the Substitution Law. For example,

	$\top$	use the **value** axiom
=	x:= x+1. (x:= x+1 **value** x)=x	use the Substitution Law, leaving the **value** expression unchanged
=	(x:= x+1 **value** x) = x+1	

The result is as if x:= x+1 were executed, then x is the result, except that the value of variable x is unchanged.

	y:= (x:= x+1 **value** x)	by the previous calculation
=	y:= x+1	

The expression P **value** 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 the modified P and evaluate the modified 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 **value** 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 **value** 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 with similar provisos,

x:= (P **value** e) + y becomes (**new** z:= y· P. x:= e+z)

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 (such as join) it is questionable. For **value** expressions containing loops, it is

unreasonable. But allowing a **value** expression to increase a time variable would be a side-effect, so here is what we do. We first include time in the **value** expression for the purpose of calculating a time bound. Then we remove the time variable from the **value** expression (to get rid of the side-effect) and we put a time increment in the program outside the **value** expression.

End of Value 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 **value** 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 *= 2;
   return r; }
```

In our notations, this would be

$$\begin{aligned} bexp \;=\; & \langle\, n\colon int\cdot \\ & \quad \textbf{new}\ r\colon int := 1\cdot \\ & \quad \textbf{for}\ i := 0;..n\ \textbf{do}\ r := r\times 2\ \textbf{od}. \\ & \quad \textbf{assert}\ r\colon int \\ & \quad \textbf{value}\ r\,\rangle \end{aligned}$$

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 **value** 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 115).

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

$$P \;=\; \langle x\colon int\cdot\ a' < x < b' \rangle$$

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

$$a' < x < b' \;\Longleftarrow\; a := x{-}1.\ \ b := x{+}1$$

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.

$\langle p{:}\ D{\cdot}\ B\rangle\ a \quad = \quad (\textbf{new}\ p{:}\ D := a{\cdot}\ B)$ if B doesn't use p' 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 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, called a variable parameter, or var parameter, or reference parameter, stands for a nonlocal variable to be supplied as argument. Here is an example, using **new** to introduce a variable parameter.

$\langle \textbf{new}\ x{:}\ int{\cdot}\ a{:=}\ 3.\ b{:=}\ 4.\ x{:=}\ 5\rangle\ a$

$=\quad a{:=}\ 3.\ b{:=}\ 4.\ a{:=}\ 5$

$=\quad a'{=}5 \wedge b'{=}4$

Variable 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

$\langle \textbf{new}\ x{:}\ int{\cdot}\ a'{=}3 \wedge b'{=}4 \wedge x'{=}5\rangle\ a$

we could not just replace x with a and 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,

$\langle \textbf{new}\ x{:}\ int{\cdot}\ x{:=}\ 5.\ b{:=}\ 4.\ a{:=}\ 3\rangle\ a$

$=\quad a{:=}\ 5.\ b{:=}\ 4.\ a{:=}\ 3$

$=\quad a'{=}3 \wedge b'{=}4$

but the result is different. Variable 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

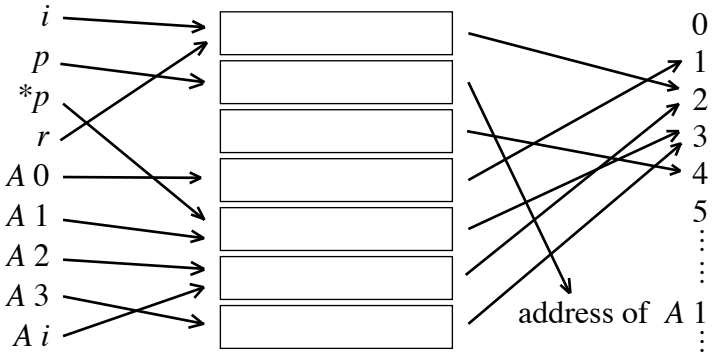
5.6 Alias optional

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.

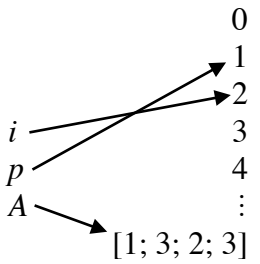
r, i	2
p	address of A 1
	4
A 0	1
$*p$, A 1	3
$A\ i$, A 2	2
A 3	3

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 variable 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 variable 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. Variable 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

$prob = §r: real \cdot 0 \le r \le 1$

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, binary 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 binary and integer variables; for rational and real variables, summations become integrals.) For example, if $n: nat+1$, then 2^{-n} is a distribution because

$(\forall n: nat+1 \cdot 2^{-n}: prob) \wedge (\Sigma n: nat+1 \cdot 2^{-n}) = 1$

If we have two variables $n, m: nat+1$, then 2^{-n-m} is a distribution because

$(\forall n, m: nat+1 \cdot 2^{-n-m}: prob) \wedge (\Sigma n, m: nat+1 \cdot 2^{-n-m}) = 1$

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

$\Sigma n, n' \cdot n' = n+1 = \infty$

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

$\Sigma n' \cdot n' = n+1 = 1$

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	=	$(x'=x) \times (y'=y) \times \ldots$
$x := e$	=	$(x'=e) \times (y'=y) \times \ldots$
if b **then** P **else** Q **fi**	=	$b \times P + (1-b) \times Q$
$P.Q$	=	$\Sigma x'', y'', \ldots \cdot$ (for $x', y', \ldots$ substitute $x'', y'', \ldots$ in P) $\times$ (for $x, y, \ldots$ substitute $x'', y'', \ldots$ 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 binary, the definitions of **if** b **then** P **else** Q **fi** 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 **fi** 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** $x:= 1$ **fi**

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,

$$\begin{array}{ll} & \textbf{if } 1/3 \textbf{ then } x:= 0 \textbf{ else } x:= 1 \textbf{ fi} \\ = & 1/3 \times (x'=0) + (1 - 1/3) \times (x'=1) \end{array}$$

Let us evaluate this expression using the value 0 for x'.

$$\begin{array}{ll} & 1/3 \times (0=0) + (1 - 1/3) \times (0=1) \\ = & 1/3 \times 1 + 2/3 \times 0 \\ = & 1/3 \end{array}$$

which is the probability that x has final value 0. Let us evaluate this expression using the value 1 for x'.

$$\begin{array}{ll} & 1/3 \times (1=0) + (1 - 1/3) \times (1=1) \\ = & 1/3 \times 0 + 2/3 \times 1 \\ = & 2/3 \end{array}$$

which is the probability that x has final value 1. Let us evaluate this expression using the value 2 for x'.

$$\begin{array}{ll} & 1/3 \times (2=0) + (1 - 1/3) \times (2=1) \\ = & 1/3 \times 0 + 2/3 \times 0 \\ = & 0 \end{array}$$

which is the probability that x has final value 2.

Here is a slightly more elaborate example in one variable x.

$$\begin{array}{ll} & \textbf{if } 1/3 \textbf{ then } x:= 0 \textbf{ else } x:= 1 \textbf{ fi}. \\ & \textbf{if } x=0 \textbf{ then if } 1/2 \textbf{ then } x:= x+2 \textbf{ else } x:= x+3 \textbf{ fi} \\ & \textbf{else if } 1/4 \textbf{ then } x:= x+4 \textbf{ else } x:= x+5 \textbf{ fi fi} \\ = & \Sigma x'' \cdot \quad ((x''=0)/3 + (x''=1)\times 2/3) \\ & \qquad \times \ (\quad (x''=0) \times ((x' = x''+2)/2 + (x' = x''+3)/2) \\ & \qquad\quad + \ (1 - (x''=0)) \times ((x' = x''+4)/4 + (x' = x''+5)\times 3/4)) \\ = & (x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2 \end{array}$$

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 <u>average</u> 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

$$\begin{array}{ll} & 2^{-n'}.\ n^2 \\ = & \Sigma n'' : nat+1 \cdot 2^{-n''} \times n''^2 \\ = & 6 \end{array}$$

After execution of the previous example, the average value of x is

if 1/3 **then** x:= 0 **else** x:= 1 **fi**.
if x=0 **then if** 1/2 **then** x:= x+2 **else** x:= x+3 **fi**
else if 1/4 **then** x:= x+4 **else** x:= x+5 **fi fi**.
x
$= \quad (x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2.\ x$
$= \quad \Sigma x''\cdot\ ((x''=2)/6 + (x''=3)/6 + (x''=5)/6 + (x''=6)/2)\ \times\ x''$
$= \quad 1/6 \times 2\ +\ 1/6 \times 3\ +\ 1/6 \times 5\ +\ 1/2 \times 6$
$= \quad 4 + 2/3$

Let P be any distribution of final states, and let b be any binary 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 binary expression. For example, after execution of the previous example, the probability that x is greater than 3 is

if 1/3 **then** x:= 0 **else** x:= 1 **fi**.
if x=0 **then if** 1/2 **then** x:= x+2 **else** x:= x+3 **fi**
else if 1/4 **then** x:= x+4 **else** x:= x+5 **fi fi**.
x>3
$= \quad (x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2.\ x>3$
$= \quad \Sigma x''\cdot\ ((x''=2)/6 + (x''=3)/6 + (x''=5)/6 + (x''=6)/2)\ \times\ (x''>3)$
$= \quad 1/6 \times (2>3)\ +\ 1/6 \times (3>3)\ +\ 1/6 \times (5>3)\ +\ 1/2 \times (6>3)$
$= \quad 2/3$

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: *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 $(r: 0,..n) / n$.

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 $(r: 0,..n) / n$. Or, if you prefer, we can replace *rand n* with variable $r: 0,..n$ 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,

x:= *rand* 2. x:= x + *rand* 3 replace one *rand* with r and one with s
= Σr: 0,..2· Σs: 0,..3· (x:= r)/2. (x:= x + s)/3 factor twice
= (Σr: 0,..2· Σs: 0,..3· (x:= r. x:= x + s))/6 replace final assignment, Substitution Law
= (Σr: 0,..2· Σs: 0,..3· (x' = r+s)) / 6 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

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: 0,..n) / n . For example, in one variable x ,

x:= *rand* 2. x:= x + *rand* 3 replace assignments
= (x': 0,..2)/2. (x': x+(0,..3))/3 sequential composition
= $\Sigma x''$· (x'': 0,..2)/2 × (x': x''+(0,..3))/3 sum
= 1/2 × (x': 0,..3)/3 + 1/2 × (x': 1,..4)/3
= (x'=0) / 6 + (x'=1) / 3 + (x'=2) / 3 + (x'=3) / 6

as before. And **if** *rand* 2 **then** A **else** B **fi** can be replaced by **if** 1/2 **then** A **else** B **fi** .

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 binary value b with distribution

Σr: 0,..8· (b = (r<3)) / 8
= (b=⊤) × 3/8 + (b=⊥) × 5/8
= 5/8 – b/4

which says that b is ⊤ with probability 3/8 , and ⊥ with probability 5/8 .

Exercise 344 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

x:= (*rand* 13) + 1. **if** x<7 **then** x:= x + (*rand* 13) + 1 **else** *ok* **fi**
replace *rand* and *ok*
= (x': (0,..13)+1)/13. **if** x<7 **then** (x': x+(0,..13)+1)/13 **else** x'=x **fi** replace . and **if**
= $\Sigma x''$· (x'': 1,..14)/13 × ((x''<7)×(x': x''+1,..x''+14)/13 + (x''≥7)×(x'=x''))
by several omitted steps
= ((2≤x'<7)×(x'–1) + (7≤x'<14)×19 + (14≤x'<20)×(20–x')) / 169

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 assertion that x wins:

c:= (*rand* 13) + 1. d:= (*rand* 13) + 1.
if $c < n$ **then** x:= c+d **else** x:= c **fi**. **if** $c < n$+1 **then** y:= c+d **else** y:= c **fi**.
$y<x\le 14 \vee x\le 14<y$ Replace *rand* and use the Functional-Imperative Law twice.

$=$ (c': (0,..13)+1 $\wedge$ d': (0,..13)+1 $\wedge$ $x'=x$ $\wedge$ $y'=y$) / 13 / 13.
x:= **if** $c < n$ **then** c+d **else** c **fi**. y:= **if** $c < n$+1 **then** c+d **else** c **fi**.
$y<x\le 14 \vee x\le 14<y$ Use the Substitution Law twice.

$=$ (c': (0,..13)+1 $\wedge$ d': (0,..13)+1 $\wedge$ $x'=x$ $\wedge$ $y'=y$) / 169.
if $c < n$+1 **then** c+d **else** c **fi** < **if** $c < n$ **then** c+d **else** c **fi** $\le$ 14
$\vee$ **if** $c < n$ **then** c+d **else** c **fi** $\le$ 14 < **if** $c < n$+1 **then** c+d **else** c **fi**

$=$ (c': (0,..13)+1 $\wedge$ d': (0,..13)+1 $\wedge$ $x'=x$ $\wedge$ $y'=y$) / 169. $c=n \wedge d>14–n$

$=$ $\Sigma c'', d'', x'', y''\cdot$
(c'': (0,..13)+1 $\wedge$ d'': (0,..13)+1 $\wedge$ $x''=x$ $\wedge$ $y''=y$) / 169 $\times$ ($c''=n \wedge d''>14–n$)

$=$ $\Sigma d''$: 1,..14$\cdot$ ($d''>14–n$)/169

$=$ (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 351 asks: If you repeatedly throw a pair of six-sided dice, how long does it take until the dice are equal? Using u and v for the dice and t for recursive time, the program is

$u'=v'$ $\Leftarrow$ u:= (*rand* 6) + 1. v:= (*rand* 6) + 1. **if** $u=v$ **then** *ok* **else** t:= t+1. $u'=v'$ **fi**

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:= (*rand* 6) + 1. v:= (*rand* 6) + 1. replace *rand* and
if $u=v$ **then** $t'=t$ **else** t:= t+1. $(t'\ge t) \times (5/6)^{t'-t} \times 1/6$ **fi** Substitution Law

$=$ (u': 1,..7 $\wedge$ $v'=v$ $\wedge$ $t'=t$)/6. ($u'=u$ $\wedge$ v': 1,..7 $\wedge$ $t'=t$)/6. replace first .
if $u=v$ **then** $t'=t$ **else** $(t'\ge t+1) \times (5/6)^{t'-t-1}$ / 6 **fi** and simplify

$=$ (u', v': 1,..7 $\wedge$ $t'=t$)/36. replace remaining .
if $u=v$ **then** $t'=t$ **else** $(t'\ge t+1) \times (5/6)^{t'-t-1}$ / 6 **fi** and replace **if**

$=$ $\Sigma u'', v''$: 1,..7$\cdot$ $\Sigma t''\cdot$ ($t''=t$)/36 $\times$ (($u''=v''$) $\times$ ($t'=t''$)
+ ($u''\neq v''$) $\times$ ($t'\ge t''+1$) $\times$ $(5/6)^{t'-t''-1}$ / 6) distribute

$=$ ($\Sigma u'', v''$: 1,..7$\cdot$ $\Sigma t''\cdot$ ($t''=t$)/36 $\times$ ($u''=v''$) $\times$ ($t'=t''$)) sum
+ ($\Sigma u'', v''$: 1,..7$\cdot$ $\Sigma t''\cdot$ ($t''=t$)/36 $\times$ ($u''\neq v''$) $\times$ ($t'\ge t''+1$) $\times$ $(5/6)^{t'-t''-1}$ / 6) sum

$=$ 6 $\times$ ($t'=t$) / 36 + 30 $\times$ ($t'\ge t+1$) $\times$ $(5/6)^{t'-t-1}$ / 6 / 36 simplify

$=$ ($t'=t$)/6 + ($t'\ge t+1$) $\times$ 5/6$\times$$(5/6)^{t'-t-1}$ / 6 simplify

$=$ ($t'=t$)/6 + ($t'\ge t+1$) $\times$ $(5/6)^{t'-t}$ / 6 combine

$=$ $(t'\ge t) \times (5/6)^{t'-t} \times 1/6$

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 binary 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 binary 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 **fi** 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 Subsection 4.2.5, 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 of Subsection 4.1.2, 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 *ok*, assignment, and sequential composition from our programming notations, and we add functions.

To illustrate, we look once again at the list summation problem (Exercise 174). This time, the specification is $\langle L\colon [*rat]\cdot\ \Sigma L\rangle$. 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.

$$\Sigma L = \langle n\colon 0,..\#L+1\cdot \Sigma L\ [n;..\#L]\rangle\ 0$$

It saves some copying to write "$\Sigma L = \ldots$" rather than "$\langle L\colon [*rat]\cdot\ \Sigma L\rangle = \ldots$", 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.

$$0,..\#L+1 \quad = \quad \square L, \#L$$

We divide the function into a selective union

$$\langle n\colon 0,..\#L+1\cdot \Sigma L\ [n;..\#L]\rangle \quad = \quad \langle n\colon \square L\cdot \Sigma L\ [n;..\#L]\rangle \mid \langle n\colon \#L\cdot \Sigma L\ [n;..\#L]\rangle$$

and continue with each part separately. In the left part, we have $n<\#L$, and in the right part $n=\#L$.

$$\langle n\colon \square L\cdot \Sigma L\ [n;..\#L]\rangle \quad = \quad \langle n\colon \square L\cdot L\ n + \Sigma L\ [n+1;..\#L]\rangle$$

$$\langle n\colon \#L\cdot \Sigma L\ [n;..\#L]\rangle \quad = \quad \langle n\colon \#L\cdot 0\rangle$$

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.

$$\Sigma L\ [n+1;..\#L] \quad = \quad \langle n\colon 0,..\#L+1\cdot \Sigma L\ [n;..\#L]\rangle\ (n+1)$$

In place of the selective union we could have used **if then else fi**; they are related by the law

$$\langle v\colon A\cdot x\rangle \mid \langle v\colon B\cdot y\rangle \quad = \quad \langle v\colon A, B\cdot \textbf{if } v\colon A \textbf{ then } x \textbf{ else } y \textbf{ fi}\rangle$$

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 $\langle L\colon [*rat]\cdot \#L\rangle$, 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.

$$\#L = \langle n\colon 0,..\#L+1\cdot \#L–n\rangle\ 0$$

The second step was to decompose the function into a selective union; we do so again.

$$\langle n\colon 0,..\#L+1\cdot \#L–n\rangle \quad = \quad \langle n\colon \square L\cdot \#L–n\rangle \mid \langle n\colon \#L\cdot \#L–n\rangle$$

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.

$$\langle n\colon \square L\cdot \#L–n\rangle \quad = \quad \langle n\colon \square L\cdot 1 + \#L–n–1\rangle$$

The right side of the selective union became a function with a constant result; according to our measure, its time must be 0.

$$\langle n\colon \#L\cdot \#L–n\rangle \quad = \quad \langle n\colon \#L\cdot 0\rangle$$

The remaining problem was solved by a recursive call; the corresponding call solves the remaining time problem.

$$\#L–n–1 \quad = \quad \langle n\colon 0,..\#L+1\cdot \#L–n\rangle\ (n+1)$$

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 $\langle L\colon [*rat]\cdot \#L\rangle$.

$$\#L \ = \ \langle n\colon 0,..\#L+1\cdot \#L–n\rangle\ 0$$
$$\langle n\colon 0,..\#L+1\cdot \#L–n\rangle \quad = \quad \langle n\colon \square L\cdot \#L–n\rangle \mid \langle n\colon \#L\cdot \#L–n\rangle$$
$$\langle n\colon \square L\cdot \#L–n\rangle \quad = \quad \langle n\colon \square L\cdot \#L–n\rangle$$
$$\langle n\colon \#L\cdot \#L–n\rangle \quad = \quad \langle n\colon \#L\cdot 0\rangle$$
$$\#L–n \quad = \quad 1 + \langle n\colon 0,..\#L+1\cdot \#L–n\rangle\ (n+1)$$

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 :	$\notin S\,x \ < \ 1$
Functional specification S is satisfiable for domain element x :	$\notin S\,x \ \geq \ 1$
Functional specification S is deterministic for domain element x :	$\notin S\,x \ \leq \ 1$
Functional specification S is nondeterministic for domain element x :	$\notin S\,x \ > \ 1$
Functional specification S is satisfiable for domain element x :	$\exists y\cdot y\colon S\,x$
Functional specification S is implementable:	$\forall x\cdot \exists y\cdot y\colon S\,x$

(x is quantified over the domain of S , and y is quantified over the range of S .) Implementability can be restated as $\forall x\cdot S\,x \neq null$.

Consider the problem of searching for an item in a list of integers. Our first attempt at specification might be

$$\langle L\colon [*int]\cdot \langle x\colon int\cdot \S n\colon \square L\cdot L\,n = x\rangle\rangle$$

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.

$$\langle L\colon [*int]\cdot \langle x\colon int\cdot \textbf{if } x\colon L\ (\square L) \textbf{ then } \S n\colon \square L\cdot L\,n = x \textbf{ else } \#L,..\infty \textbf{ fi}\rangle\rangle$$

This specification is implementable, and often nondeterministic.

Functional refinement is similar to imperative refinement. An imperative specification is a binary expression, and imperative refinement is reverse implication. Functional specification is a function, and functional refinement is function inclusion. 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. 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: P the other way round. Inclusion is antisymmetric, so its symbol should not be symmetric, but unfortunately it is. We write :: for "backward colon", so that "P is refined by S" is written P:: 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.

if x: L ($\Box L$) **then** §n: $\Box L\cdot$ $L\,n = x$ **else** $\#L$,..∞ **fi**::
    $\langle i$: $nat\cdot$ **if** x: L (i,..$\#L$) **then** §n: i,..$\#L\cdot$ $L\,n = x$ **else** $\#L$,..∞ **fi**$\rangle$ 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 fi** .

if x: L (i,..$\#L$) **then** §n: i,..$\#L\cdot$ $L\,n = x$ **else** $\#L$,..∞ **fi**::
    if $i = \#L$ **then** $\#L$
    else **if** $x = L\,i$ **then** i
        else $\langle i$: $nat\cdot$ **if** x: L (i,..$\#L$) **then** §n: i,..$\#L\cdot$ $L\,n = x$ **else** $\#L$,..∞ **fi**$\rangle$ (i+1) **fi fi**

The timing specification, recursive measure, is $\langle L\cdot \langle x\cdot\ 0,..\#L{+}1\rangle\rangle$, which means that the time is less than $\#L$+1 . To prove that this is the execution time, we must prove

0,..$\#L$+1:: $\langle i$: $nat\cdot$ 0,..$\#L$–i+1$\rangle$ 0

and

0,..$\#L$–i+1:: **if** $i = \#L$ **then** 0
    else **if** $x = L\,i$ **then** 0
        else 1 + $\langle i$: $nat\cdot$ 0,..$\#L$–i+1$\rangle$ (i+1) **fi fi**

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

$\langle v$: $D\cdot$ $b\rangle$ x = (for v substitute x in b)

At the heart of imperative programming we have the Substitution Law

x:= e. P = (for x substitute e in P)

Functional programming and imperative programming differ mainly in the notations 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 Section 2.0 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.

	⊤	by the axiom, 0: *nat*
⇒	0: *nat*	add 1 to each side
⇒	0+1: *nat*+1	by arithmetic, 0+1 = 1 ; by the axiom, *nat*+1: *nat*
⇒	1: *nat*	add 1 to each side
⇒	1+1: *nat*+1	by arithmetic, 1+1 = 2 ; by the axiom, *nat*+1: *nat*
⇒	2: *nat*	

and so on.

From the construction axioms we can prove 2: *nat* but we cannot prove ¬ –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* ∧ *B*+1: *B* ⇒ *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* ⇒ *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*→*bin* ,

P 0 ∧ ∀*n*: *nat*· *P n* ⇒ *P* (*n*+1) ⟹ ∀*n*: *nat*· *P n*

We prove first that the bunch form implies the predicate form.

	0: *B* ∧ *B*+1: *B* ⇒ *nat*: *B*	Let *B* = §*n*: *nat*· *P n* . Then *B*: *nat* ,
⟹	0: *B* ∧ (∀*n*: *nat*· *n*: *B* ⇒ *n*+1: *B*) ⇒ ∀*n*: *nat*· *n*: *B*	and ∀*n*: *nat*· (*n*: *B*) = *P n* .
=	*P* 0 ∧ (∀*n*: *nat*· *P n* ⇒ *P* (*n*+1)) ⇒ ∀*n*: *nat*· *P n*	

The reverse is proved similarly.

$P\,0 \wedge (\forall n{:}\ nat{\cdot}\ P\,n \Rightarrow P\,(n{+}1)) \ \Rightarrow\ \forall n{:}\ nat{\cdot}\ P\,n$

For arbitrary bunch B, let $P = \langle n{:}\ nat{\cdot}\ n{:}\ B\rangle$. Then again $\forall n{:}\ nat{\cdot}\ P\,n = (n{:}\ B)$.

$\Rightarrow\ 0{:}\ B \ \wedge\ (\forall n{:}\ nat{\cdot}\ n{:}\ B \ \Rightarrow\ n{+}1{:}\ B) \ \Rightarrow\ \forall n{:}\ nat{\cdot}\ n{:}\ B$

$=\ \ 0{:}\ B \ \wedge\ (\forall n{:}\ nat`B{\cdot}\ n{+}1{:}\ B) \ \Rightarrow\ \forall n{:}\ nat{\cdot}\ n{:}\ B$

$=\ \ 0{:}\ B \ \wedge\ (nat`B){+}1{:}\ B \ \Rightarrow\ nat{:}\ B$

$\Rightarrow\ 0{:}\ B \ \wedge\ B{+}1{:}\ B \ \Rightarrow\ nat{:}\ B$

Therefore the bunch and predicate forms of *nat* induction are equivalent.

The predicate form of *nat* construction can be stated as follows: If $P{:}\ nat{\rightarrow}bin$,

$P\,0 \wedge \forall n{:}\ nat{\cdot}\ P\,n \Rightarrow P\,(n{+}1) \ \Leftarrow\ \forall n{:}\ nat{\cdot}\ P\,n$

This is the same as induction but with the main implication reversed. We prove first that the bunch form implies the predicate form.

$\forall n{:}\ nat{\cdot}\ P\,n$ domain change using *nat* construction, bunch version

$\Rightarrow\ \forall n{:}\ 0, nat{+}1{\cdot}\ P\,n$ axiom about $\forall$

$=\ \ (\forall n{:}\ 0{\cdot}\ P\,n) \wedge (\forall n{:}\ nat{+}1{\cdot}\ P\,n)$ One-Point Law and variable change

$=\ \ P\,0 \wedge \forall n{:}\ nat{\cdot}\ P\,(n{+}1)$

$\Rightarrow\ P\,0 \wedge \forall n{:}\ nat{\cdot}\ P\,n \Rightarrow P\,(n{+}1)$

And now we prove that the predicate form implies the bunch form.

$P\,0 \wedge (\forall n{:}\ nat{\cdot}\ P\,n \Rightarrow P\,(n{+}1)) \ \Leftarrow\ \forall n{:}\ nat{\cdot}\ P\,n$ Let $P = \langle n{:}\ nat{\cdot}\ n{:}\ nat\rangle$

$\Rightarrow\ 0{:}\ nat \wedge (\forall n{:}\ nat{\cdot}\ n{:}\ nat \ \Rightarrow\ n{+}1{:}\ nat) \ \Leftarrow\ \forall n{:}\ nat{\cdot}\ n{:}\ nat$

$=\ \ 0{:}\ nat \wedge (\forall n{:}\ nat{\cdot}\ n{+}1{:}\ nat) \ \Leftarrow\ \top$

$=\ \ 0{:}\ nat \ \wedge\ nat{+}1{:}\ nat$

A corollary is that *nat* can be defined by the single axiom

$P\,0 \wedge \forall n{:}\ nat{\cdot}\ P\,n \Rightarrow P\,(n{+}1) \ =\ \forall n{:}\ nat{\cdot}\ P\,n$

There are other predicate versions of induction; here is the usual one again plus three more.

$P\,0 \wedge \forall n{:}\ nat{\cdot}\ P\,n \Rightarrow P\,(n{+}1) \ \Rightarrow\ \forall n{:}\ nat{\cdot}\ P\,n$

$P\,0 \vee \exists n{:}\ nat{\cdot}\ \neg\, P\,n \wedge P\,(n{+}1) \ \Leftarrow\ \exists n{:}\ nat{\cdot}\ P\,n$

$\forall n{:}\ nat{\cdot}\ P\,n \Rightarrow P\,(n{+}1) \ \Rightarrow\ \forall n{:}\ nat{\cdot}\ P\,0 \Rightarrow P\,n$

$\exists n{:}\ nat{\cdot}\ \neg\, P\,n \wedge P\,(n{+}1) \ \Leftarrow\ \exists n{:}\ nat{\cdot}\ \neg\, P\,0 \wedge P\,n$

The first version says that to prove P of all naturals, prove P of 0, and assuming P of natural n, prove P 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.

$\forall n{:}\ nat{\cdot}\ (\forall m{:}\ nat{\cdot}\ m{<}n \Rightarrow P\,m) \Rightarrow P\,n \ \Rightarrow\ \forall n{:}\ nat{\cdot}\ P\,n$

$\exists n{:}\ nat{\cdot}\ (\forall m{:}\ nat{\cdot}\ m{<}n \Rightarrow \neg\, P\,m) \wedge P\,n \ \Leftarrow\ \exists n{:}\ nat{\cdot}\ P\,n$

The first is like the first version of induction, except that the base case $P\,0$ 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 property P).

Proof by induction does not require any special notation or format. For example, Exercise 368 asks us to prove that the square of an odd natural number is $8\times m + 1$ for some natural m. Quantifying over nat,

$$\begin{array}{lll}
 & \forall n\cdot \exists m\cdot (2\times n + 1)^2 = 8\times m + 1 & \text{various number laws} \\
= & \forall n\cdot \exists m\cdot 4\times n\times(n+1) + 1 = 8\times m + 1 & \text{various number laws} \\
= & \forall n\cdot \exists m\cdot n\times(n+1) = 2\times m & \text{the usual predicate form of induction} \\
\Longleftarrow & \quad (\exists m\cdot 0\times(0+1) = 2\times m) & \text{generalization and} \\
 & \wedge\ (\forall n\cdot (\exists m\cdot n\times(n+1) = 2\times m) \Rightarrow (\exists l\cdot (n+1)\times(n+2) = 2\times l)) & \text{antidistribution} \\
\Longleftarrow & \quad 0\times(0+1) = 2\times 0 & \text{arithmetic and} \\
 & \wedge\ (\forall n, m\cdot n\times(n+1) = 2\times m \Rightarrow (\exists l\cdot (n+1)\times(n+2) = 2\times l)) & \text{generalization} \\
\Longleftarrow & \forall n, m\cdot n\times(n+1) = 2\times m \Rightarrow (n+1)\times(n+2) = 2\times(m+n+1) & \text{various number laws} \\
= & \top &
\end{array}$$

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

$$pow = \S p: nat\cdot \exists m: nat\cdot p = 2^m$$

But let us do it the same way we defined nat. The pow construction axiom is

$$1, 2\times pow: pow$$

and the pow induction axiom is

$$1, 2\times B: B \Rightarrow pow: B$$

Induction is not just for nat. In predicate form, we can define pow with the axiom

$$P\,1 \wedge \forall p: pow\cdot P\,p \Rightarrow P\,(2\times p) = \forall p: pow\cdot P\,p$$

We can define the bunch of integers as

$$int = nat, -nat$$

or equivalently we can use the construction and induction axioms

$$0, int+1, int-1: int$$

$$0, B+1, B-1: B \Rightarrow int: B$$

or we can use the axiom

$$P\,0 \wedge (\forall i: int\cdot P\,i \Rightarrow P\,(i+1)) \wedge (\forall i: int\cdot P\,i \Rightarrow P\,(i-1)) = \forall i: int\cdot P\,i$$

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

$$1, rat+rat, rat-rat, rat\times rat, rat/(\S r: rat\cdot r\neq 0): rat$$

$$1, B+B, B-B, B\times B, B/(\S b: B\cdot b\neq 0): B \Rightarrow rat: B$$

or with the axiom

$$\begin{array}{ll}
 & P\,1 \\
 & \wedge\ (\forall r, s: rat\cdot P\,r \wedge P\,s \Rightarrow P\,(r+s)) \\
 & \wedge\ (\forall r, s: rat\cdot P\,r \wedge P\,s \Rightarrow P\,(r-s)) \\
 & \wedge\ (\forall r, s: rat\cdot P\,r \wedge P\,s \Rightarrow P\,(r\times s)) \\
 & \wedge\ (\forall r, s: rat\cdot P\,r \wedge P\,s \wedge s\neq 0 \Rightarrow P\,(r/s)) \\
= & \forall r: rat\cdot P\,r
\end{array}$$

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 Section 2.0. 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

Or we can write them using reverse inclusion

$nat:: 0, nat{+}1$ *nat* construction

$B:: 0, B{+}1 \Rightarrow nat: B$ *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: $x = f\ x$. A least fixed-point of f is a smallest such x. Fixed-point construction has the form

$$name = \text{(expression involving } name\text{)}$$

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 $0, nat{+}1$

$\Longrightarrow$ $0, (0, nat{+}1){+}1: 0, nat{+}1 \Rightarrow nat: 0, nat{+}1$ strengthen the antecedent by cancelling the "0"s and "+1"s from the two sides of the first ":"

$\Longrightarrow$ $0, nat{+}1: nat \Rightarrow nat: 0, nat{+}1$ 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,

$$exp = \text{"x"},\ exp; \text{"+"}; exp$$

In this context, union is usually denoted by | and join 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 solutions from constructors. It usually works, but not always. We seek a solution of either of the following:

name:: (expression involving *name*)
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 a solution, but not always, so we must test it. We test one of the following:
$name_\infty$ :: (expression involving $name_\infty$)
$name_\infty$ = (expression involving $name_\infty$)

4. If we want the smallest solution, then we test one of the following:
B:: (expression involving B) $\Longrightarrow$ $name_\infty$: B
B = (expression involving B) $\Longrightarrow$ $name_\infty$: B

We illustrate recursive construction on *pow*. Its fixed-point construction and induction axioms are

$pow = 1, 2\times pow$
$B = 1, 2\times B \Rightarrow pow: B$

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
$$\begin{aligned} pow_\infty &= 2^{0,..\infty} \\ &= 2^{nat} \end{aligned}$$
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.
$$\begin{aligned} & 2^{nat} = 1, 2{\times}2^{nat} \\ = \quad & 2^{nat} = 2^0, 2^{1}{\times}2^{nat} \\ = \quad & 2^{nat} = 2^0, 2^{1+nat} \\ = \quad & 2^{nat} = 2^{0,\,1+nat} \\ \Leftarrow \quad & nat = 0, nat{+}1 && nat \text{ fixed-point construction} \\ = \quad & \top \end{aligned}$$

4. We must test pow_∞ to see if it is the least fixed-point.
$$\begin{aligned} & 2^{nat}{:}\ B \\ = \quad & \forall n{:}\ nat{\cdot}\ 2^n{:}\ B && \text{use the predicate form of } nat \text{ induction} \\ \Leftarrow \quad & 2^0{:}\ B \ \wedge\ \forall n{:}\ nat{\cdot}\ 2^n{:}\ B \ \Rightarrow\ 2^{n+1}{:}\ B && \text{change variable} \\ = \quad & 1{:}\ B \ \wedge\ \forall m{:}\ 2^{nat}{\cdot}\ m{:}\ B \ \Rightarrow\ 2{\times}m{:}\ B && \text{increase domain} \\ \Leftarrow \quad & 1{:}\ B \ \wedge\ \forall m{:}\ nat{\cdot}\ m{:}\ B \ \Rightarrow\ 2{\times}m{:}\ B && \text{Domain Change Law} \\ = \quad & 1{:}\ B \ \wedge\ \forall m{:}\ nat`B{\cdot}\ 2{\times}m{:}\ B && \text{increase domain} \\ \Leftarrow \quad & 1{:}\ B \ \wedge\ \forall m{:}\ B{\cdot}\ 2{\times}m{:}\ B \\ = \quad & 1{:}\ B \ \wedge\ 2{\times}B{:}\ B \\ \Leftarrow \quad & B = 1, 2{\times}B \end{aligned}$$
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 smallest solution, or some other solution.

In step 2, from $name_n$ we obtain a candidate $name_\infty$ for a solution 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
$$\S x{\cdot}\ \Updownarrow n{\cdot}\ x{:}\ name_n$$
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 solution. 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

$$bad \;=\; §n\colon nat\cdot \neg\, n\colon bad$$

an axiom. Thus bad is defined as the bunch of all naturals that are not in bad. From this axiom we find

$$\begin{aligned} & 0\colon bad \\ =\;& 0\colon §n\colon nat\cdot \neg\, n\colon bad \\ =\;& \neg\, 0\colon bad \end{aligned}$$

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 396).

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

$$zap \;=\; \textbf{if } x{=}0 \textbf{ then } y{:=}\,0 \textbf{ else } x{:=}\,x{-}1.\;\; t{:=}\,t{+}1.\;\; zap \textbf{ fi}$$

is given as an axiom. The right side of the equation is the constructor. Here are six solutions to this equation, and a picture of the relationships among the solutions.

(a) $x{\geq}0 \;\Rightarrow\; x'{=}y'{=}0 \wedge t' = t{+}x$

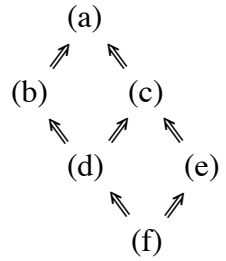
(b) $\textbf{if } x{\geq}0 \textbf{ then } x'{=}y'{=}0 \wedge t' = t{+}x \textbf{ else } t'{=}\infty \textbf{ fi}$

(c) $x'{=}y'{=}0 \wedge (x{\geq}0 \Rightarrow t' = t{+}x)$

(d) $x'{=}y'{=}0 \wedge \textbf{if } x{\geq}0 \textbf{ then } t' = t{+}x \textbf{ else } t'{=}\infty \textbf{ fi}$

(e) $x'{=}y'{=}0 \wedge t' = t{+}x$

(f) $x{\geq}0 \wedge x'{=}y'{=}0 \wedge t' = t{+}x$



Solution (a) is the weakest and solution (f) is the strongest, although the solutions are not totally ordered. 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

$$x{\geq}0 \Rightarrow x'{=}y'{=}0 \wedge t' = t{+}x \;\Leftarrow\; zap$$

so we can use it to solve problems. And it is refined by its constructor

$$zap \;\Leftarrow\; \textbf{if } x{=}0 \textbf{ then } y{:=}\,0 \textbf{ else } x{:=}\,x{-}1.\;\; t{:=}\,t{+}1.\;\; zap \textbf{ fi}$$

so we can execute it. For all practical purposes, that is all we need. But if we want to define zap as solution (a), we can do so by adding a fixed-point induction axiom

$$\begin{aligned} & \forall \sigma, \sigma'\cdot (Z \;=\; \textbf{if } x{=}0 \textbf{ then } y{:=}\,0 \textbf{ else } x{:=}\,x{-}1.\;\; t{:=}\,t{+}1.\;\; Z \textbf{ fi}) \\ \Rightarrow\;& \forall \sigma, \sigma'\cdot zap \;\Leftarrow\; Z \end{aligned}$$

This induction axiom says: if any specification Z satisfies the construction axiom, then zap is weaker than or equal to Z. So zap is the weakest solution of the construction axiom.

Although we defined zap using fixed-point construction and induction, we could have defined it equivalently using ordinary construction and induction.

if x=0 **then** y:= 0 **else** x:= x–1. t:= t+1. zap **fi** $\Leftarrow$ zap
  $\forall\sigma, \sigma' \cdot$ (**if** x=0 **then** y:= 0 **else** x:= x–1. t:= t+1. Z **fi** $\Leftarrow$ Z)
$\Longrightarrow$ $\forall\sigma, \sigma' \cdot zap \Leftarrow Z$

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 (defined either by ordinary construction and induction or by fixed-point construction and induction). 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.

$zap_1 =$ **if** x=0 **then** y:= 0 **else** x:= x–1. t:= t+1. zap_0 **fi**
  $= 0 \le x < 1 \Rightarrow x' = y' = 0 \wedge t' = t$
$zap_2 =$ **if** x=0 **then** y:= 0 **else** x:= x–1. t:= t+1. zap_1 **fi**
  $= 0 \le x < 2 \Rightarrow x' = y' = 0 \wedge t' = t + x$

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)

$zap_n = 0 \le x < n \Rightarrow x' = y' = 0 \wedge t' = t + x$

The next step is to replace n with ∞.

$zap_\infty = 0 \le x < \infty \Rightarrow x' = y' = 0 \wedge t' = t + x$

Finally, we must test the result to see if it satisfies the axiom.

  (right side of equation with zap_∞ for zap)
$=$ **if** x=0 **then** y:= 0 **else** x:= x–1. t:= t+1. $0 \le x \Rightarrow x' = y' = 0 \wedge t' = t + x$ **fi**
$=$ **if** x=0 **then** $x' = y' = 0 \wedge t' = t$ **else** $0 \le x - 1 \Rightarrow x' = y' = 0 \wedge t' = t + x$ **fi**
$=$ $0 \le x \Rightarrow x' = y' = 0 \wedge t' = t + x$
$=$ (left side of equation with zap_∞ for zap)

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.

$zap_0 = t' \ge t$
$zap_1 =$ **if** x=0 **then** y:= 0 **else** x:= x–1. t:= t+1. zap_0 **fi**
  $=$ **if** $0 \le x < 1$ **then** $x' = y' = 0 \wedge t' = t$ **else** $t' \ge t + 1$ **fi**
$zap_2 =$ **if** x=0 **then** y:= 0 **else** x:= x–1. t:= t+1. zap_1 **fi**
  $=$ **if** x=0 **then** $x' = y' = 0 \wedge t' = t$ **else if** x=1 **then** $x' = y' = 0 \wedge t' = t + 1$ **else** $t' \ge t + 2$ **fi fi**
  $=$ **if** $0 \le x < 2$ **then** $x' = y' = 0 \wedge t' = t + x$ **else** $t' \ge t + 2$ **fi**

In general, zap_n describes what we know up to time n. We can now guess (and prove using nat induction if we want)

$zap_n =$ **if** $0 \le x < n$ **then** $x' = y' = 0 \wedge t' = t + x$ **else** $t' \ge t + n$ **fi**

We replace n with ∞

zap_∞ = **if** $0 \le x$ **then** $x'=y'=0 \wedge t'=t+x$ **else** $t'=\infty$ **fi**

and test the result

(right side of equation with zap_∞ for zap)

= **if** $x=0$ **then** $y:= 0$ **else** $x:= x-1.\ t:= t+1.$ **if** $0 \le x$ **then** $x'=y'=0 \wedge t'=t+x$ **else** $t'=\infty$ **fi fi**

= **if** $x=0$ **then** $x'=y'=0 \wedge t'=t$ **else if** $0 \le x-1$ **then** $x'=y'=0 \wedge t'=t+x$ **else** $t'=\infty$ **fi fi**

= **if** $0 \le x$ **then** $x'=y'=0 \wedge t' = t+x$ **else** $t'=\infty$ **fi**

= (left side of equation with zap_∞ for zap)

Beginning our recursive construction with $t' \ge t$, we have constructed a stronger but still implementable fixed-point. We can choose any specification to start with, and we may obtain different fixed-points. 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 $\Updownarrow n\cdot zap_n$. 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

$t' \ge t$ $\Leftarrow$ **while** b **do** P **od**

if b **then** $P.\ t:= t+1.$ **while** b **do** P **od else** ok **fi** $\Leftarrow$ **while** b **do** P **od**

$\forall \sigma, \sigma' \cdot\ t' \ge t \wedge$ **if** b **then** $P.\ t:= t+1.\ W$ **else** ok **fi** $\Leftarrow W$

$\Longrightarrow$ $\forall \sigma, \sigma' \cdot$ **while** b **do** P **od** $\Leftarrow W$

Recursive timing has been included, but this can be changed to any other timing policy. These three axioms are closely analogous to the axioms

$0: nat$

$nat+1: nat$

$0, B+1: B \ \Rightarrow\ nat: B$

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 **od** 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

while b **do** P **od** = $t' \ge t \wedge$ **if** b **then** $P.\ t:= t+1.$ **while** b **do** P **od else** ok **fi**

$\forall \sigma, \sigma' \cdot (W =\ t' \ge t \wedge$ **if** b **then** $P.\ t:= t+1.\ W$ **else** ok **fi**)

$\Longrightarrow$ $\forall \sigma, \sigma' \cdot$ **while** b **do** P **od** $\Leftarrow W$

This account differs from that presented in Section 5.2; we have gained some theorems, and also lost some theorems. For example, from this least fixed-point definition, we cannot prove

$x' \ge x$ $\Leftarrow$ **while** b **do** $x' \ge x$ **od**

which was easily proved according to Section 5.2. The account of loops presented in Section 5.2 is best for practical purposes: it tells us how we can use them in programming.

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 binary values, 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*→*X*→*stack*
pop: *stack*→*stack*
top: *stack*→*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*
$P\ empty \wedge \forall s{:}\ stack{\cdot}\ \forall x{:}\ X{\cdot}\ P\ s \Rightarrow P\ (push\ s\ x) \iff \forall s{:}\ stack{\cdot}\ P\ s$

where *push* is allowed to distribute over bunch union, and P: *stack*→*bin* . 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*: $B \Rightarrow stack{:}\ B$
$P\ empty \wedge \forall s{:}\ stack{\cdot}\ \forall x{:}\ X{\cdot}\ P\ s \Rightarrow P\ (push\ s\ x) \Longrightarrow \forall s{:}\ stack{\cdot}\ P\ s$

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 ≠ *empty*

push s x = *push t y* = *s*=*t* ∧ *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 = ⟨*s*: *stack*· ⟨*x*: *int*· *s*;;[*x*]⟩⟩

pop = ⟨*s*: *stack*· **if** *s*=*empty* **then** *empty* **else** *s* [0;..#*s*–1] **fi**⟩

top = ⟨*s*: *stack*· **if** *s*=*empty* **then** 0 **else** *s* (#*s*–1) **fi**⟩

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* (⟨*s*: *stack*· ⟨*x*: *int*· *s*;;[*x*]⟩⟩ *s x*) = *x*	apply function
=	*top* (*s*;;[*x*]) = *x*	replace *top*
=	⟨*s*: *stack*· **if** *s*=*empty* **then** 0 **else** *s* (#*s*–1) **fi**⟩ (*s*;;[*x*]) = *x*	apply function
=	**if** *s*;;[*x*]=*empty* **then** 0 **else** (*s*;;[*x*]) (#(*s*;;[*x*])–1) **fi** = *x*	replace *empty*
=	**if** *s*;;[*x*]=[*nil*] **then** 0 **else** (*s*;;[*x*]) (#(*s*;;[*x*])–1) **fi** = *x*	simplify the **if** and the index
=	(*s*;;[*x*]) (#*s*) = *x*	index the list
=	*x*=*x*	reflexive law
=	⊤	

End of Data-Stack Implementation

Is stack theory consistent? Since we implemented it using list and function theory, we know that if list and function theory are consistent, so is stack theory. Is stack theory complete? To show that a binary 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 = ⟨*s*: *stack*· **if** *s*=*empty* **then** *push empty* 0 **else** *s* [0;..#*s*–1] **fi**⟩

top = ⟨*s*: *stack*· **if** *s*=*empty* **then** 1 **else** *s* (#*s*–1) **fi**⟩

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*→*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*→*stack* and remove the implementer's obligation to provide something that is not wanted. The weaker axiom

s≠*empty* ⟹ *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*→*X* we can prove *top empty*: *X* ; deleting the axiom *top*: *stack*→*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* ≠ *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 ≠ *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 (but not so strong as to cause inconsistency). 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*→X→*queue*

To say that the constructors construct distinct queues, with no repetitions, we need

join q x ≠ *emptyq*
join q x = *join r y* = $q{=}r \wedge x{=}y$

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* ⇒ *leave q*: *queue*

And similarly, we want

$q{\neq}$*emptyq* ⇒ *front q*: X

If we want to prove something about all queues, we need *queue* induction:

emptyq, *join B X*: B ⇒ *queue*: B

And finally, we need to give queues their "first in, first out" character:

leave (*join emptyq x*) = *emptyq*
$q{\neq}$*emptyq* ⇒ *leave* (*join q x*) = *join* (*leave q*) *x*
front (*join emptyq x*) = *x*
$q{\neq}$*emptyq* ⇒ *front* (*join q x*) = *front q*

If we decide to keep the *queue* induction axiom, we can throw away the two earlier axioms

$q{\neq}$*emptyq* ⇒ *leave q*: *queue*
$q{\neq}$*emptyq* ⇒ *front q*: X

since they can now be proved.

End of Data-Queue Theory

After data-stack implementation, data-queue implementation raises no new issues, so we leave it as Exercise 426.

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: *tree*
graft: *tree*→*X*→*tree*→*tree*
emptree, *graft B X B*: *B* ⇒ *tree*: *B*
graft t x u ≠ *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 ≠ *null*
graft t x u: *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

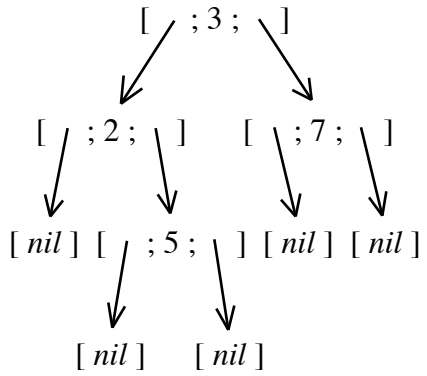
Suppose lists and recursive data definition are implemented. Then we can implement a tree of integers by the following definitions:

tree = *emptree*, *graft tree int tree*
emptree = [*nil*]
graft = ⟨*t*: *tree*· ⟨*x*: *int*· ⟨*u*: *tree*· [*t*; *x*; *u*]⟩⟩⟩
left = ⟨*t*: *tree*· *t* 0⟩
right = ⟨*t*: *tree*· *t* 2⟩
root = ⟨*t*: *tree*· *t* 1⟩

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 = ⟨*t*: *tree*· ⟨*x*: *int*· ⟨*u*: *tree*· (“left”→*t* | “root”→*x* | “right”→*u*)⟩⟩⟩
left = ⟨*t*: *tree*· *t* “left”⟩
right = ⟨*t*: *tree*· *t* “right”⟩
root = ⟨*t*: *tree*· *t* “root”⟩

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 theory is the boundary between an implementer and a user of a data structure. 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 (see Section 5.8) and imperative (see Chapter 4).

7.1 Program Theories

Program (state) variables used to implement a data structure are called implementer's variables. Program variables used to pass data into or out of a data structure are called user's variables. Users and implementers of a data structure can freely see and change their own variables, but they cannot freely see or change each other's variables. A user can use the implementer's variables only as allowed by the theory, and an implementer can use the user's variables only as allowed by the theory. Some programming languages have a "module" or "object" construct for the purpose of prohibiting access to variables on the wrong 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, *top* has value 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 sequential 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

$$push = \langle x: X\cdot s:= s;;[x]\rangle$$
$$pop = s:= s\ [0;..\#s-1]$$
$$top = s\ (\#s-1)$$

And, of course, we must show that these definitions satisfy the axioms. We'll do the first axiom, and leave the other as Exercise 429.

	($top'=x$ $\Longleftarrow$ $push\ x$)	use definition of *push* and *top*
=	($s'(\#s'-1)=x$ $\Longleftarrow$ $s:=s;;[x]$)	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 binary variable to say whether the stack is empty). Letting x: X , the axioms are

$top'=x \ \wedge\ \neg isempty' \Longleftarrow push\ x$
$ok \Longleftarrow push\ x.\ pop$
$isempty' \Longleftarrow mkempty$

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, to specify that popping an empty stack causes an error message to be printed, we could add the axiom

$screen!$ "error" $\Longleftarrow$ $mkempty.\ pop$

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

$top'=x \Longleftarrow push\ x$

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

$top'=top \Longleftarrow balance$
$balance \Longleftarrow ok$
$balance \Longleftarrow push\ x.\ balance.\ pop$

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

$count' = 0 \Longleftarrow start$
$count' = count+1 \Longleftarrow push\ x$
$count' = count+1 \Longleftarrow pop$

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 binary variable 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

$$isemptyq' \Leftarrow mkemptyq$$
$$isemptyq \Rightarrow front'{=}x \wedge \neg isemptyq' \Leftarrow join\ x$$
$$\neg isemptyq \Rightarrow front'{=}front \wedge \neg isemptyq' \Leftarrow join\ x$$
$$isemptyq \Rightarrow (join\ x.\ leave = mkemptyq)$$
$$\neg isemptyq \Rightarrow (join\ x.\ leave = leave.\ join\ x)$$

End of Program-Queue Theory

7.1.5 Program-Tree Theory

Imagine a binary 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* (toward the parent of this node), *left* (toward the left child of this node), or *right* (toward 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.

$$(aim{=}up) = (aim'{\neq}up) \Leftarrow go$$
$$node'{=}node \wedge aim'{=}aim \Leftarrow go.\ work.\ go$$
$$work \Leftarrow ok$$
$$work \Leftarrow node{:=}\ x$$
$$work \Leftarrow a{=}aim{\neq}b \wedge (aim{:=}\ b.\ go.\ work.\ go.\ aim{:=}\ a)$$
$$work \Leftarrow work.\ work$$

Here is another way to define program-trees. Let *T* (for tree) and *p* (for pointer) be implementer's variables. The axioms are

$$tree = [tree; X; tree]$$
$$T{:}\ tree$$
$$p{:}\ *(0, 1, 2)$$
$$node = T@(p; 1)$$
$$change = \langle x{:}\ X{\cdot}\ T{:=}\ (p; 1){\rightarrow}x \mid T\rangle$$
$$goUp = p{:=}\ p_{0;..\leftrightarrow p-1}$$
$$goLeft = p{:=}\ p;0$$
$$goRight = p{:=}\ p;2$$

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 *old* by new implementer's variables *new* using a data transformer, which is a binary expression *D* relating *old* and *new* such that

$$\forall new \cdot \exists old \cdot D$$

Here, *old* and *new* 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

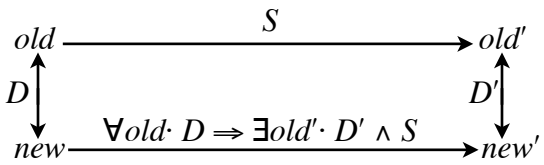
$$\forall old \cdot D \Rightarrow \exists old' \cdot D' \wedge S$$

Specification *S* talks about its nonlocal variables *old* and old' (and the user's variables), and the transformed specification talks about its nonlocal variables *new* and new' (and the user's variables).

Data transformation is invisible to the user. The user imagines that the implementer's variables are initially in state *old*, and then, according to specification *S*, they are finally in state old'. Actually, the implementer's variables will initially be in state *new* related to *old* by *D*; the user will be able to suppose they are in a state *old* because $\forall new \cdot \exists old \cdot D$. The implementer's variables will change state from *new* to new' according to the transformed specification

$$\forall old \cdot D \Rightarrow \exists old' \cdot D' \wedge S$$

This says that whatever related initial state *old* the user was imagining, there is a related final state old' for the user to imagine as the result of *S*, and so the fiction is maintained. Here is a picture.



Implementability of *S* in its variables (old, old') becomes, via the transformer (D, D'), the new specification $\forall old \cdot D \Rightarrow \exists old' \cdot D' \wedge S$ in the new variables (new, new').

Our first example is Exercise 454(a). The user's variable is u: *bin* 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: *bin* according to the data transformer $w =$ *even* v. The first operation *zero* becomes

$\forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \wedge\ (v{:=}\ 0)$

The assignment refers to a state consisting of u and v.

$=$ $\forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \wedge\ u'{=}u\ \wedge\ v'{=}0$ One-Point law

$=$ $\forall v\cdot\ w = even\ v\ \Rightarrow\ w' = even\ 0\ \wedge\ u'{=}u$ change of variable law, simplify

$=$ $\forall r{:}\ even\ nat\cdot\ w{=}r\ \Rightarrow\ w'{=}\top\ \wedge\ u'{=}u$ One-Point law

$=$ $w'{=}\top\ \wedge\ u'{=}u$ The state now consists of u and w.

$=$ $w{:=}\ \top$

Operation *increase* becomes

$\forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \wedge\ (v{:=}\ v{+}1)$

$=$ $\forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \wedge\ u'{=}u\ \wedge\ v'{=}v{+}1$ One-Point law

$=$ $\forall v\cdot\ w = even\ v\ \Rightarrow\ w' = even\ (v{+}1)\ \wedge\ u'{=}u$ change of variable law, simplify

$=$ $\forall r{:}\ even\ nat\cdot\ w{=}r\ \Rightarrow\ w' = \neg r\ \wedge\ u'{=}u$ One-Point law

$=$ $w' = \neg w\ \wedge\ u'{=}u$

$=$ $w{:=}\ \neg w$

Operation *inquire* becomes

$\forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \wedge\ (u{:=}\ even\ v)$

$=$ $\forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \wedge\ u' = even\ v\ \wedge\ v'{=}v$ One-Point law

$=$ $\forall v\cdot\ w = even\ v\ \Rightarrow\ w' = even\ v\ \wedge\ u' = even\ v$ change of variable law

$=$ $\forall r{:}\ even\ nat\cdot\ w{=}r\ \Rightarrow\ w'{=}r\ \wedge\ u'{=}r$ One-Point law

$=$ $w'{=}w\ \wedge\ u'{=}w$

$=$ $u{:=}\ w$

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 455(a). The user's variable is u: *bin* and the implementer's variable is v: *bin*. 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

$\forall v\cdot\ v = even\ w\ \Rightarrow\ \exists v'\cdot\ v' = even\ w'\ \wedge\ (v{:=}\ \top)$ One-Point law twice

$=$ $even\ w'\ \wedge\ u'{=}u$

$\Leftarrow$ $w{:=}\ 0$

Operation *flip* becomes

$\forall v\cdot\ v = even\ w\ \Rightarrow\ \exists v'\cdot\ v' = even\ w'\ \wedge\ (v{:=}\ \neg v)$ One-Point law twice

$=$ $even\ w' \neq even\ w\ \wedge\ u'{=}u$

$\Leftarrow$ $w{:=}\ w{+}1$

Operation *ask* becomes

$\forall v\cdot\ v = even\ w\ \Rightarrow\ \exists v'\cdot\ v' = even\ w'\ \wedge\ (u{:=}\ v)$ One-Point law twice

$=$ $even\ w' = even\ w = u'$

$\Leftarrow$ $u{:=}\ even\ w$

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 460 is to design a security switch. It has three binary 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 binary 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:

$a{:=}\ \neg a.$ **if** $a{\neq}A \wedge b{\neq}B$ **then** $c{:=}\ \neg c.\ A{:=}\ a.\ B{:=}\ b$ **else** ok **fi**

$b{:=}\ \neg b.$ **if** $a{\neq}A \wedge b{\neq}B$ **then** $c{:=}\ \neg c.\ A{:=}\ a.\ B{:=}\ b$ **else** ok **fi**

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

$\Longleftarrow$ $\top$

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.

$$
\begin{array}{lll}
 & \forall A, B{\cdot} & A{=}B{=}c \\
 & \quad \Rightarrow\ \exists A', B'{\cdot} & A'{=}B'{=}c' \\
 & \qquad \wedge\ \textbf{if}\ a{\neq}A \wedge b{\neq}B\ \textbf{then}\ c{:=}\ \neg c.\ A{:=}\ a.\ B{:=}\ b\ \textbf{else}\ ok\ \textbf{fi} \\
 & & \text{expand assignments and } ok \\
= & \forall A, B{\cdot} & A{=}B{=}c \\
 & \quad \Rightarrow\ \exists A', B'{\cdot} & A'{=}B'{=}c' \\
 & \qquad \wedge\ \textbf{if}\ a{\neq}A \wedge b{\neq}B\ \textbf{then}\ a'{=}a \wedge b'{=}b \wedge c'{=}\neg c \wedge A'{=}a \wedge B'{=}b \\
 & & \quad \textbf{else}\ a'{=}a \wedge b'{=}b \wedge c'{=}c \wedge A'{=}A \wedge B'{=}B\ \textbf{fi} \\
 & & \text{one-point for } A' \text{ and } B' \\
= & \forall A, B{\cdot}\ A{=}B{=}c \Rightarrow & \textbf{if}\ a{\neq}A \wedge b{\neq}B\ \textbf{then}\ a'{=}a \wedge b'{=}b \wedge c'{=}\neg c \wedge c'{=}a \wedge c'{=}b \\
 & & \quad \textbf{else}\ a'{=}a \wedge b'{=}b \wedge c'{=}c \wedge c'{=}A \wedge c'{=}B\ \textbf{fi} \\
 & & \text{one-point for } A \text{ and } B \\
= & \textbf{if}\ a{\neq}c \wedge b{\neq}c\ \textbf{then}\ a'{=}a \wedge b'{=}b \wedge c'{=}\neg c \wedge c'{=}a \wedge c'{=}b \\
 & \textbf{else}\ a'{=}a \wedge b'{=}b \wedge c'{=}c \wedge c'{=}c \wedge c'{=}c\ \textbf{fi} \\
 & & \text{use } \textbf{if}\text{-part as context to change } \textbf{then}\text{-part} \\
= & \textbf{if}\ a{\neq}c \wedge b{\neq}c\ \textbf{then}\ a'{=}a \wedge b'{=}b \wedge c'{=}\neg c \wedge c'{=}\neg c \wedge c'{=}\neg c \\
 & \textbf{else}\ a'{=}a \wedge b'{=}b \wedge c'{=}c \wedge c'{=}c \wedge c'{=}c\ \textbf{fi} \\
= & \textbf{if}\ a{\neq}c \wedge b{\neq}c\ \textbf{then}\ c{:=}\ \neg c\ \textbf{else}\ ok\ \textbf{fi} \\
= & c{:=}\ (a{\neq}c \wedge b{\neq}c) \neq c
\end{array}
$$

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 462 (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

$$start \;=\; s'=\{null\} \;\wedge\; n'=n$$
$$take \;=\; \neg\, n'\in s \;\wedge\; s' = s\cup\{n'\}$$
$$give \;=\; n\in s \;\Rightarrow\; \neg\, n\in s' \;\wedge\; s'\cup\{n\} = s \;\wedge\; n'=n$$

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.

$$\forall s\cdot\, s\subseteq\{0,..m\} \;\Rightarrow\; \exists s'\cdot\, s'\subseteq\{0,..m'\} \wedge s'=\{null\} \wedge n'=n \qquad \text{one-point and identity}$$
$$=\quad n'=n$$
$$\Leftarrow\quad ok$$

The transformed specification is just $n'=n$, 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.

$$\forall s\cdot\, s\subseteq\{0,..m\} \;\Rightarrow\; \exists s'\cdot\, s'\subseteq\{0,..m'\} \;\wedge\; \neg\, n'\in s \;\wedge\; s' = s\cup\{n'\}$$

several omitted steps

$$=\quad m \le n' < m'$$
$$\Leftarrow\quad n:= m.\;\; m:= m+1$$

Operation *give* is transformed as follows.

$$\forall s\cdot\, s\subseteq\{0,..m\} \;\Rightarrow\; \exists s'\cdot\, s'\subseteq\{0,..m'\} \;\wedge\; (n\in s \;\Rightarrow\; \neg\, n\in s' \;\wedge\; s'\cup\{n\} = s \;\wedge\; n'=n)$$

several omitted steps

$$=\quad (n+1 = m \Rightarrow n \le m') \wedge (n+1 < m \Rightarrow m \le m') \wedge n'=n$$
$$\Leftarrow\quad ok$$

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 $s \subseteq \{0,..i\uparrow j\}$. Operation *take* becomes

$$\forall s\cdot\, s\subseteq\{0,..i\uparrow j\} \;\Rightarrow\; \exists s'\cdot\, s'\subseteq\{0,..i'\uparrow j'\} \;\wedge\; \neg\, n'\in s \;\wedge\; s' = s\cup\{n'\}$$

several omitted steps

$$=\quad i\uparrow j \le n' < i'\uparrow j'$$
$$\Leftarrow\quad n:= i\uparrow j.\;\; \textbf{if } i\ge j \textbf{ then } i:= i+1 \textbf{ else } j:= j+1 \textbf{ fi}$$

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

$\forall k$: $\sim s\cdot$ $even\ k \wedge k<i \ \vee \ odd\ k \wedge k<j$

Now *take* becomes

$\forall s\cdot \quad (\forall k: \sim s\cdot \ even\ k \wedge k<i \ \vee \ odd\ k \wedge k<j)$
$\Rightarrow \ \exists s'\cdot \ (\forall k: \sim s'\cdot \ even\ k \wedge k<i' \ \vee \ odd\ k \wedge k<j') \ \wedge \ \neg\ n'\in s \ \wedge \ s' = s\cup\{n'\}$
several omitted steps
$=\quad even\ n' \ \wedge \ i \le n' < i' \ \vee \ odd\ n' \ \wedge \ j \le n' < j'$
$\Leftarrow\quad (n:= i.\ \ i:= i+2) \ \vee \ (n:= j.\ \ j:= j+2)$

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 451 (parsing): Define E as a bunch of strings of lists of characters satisfying

E = ["x"], ["if"]; E; ["then"]; E; ["else"]; E; ["fi"]

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 binary 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:

$q' = (s_{0;..\leftrightarrow s-1} : E) \ \Leftarrow \ A:= E;["eog"].\ \ n:= 0.\ \ P$

where $P \ = \ n\le\leftrightarrow s \ \wedge \ A_{0;..n} = s_{0;..n} \ \Rightarrow \ q' = (s_{0;..\leftrightarrow s-1};["eog"] : A)$. 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:= (§a: $A\cdot\ a_n = s_n$). n:= n+1. P
else q:= ["eog"]: $A_n \ \wedge \ s_n$=["eos"] **fi**

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"]; ["fi"]

represents the bunch of strings

["if"]; ["x"]; ["then"]; E; ["else"]; E; ["fi"]

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 $b{:=}b_{0;..n};["if"];["E"];["then"];["E"];["else"];["E"];["fi"];b_{n+1;..\leftrightarrow b}.$
$n{:=}\,n{+}1.\ Q$
else $q{:=}\ b_n{=}["eog"] \wedge s_n{=}["eos"]$ **fi fi fi**

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 $b{:=}b_{0;..n};["if"];["x"];["then"];["x"];["else"];["x"];["fi"];b_{n+1;..\leftrightarrow b}.$
$n{:=}\,n{+}1.\ Q$
else $q{:=}\ b_n{=}["eog"] \wedge s_n{=}["eos"]$ **fi fi**

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 $c{:=}\,["x"];["then"];["x"];["else"];["x"];["fi"]c.\ n{:=}\,n{+}1.\ R$
else $q{:=}\ c_0{=}["eog"] \wedge s_n{=}["eos"]$ **fi fi**

End of Parsing

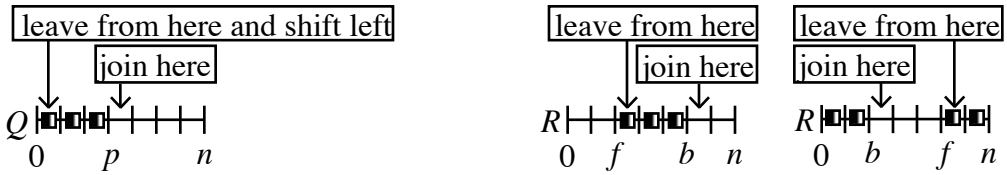
7.2.3 Limited Queue

The next example, Exercise 464, 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 n: $nat{+}1$, and let Q: $[n{*}X]$ and p: $0{,..}n{+}1$ be implementer's variables. Then the original implementation is as follows.

$mkemptyq \;=\; p{:=}\,0$
$isemptyq \;=\; p{=}0$
$isfullq \;=\; p{=}n$
$join\ x \;=\; Q\ p{:=}\,x.\ \ p{:=}\,p{+}1$
$leave \;=\; \mathbf{for}\ i{:=}\,1{;..}p\ \mathbf{do}\ Q\ (i{-}1){:=}\,Q\ i\ \mathbf{od}.\ \ p{:=}\,p{-}1$
$front \;=\; Q\ 0$

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: $0{,..}n{+}1$ 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 .

$Q[0;..p] = R[f;..b] \ \vee\ Q[0;..p] = R[(f;..n);\,(0;..b)]$

The conjuncts $0{\le}p{\le}n \wedge 0{\le}f{\le}b{\le}n \wedge p{=}b{-}f$ are implicit in the left disjunct, and the conjuncts $0{\le}p{\le}n \wedge 0{\le}f{\le}n \wedge 0{\le}b{\le}n \wedge p{=}n{-}f{+}b$ are implicit in the right disjunct.

Now we transform. First $mkemptyq$.

$\forall Q, p{\cdot}\ D \Rightarrow \exists Q', p'{\cdot}\ D' \wedge p'{=}0 \wedge Q'{=}Q$ several omitted steps
$=\quad f'{=}b' \ \vee\ b'{=}0 \wedge f'{=}n$
$\Longleftarrow\quad f{:=}\,0.\ \ b{:=}\,0$

Next we transform $isemptyq$. Although $isemptyq$ happens to be binary and can be interpreted as an unimplementable specification, its purpose (like $front$, which isn't binary) 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 binary user's variable, and transform $c{:=}\,isemptyq$.

$\forall Q, p{\cdot}\ D \Rightarrow \exists Q', p'{\cdot}\ D' \wedge c'{=}(p{=}0) \wedge p'{=}p \wedge Q'{=}Q$ several omitted steps
$=\quad f{<}b \ \wedge\ f'{<}b' \ \wedge\ R[f;..b] = R'[f';..b'] \ \wedge\ \neg c'$
$\vee\quad f{<}b \ \wedge\ f'{>}b' \ \wedge\ R[f;..b] = R'[(f';..n);\,(0;..b')] \ \wedge\ \neg c'$
$\vee\quad f{>}b \ \wedge\ f'{<}b' \ \wedge\ R[(f;..n);\,(0;..b)] = R'[f';..b'] \ \wedge\ \neg c'$
$\vee\quad f{>}b \ \wedge\ f'{>}b' \ \wedge\ R[(f;..n);\,(0;..b)]{=}R'[(f';..n);\,(0;..b')] \ \wedge\ \neg c'$

Initially R might be in the "inside" or "outside" configuration, and finally R' might be either way, so that gives us four disjuncts. 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: *bin* to say whether we have the "inside" mode or "outside" mode. We revise the transformer D as follows:

$$m \;\wedge\; Q[0;..p] = R[f;..b] \quad \vee \quad \neg m \;\wedge\; Q[0;..p] = R[(f;..n);\,(0;..b)]$$

Now we have to retransform *mkemptyq* .

$$\begin{array}{lll} & \forall Q, p{\cdot}\; D \Rightarrow \exists Q', p'{\cdot}\; D' \wedge p'{=}0 \wedge Q'{=}Q & \text{several omitted steps} \\ = & m' \wedge f'{=}b' \;\vee\; \neg m' \wedge f'{=}n \wedge b'{=}0 & \\ \Leftarrow & m{:=}\,\top.\;\; f{:=}\,0.\;\; b{:=}\,0 & \end{array}$$

Next we retransform $c{:=}$ *isemptyq* .

$$\begin{array}{lll} & \forall Q, p{\cdot}\; D \Rightarrow \exists Q', p'{\cdot}\; D' \wedge c'{=}(p{=}0) \wedge p'{=}p \wedge Q'{=}Q & \text{several omitted steps} \\ = & \quad m \;\wedge\; m' \;\wedge\; R[f;..b] = R'[f';..b'] \;\wedge\; c'{=}(f{=}b) & \\ & \vee\; m \;\wedge\; \neg m' \;\wedge\; R[f;..b] = R'[(f';..n');\,(0;..b')] \;\wedge\; c'{=}(f{=}b)\big) & \\ & \vee\; \neg m \;\wedge\; m' \;\wedge\; R[(f;..n);\,(0;..b)] = R'[f';..b'] \;\wedge\; c'{=}(b{=}0 \wedge f{=}n) & \\ & \vee\; \neg m \;\wedge\; \neg m' \;\wedge\; R[(f;..n);\,(0;..b)] = R'[(f';..n');\,(0;..b')] \;\wedge\; c'{=}(b{=}0 \wedge f{=}n) & \\ \Leftarrow & c' = (m \wedge f{=}b \;\vee\; \neg m \wedge b{=}0 \wedge f{=}n) \wedge f'{=}f \wedge b'{=}b \wedge R'{=}R \wedge m'{=}m & \\ = & c{:=}\;\; \textbf{if}\; m \;\textbf{then}\; f{=}b \;\textbf{else}\; b{=}0 \wedge f{=}n \;\textbf{fi} & \end{array}$$

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* .

$$\begin{array}{lll} & \forall Q, p{\cdot}\; D \Rightarrow \exists Q', p'{\cdot}\; D' \wedge c'{=}(p{=}n) \wedge p'{=}p \wedge Q'{=}Q & \text{several omitted steps} \\ \Leftarrow & c{:=}\;\; \textbf{if}\; m \;\textbf{then}\; f{=}0 \wedge b{=}n \;\textbf{else}\; f{=}b \;\textbf{fi} & \end{array}$$

Next we transform *join* x . Before this operation, there should be a check that the queue is not full.

$$\begin{array}{lll} & \forall Q, p{\cdot}\; D \Rightarrow \exists Q', p'{\cdot}\; D' \;\wedge\; Q' = p{\rightarrow}x \mid Q \;\wedge\; p'{=}p{+}1 & \text{several omitted steps} \\ \Leftarrow & \textbf{if}\; b{<}n \;\textbf{then}\; R\; b{:=}\,x.\;\; b{:=}\,b{+}1 \;\textbf{else}\; R\; 0{:=}\,x.\;\; b{:=}\,1.\;\; m{:=}\,\bot \;\textbf{fi} & \end{array}$$

Next we transform *leave* . Before this operation, there should be a check that the queue is not empty.

$$\begin{array}{lll} & \forall Q, p{\cdot}\; D \Rightarrow \exists Q', p'{\cdot}\; D' \wedge Q'{=}Q[(1;..p);\,(p{-}1;..n)] \wedge p'{=}p{-}1 & \text{several omitted steps} \\ \Leftarrow & \textbf{if}\; f{<}n \;\textbf{then}\; f{:=}\,f{+}1 \;\textbf{else}\; f{:=}\,1.\;\; m{:=}\,\top \;\textbf{fi} & \end{array}$$

Last we transform $x{:=}$ *front* where x is a user's variable of the same type as the items. Before this operation, there should be a check that the queue is not empty.

$$\begin{array}{lll} & \forall Q, p{\cdot}\; D \Rightarrow \exists Q', p'{\cdot}\; D' \wedge x' = Q\,0 \wedge p'{=}p \wedge Q'{=}Q & \text{several omitted steps} \\ \Leftarrow & \textbf{if}\; f{<}n \;\textbf{then}\; x{:=}\;\; R\,f \;\textbf{else}\; x{:=}\;\; R\,0 \;\textbf{fi} & \end{array}$$

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 465 illustrates the problem. The user's variable is i and the implementer's variable is j, both of type $0, 1, 2$. The operations are:

initialize $=$ $i'=0$

step $=$ **if** $j>0$ **then** $i := i+1$. $j := j-1$ **else** *ok* **fi**

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:

$\forall j \cdot j=0 \Rightarrow \exists j' \cdot j'=0 \wedge i'=j'=0$

$=$ $i := 0$

And it transforms *step* as follows:

$\forall j \cdot j=0 \Rightarrow \exists j' \cdot j'=0 \ \wedge$ **if** $j>0$ **then** $i := i+1$. $j := j-1$ **else** *ok* **fi**

$=$ *ok*

If this were a practical problem, we would be done. But the theoretical problem is to replace j with binary variable b without resolving the nondeterminism, so that

initialize is transformed to $i'=0$

step is transformed to **if** $b \wedge i<2$ **then** $i' = i+1$ **else** *ok* **fi**

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 the increased value of i will not exceed 2. 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. That's because the initial value of j gives us 3 different behaviors, but the initial value of binary variable b cannot distinguish among these 3 behaviors.

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 misused 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 Concurrent Composition

We define the concurrent 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, concurrently, according to Q. The operands of $\|$ are called processes.

When we defined the sequential 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 concurrent 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, concurrent composition is conjunction.

$$P \| Q \;=\; P \wedge Q$$

When we decide to create a concurrent 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 concurrent composition

$$x := x+1 \;\|\; y := y+2$$

if we partition the variables. For the assignments to make sense, x has to belong to the left process and 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 concurrent composition is responsible for deciding how to partition the variables. If we are presented with a concurrent 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 binary variable b and integer variable x,

$b := x=x \parallel x := x+1$ replace $x=x$ by $\top$

$= \quad b := \top \parallel x := x+1$

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 sequential composition as defined in Subsection 4.0.0, the intermediate values of variables are local to the sequential composition; they are hidden by the quantifier $\exists x'', y'', \ldots$. If one process is a sequential composition, the other cannot see its intermediate values. For example,

$(x := x+1.\ x := x-1) \parallel y := x$

$= \quad ok \parallel y := x$

$= \quad y := x$

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 so processes 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

$ok \parallel x := 3$

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

$(x := x+1.\ x := x-1) \parallel x := 3$

Sometimes the need for shared memory arises from poor program structure. For example, suppose we decide to have two processes, as follows.

$(x := x+y.\ x := x \times y) \parallel (y := x-y.\ y := x/y)$

The left process modifies x twice, and the right process 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

$(x := x+y \parallel y := x-y).\ (x := x \times y \parallel y := x/y)$

So far, concurrent composition is just conjunction, and there is no need to introduce a second symbol $\parallel$ for conjunction. But now we consider time. The time variable is not subject to partitioning; it belongs to both processes. In $P \parallel Q$, both P and Q begin execution at time t, but their executions may finish at different times. Execution of the composition $P \parallel Q$ finishes when both P and Q are finished. With time, concurrent composition is defined as

$P \parallel Q \quad = \quad \exists t_P, t_Q \cdot \ \langle t' \cdot P \rangle\, t_P \ \wedge \ \langle t' \cdot Q \rangle\, t_Q \ \wedge \ t' = t_P \uparrow t_Q$

$= \quad \exists t_P, t_Q \cdot$ (substitute t_P for t' in P) $\wedge$ (substitute t_Q for t' in Q) $\wedge \ t' = t_P \uparrow t_Q$

8.0.0 Laws of Concurrent Composition

Let x and y be different state variables, let t be the time, let e, f, and b be expressions of the prestate, and let P, Q, R, and S be specifications. Then

$(x{:=}\,e \parallel y{:=}f).\ P \ =\ (\text{for } x \text{ substitute } e \text{ and concurrently for } y \text{ substitute } f \text{ in } P)$ concurrent substitution

$P \parallel Q \ =\ Q \parallel P$ symmetry

$P \parallel (Q \parallel R) \ =\ (P \parallel Q) \parallel R$ associativity

$P \parallel Q \vee R \ =\ (P \parallel Q) \vee (P \parallel R)$ distributivity

$P \parallel \textbf{if } b \textbf{ then } Q \textbf{ else } R \textbf{ fi} \ =\ \textbf{if } b \textbf{ then } P \parallel Q \textbf{ else } P \parallel R \textbf{ fi}$ distributivity

$\textbf{if } b \textbf{ then } P \parallel Q \textbf{ else } R \parallel S \textbf{ fi} \ =\ \textbf{if } b \textbf{ then } P \textbf{ else } R \textbf{ fi} \parallel \textbf{if } b \textbf{ then } Q \textbf{ else } S \textbf{ fi}$ 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 concurrent composition:

If $A \Leftarrow B \parallel C$ and $B \Leftarrow D$ and $C \Leftarrow E$ are theorems, then $A \Leftarrow D \parallel E$ is a theorem.

So does Refinement by Parts:

If $A \Leftarrow B \parallel C$ and $D \Leftarrow E \parallel F$ are theorems, then $A \wedge D \ \Leftarrow\ B \wedge E \parallel C \wedge F$ is a theorem.

End of Laws of Concurrent Composition

8.0.1 List Concurrency

We have defined concurrent composition by partitioning the variables. For finer-grained concurrency, we can extend this same idea to the individual items within list variables. In Subsection 5.1.0 we defined assignment to a list item as

$L\,i{:=}\,e \ =\ L'i{=}e \ \wedge\ (\forall j{\cdot}\ j{\neq}i \Rightarrow L'j = L\,j) \ \wedge\ x'{=}x \ \wedge\ y'{=}y \ \wedge \ldots$

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 concurrent 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 172: find the maximum item in a list. The maximum of a list is easily expressed with the $\Uparrow$ quantifier, but we will assume $\Uparrow$ is not implemented. The easiest and simplest solution is probably functional, with concurrency coming from the fact that the arguments of a function (operands of an operator) can always be evaluated concurrently. To use concurrent composition, 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

$L'\,0 = \Uparrow L \ \wedge\ t' = t + ceil(log(\#L))$

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

$findmax \ =\ \langle i, j{\cdot}\ i{<}j \ \Rightarrow\ L'i = \Uparrow L\,[i;..j] \ \wedge\ t' = t + ceil(log(j{-}i))\rangle$

Our specification is $findmax\ 0\ (\#L)$. We refine as follows.

findmax i j ⟸ **if** *j–i* = 1 **then** *ok*
else *t*:= *t*+1. (*findmax i* (*div* (*i*+*j*) 2) || *findmax* (*div* (*i*+*j*) 2) *j*).
L i:= *L i* ↑ *L* (*div* (*i*+*j*) 2) **fi**

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 concurrent 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 = ⟨*i*, *j*· *i*<*j* ⇒ *M* ′*i* = ⇑ *L* [*i*;..*j*] ∧ *t*′ = *t* + *ceil*(*log*(*j–i*))⟩

and in the program we change *ok* to *M i* := *L i* and we change the final assignment to

M i:= *M i* ↑ *M* (*div* (*i*+*j*) 2)

End of List Concurrency

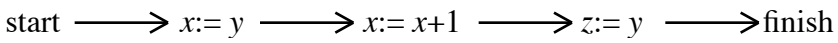
End of Concurrent Composition

8.1 Sequential to Concurrent Transformation

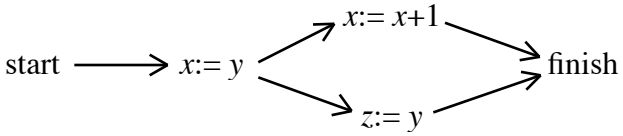
The goal of this section is to transform programs without concurrency into programs with concurrency. A simple example illustrates the idea.

x:= *y*. *x*:= *x*+1. *z*:= *y*
= *x*:= *y*. (*x*:= *x*+1 || *z*:= *y*)
= (*x*:= *y*. *x*:= *x*+1) || *z*:= *y*

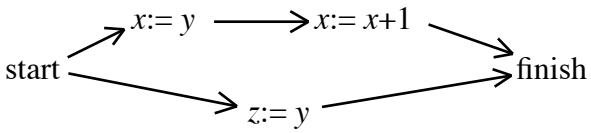
Execution of the program on the first line can be depicted as follows.



The first two assignments cannot be executed concurrently, but the last two can, so we transform the program. Execution can now be depicted as



Now we have the first and last assignments next to each other, in sequence; they too can be executed concurrently. Execution can be



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 a compiler. Sometimes it can only be performed by a compiler 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. Using dots for uninteresting program parts,

produce = ·····b:= e·····
consume = ·····x:= b·····

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. But 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*

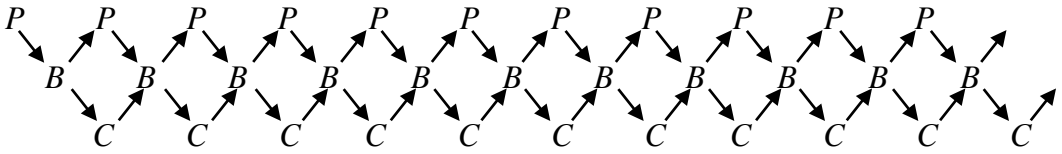
and *newcontrol* can be transformed to

newcontrol = (*consume* || *produce*). *newcontrol*

In this transformed program, a compiler will have to capture a copy of the initial value of b for *consume* to use. Or, we could do this capture at source level, using variable c, as follows.

produce = ·····b:= e·····
consume = ·····x:= c·····
control = *produce*. *newcontrol*
newcontrol = c:= b. (*consume* || *produce*). *newcontrol*

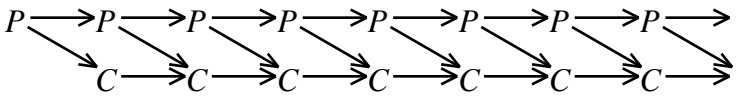
Using B for the assignment c:= b, execution is



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 = ·····b w:= e. w:= w+1·····
consume = ·····x:= b r. r:= r+1·····
control = *produce*. *consume*. *control*

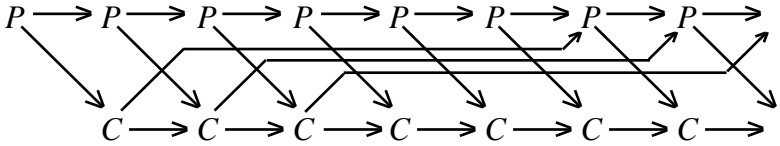
Whenever $w \neq r$, *produce* and *consume* can be executed concurrently. Here is the execution pattern:



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 concurrent execution can be found automatically by the programming language compiler, or it can be told to the compiler 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 = ·····*b* $w{:=}$ *e*. $w{:=}$ *mod* $(w{+}1)$ *n*·····
consume = ·····*x*:= *b r*. $r{:=}$ *mod* $(r{+}1)$ *n*·····
control = *produce*. *consume*. *control*



This picture shows the execution pattern for $n{=}6$. 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 compiler. Synchronization is what remains of sequential execution after all opportunities for concurrency have been found.

8.1.1 Insertion Sort

Exercise 209 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

$$sort = \langle n{\cdot}\ \forall i, j{:}\ 0{,}..n{\cdot}\ i{\le}j \Rightarrow L\,i \le L\,j \rangle$$

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 using

$$swap\ i\ j = L\,i{:=}\,L\,j \parallel L\,j{:=}\,L\,i$$

to make changes to L. We ignore the last conjunct; program transformation will give a linear time solution. The second conjunct is equal to *sort* $0 \Rightarrow sort'\ (\#L)$ since *sort* 0 is a theorem.

$$sort\ 0 \Rightarrow sort'\ (\#L) \Leftarrow \textbf{for}\ n{:=}\,0{;}..\#L\ \textbf{do}\ sort\ n \Rightarrow sort'\ (n{+}1)$$

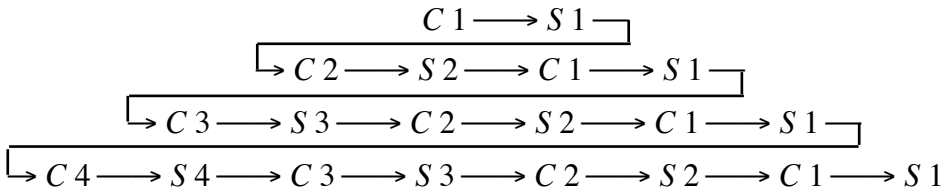
To solve *sort n* $\Rightarrow$ *sort'* $(n{+}1)$, it may help to refer to an example list.

$$\underset{0}{[}\ L\,0\ \underset{1}{;}\ L\,1\ \underset{2}{;}\ L\,2\ \underset{3}{;}\ L\,3\ \underset{4}{;}\ L\,4\ \underset{5}{]}$$

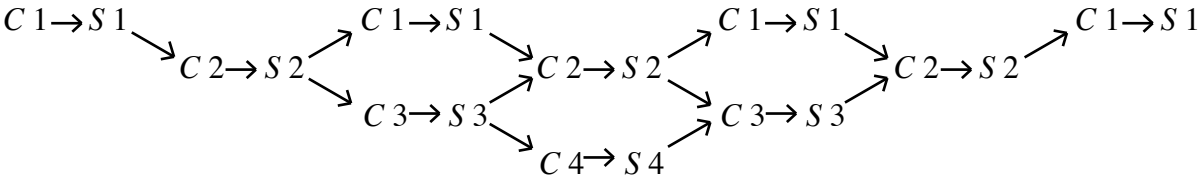
sort n $\Rightarrow$ *sort'* $(n{+}1)$ $\Leftarrow$ **if** $n{=}0$ **then** *ok*
else if $L\ (n{-}1) \le L\ n$ **then** *ok*
else *swap* $(n{-}1)\ n$. *sort* $(n{-}1) \Rightarrow sort'\ n$ **fi fi**

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 concurrent 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 compiler, we obtain a linear-time parallel sort.

End of Insertion Sort

8.1.2 Dining Philosophers

Exercise 491: 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).

$$\begin{array}{lcl} \mathit{life} & = & (P\,0 \vee P\,1 \vee P\,2 \vee P\,3 \vee P\,4).\ \mathit{life} \\ P\,i & = & \mathit{up}\,i.\ \mathit{up}(i{+}1).\ \mathit{eat}\,i.\ \mathit{down}\,i.\ \mathit{down}(i{+}1) \\ \mathit{up}\,i & = & \mathit{chopstick}\,i := \top \\ \mathit{down}\,i & = & \mathit{chopstick}\,i := \bot \\ \mathit{eat}\,i & = & \cdots\cdots \mathit{chopstick}\,i \cdots\cdots \mathit{chopstick}(i{+}1) \cdots\cdots \end{array}$$

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 (this is where the criterion that each philosopher eats infinitely often would be met). It is unclear how to define $\mathit{eat}\,i$, except that it uses two chopsticks. (If this program were intended to accomplish some purpose, we could eliminate variable $\mathit{chopstick}$, replacing both occurrences in $\mathit{eat}\,i$ 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 $i{\neq}j$, $(\mathit{up}\,i.\,\mathit{up}\,j)$ becomes $(\mathit{up}\,i \parallel \mathit{up}\,j)$.
If $i{\neq}j$, $(\mathit{up}\,i.\,\mathit{down}\,j)$ becomes $(\mathit{up}\,i \parallel \mathit{down}\,j)$.
If $i{\neq}j$, $(\mathit{down}\,i.\,\mathit{up}\,j)$ becomes $(\mathit{down}\,i \parallel \mathit{up}\,j)$.
If $i{\neq}j$, $(\mathit{down}\,i.\,\mathit{down}\,j)$ becomes $(\mathit{down}\,i \parallel \mathit{down}\,j)$.
If $i{\neq}j \wedge i{+}1{\neq}j$, $(\mathit{eat}\,i.\,\mathit{up}\,j)$ becomes $(\mathit{eat}\,i \parallel \mathit{up}\,j)$.
If $i{\neq}j \wedge i{\neq}j{+}1$, $(\mathit{up}\,i.\,\mathit{eat}\,j)$ becomes $(\mathit{up}\,i \parallel \mathit{eat}\,j)$.
If $i{\neq}j \wedge i{+}1{\neq}j$, $(\mathit{eat}\,i.\,\mathit{down}\,j)$ becomes $(\mathit{eat}\,i \parallel \mathit{down}\,j)$.
If $i{\neq}j \wedge i{\neq}j{+}1$, $(\mathit{down}\,i.\,\mathit{eat}\,j)$ becomes $(\mathit{down}\,i \parallel \mathit{eat}\,j)$.
If $i{\neq}j \wedge i{+}1{\neq}j \wedge i{\neq}j{+}1$, $(\mathit{eat}\,i.\,\mathit{eat}\,j)$ becomes $(\mathit{eat}\,i \parallel \mathit{eat}\,j)$.

Different chopsticks can be picked up or put down at the same time. Eating can be concurrent 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 concurrent 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 compiler 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.

$$\begin{array}{lcl} \mathit{life} & = & P\,0 \parallel P\,1 \parallel P\,2 \parallel P\,3 \parallel P\,4 \\ P\,i & = & (\mathit{up}\,i \parallel \mathit{up}(i{+}1)).\ \mathit{eat}\,i.\ (\mathit{down}\,i \parallel \mathit{down}(i{+}1)).\ P\,i \end{array}$$

$P\,0$ cannot be placed in parallel with $P\,1$ because they both assign and use $\mathit{chopstick}\,1$. 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 Concurrent 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

new x: $T\cdot S \;=\; \exists x, x'\colon T\cdot S$

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 sequential composition, but these intermediate values are local to the definition of sequential composition.

$P.Q \;=\; \exists x'', y'', ...\cdot \langle x', y', ...\cdot P\rangle\, x''\, y''\, ...\; \wedge\; \langle x, y, ...\cdot Q\rangle\, x''\, y''\, ...$

Consider $(P.Q) \parallel R$. The intermediate values between P and Q are hidden in the sequential 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 concurrent 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 **new** x: $time\to T\cdot S$ declare x to be an interactive variable of type T and scope S. It is defined as follows.

new x: $time\to T\cdot S \;=\; \exists x\colon time\to T\cdot S$

where $time$ is the domain of time, either the extended naturals or the nonnegative extended reals. An interactive variable is a function of time. The value of variable x at time t is $x\,t$. We sometimes omit the argument t when the context makes it clear, and write x for $x\,t$. We will similarly write x' to mean $x\,t'$, and x'' to mean $x\,t''$.

Suppose a and b are boundary variables, x and y are interactive variables, and t is time. The definition of ok says that the boundary variables and time are unchanged.

$ok \;=\; a'{=}a \;\wedge\; b'{=}b \;\wedge\; t'{=}t$

We could also say $x'{=}x \;\wedge\; y'{=}y$, which means $x\,t' = x\,t \;\wedge\; y\,t' = y\,t$, but there is no need to say that because $t'{=}t$.

Assignment to a boundary variable is similar. In the same variables,

$a{:=}\,e \;=\; a'{=}e \;\wedge\; b'{=}b \;\wedge\; t'{=}t$

Assignment to an interactive variable cannot be instantaneous because it is time that distinguishes its values. In the variables of the previous paragraph,

$x{:=}\,e \;=\; a'{=}a \;\wedge\; b'{=}b \;\wedge\; x'{=}e \;\wedge\; (\forall t''\cdot t{\le}t''{\le}t' \Rightarrow y''{=}y)$
$\qquad\qquad \wedge\; t' = t{+}(\text{the time required to evaluate and store } e)$

At the final time t', interactive variable x has value e, but nothing is said about the value of x during the assignment. Interactive variable y remains unchanged throughout the duration of the assignment to x.

Assignment to a boundary variable is instantaneous in the recursive time measure. In the real time measure, assignment to a boundary variable takes some time, so we must say that all interactive variables remain unchanged during the assignment.

Sequential 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

$$P.Q \;=\; \exists a'', b'', t''\cdot\ \langle a', b', t'\cdot P\rangle\ a''\ b''\ t''\ \wedge\ \langle a, b, t\cdot Q\rangle\ a''\ b''\ t''$$

For concurrent composition we partition all the variables, both boundary and interactive (but not time). Suppose a and x belong to P, and b and y belong to Q.

$$\begin{aligned} P\|Q \;=\; \exists t_P, t_Q\cdot\quad & \langle t'\cdot P\rangle\ t_P\ \wedge\ (\forall t''\cdot\ t_P\le t''\le t' \Rightarrow x\ t'' = x\ (t_P)) \\ \wedge\ & \langle t'\cdot Q\rangle\ t_Q\ \wedge\ (\forall t''\cdot\ t_Q\le t''\le t' \Rightarrow y\ t'' = y\ (t_Q)) \\ \wedge\ & t' = t_P \uparrow t_Q \end{aligned}$$

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, variables, and partitions 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 P, or their values at the start of P if 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 value at the start of Q 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 both P and Q.

Most of the specification laws and refinement laws survive the addition of interactive variables, but sadly, the Substitution Law no longer works.

Exercise 495 is an example in the same variables a, b, x, y, and t. Suppose that time is an extended natural, and that each assignment takes time 1.

$$\begin{aligned} & (x{:=}\ 2.\ \ x{:=}\ x{+}y.\ \ x{:=}\ x{+}y)\ \|\ (y{:=}\ 3.\ \ y{:=}\ x{+}y) && \text{x is a variable in the left process and y is a variable in the right process.} \\ & && \text{Let's put a in the left process and b in the right process.} \\ =\quad & (a'{=}a\ \wedge\ x'{=}2\ \wedge\ t'{=}t{+}1.\ a'{=}a\ \wedge\ x'{=}\ x{+}y\ \wedge\ t'{=}t{+}1.\ a'{=}a\ \wedge\ x'{=}\ x{+}y\ \wedge\ t'{=}t{+}1) \\ & \|\ (b'{=}b\ \wedge\ y'{=}3\ \wedge\ t'{=}t{+}1.\ \ b'{=}b\ \wedge\ y'{=}\ x{+}y\ \wedge\ t'{=}t{+}1) \\ =\quad & (a'{=}a\ \wedge\ x(t{+}1){=}2\ \wedge\ x(t{+}2){=}\ x(t{+}1){+}y(t{+}1)\ \wedge\ x(t{+}3){=}\ x(t{+}2){+}y(t{+}2)\ \wedge\ t'{=}t{+}3) \\ & \|\ (b'{=}b\ \wedge\ y(t{+}1){=}3\ \wedge\ y(t{+}2){=}\ x(t{+}1){+}y(t{+}1)\ \wedge\ t'{=}t{+}2) \\ =\quad & a'{=}a\ \wedge\ x(t{+}1){=}2\ \wedge\ x(t{+}2){=}\ x(t{+}1){+}y(t{+}1)\ \wedge\ x(t{+}3){=}\ x(t{+}2){+}y(t{+}2) \\ & \wedge\ b'{=}b\ \wedge\ y(t{+}1){=}3\ \wedge\ y(t{+}2){=}\ x(t{+}1){+}y(t{+}1)\ \wedge\ y(t{+}3){=}y(t{+}2)\ \wedge\ t'{=}t{+}3 \\ =\quad & a'{=}a\ \wedge\ x(t{+}1){=}2\ \wedge\ x(t{+}2){=}5\ \wedge\ x(t{+}3){=}10 \\ & \wedge\ b'{=}b\ \wedge\ y(t{+}1){=}3\ \wedge\ y(t{+}2){=}y(t{+}3){=}5\ \wedge\ \ t'{=}t{+}3 \end{aligned}$$

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 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.

Let a be any number of boundary variables, let x be any number of interactive variables, and let t be time. Specification S is implementable if

$$\forall a, X, t \cdot \exists a', x, t' \cdot \; S \;\wedge\; t \le t' \;\wedge\; \forall t'' \cdot\; t < t'' \le t' \;\vee\; x\, t'' = X\, t''$$

The inputs are a and $x\, t$, and the outputs are a' and $x\, t''$ for $t < t'' \le t'$. As before, S must be satisfiable with nondecreasing time; the new part says that S must not constrain its interactive variables x outside the interval from t to t'.

9.0.0 Thermostat

Exercise 500: specify a thermostat for a gas burner. The thermostat operates concurrently with other processes

$$\mathit{thermometer} \parallel \mathit{control} \parallel \mathit{thermostat} \parallel \mathit{burner}$$

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.
- binary *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:

- binary *gas*; assigning it $\top$ turns the gas on and $\bot$ turns the gas off.
- binary *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:

$$\mathit{thermostat} \;=\; (\mathit{gas} := \bot \parallel \mathit{spark} := \bot).\; \mathit{GasIsOff}$$

$$\begin{aligned}\mathit{GasIsOff} \;=\;& \textbf{if } \mathit{temperature} < \mathit{desired} - \varepsilon \\ & \textbf{then } (\mathit{gas} := \top \parallel \mathit{spark} := \top \parallel t' \ge t{+}1) \;\wedge\; t' \le t{+}3.\; \mathit{spark} := \bot.\; \mathit{GasIsOn} \\ & \textbf{else } ((\textbf{frame } \mathit{gas}, \mathit{spark} \cdot \mathit{ok}) \parallel t' \ge t) \;\wedge\; t' \le t{+}1.\; \mathit{GasIsOff}\ \textbf{fi}\end{aligned}$$

$$\begin{aligned}\mathit{GasIsOn} \;=\;& \textbf{if } \mathit{temperature} < \mathit{desired} + \varepsilon \;\wedge\; \mathit{flame} \\ & \textbf{then } ((\textbf{frame } \mathit{gas}, \mathit{spark} \cdot \mathit{ok}) \parallel t' \ge t) \;\wedge\; t' \le t{+}1.\; \mathit{GasIsOn} \\ & \textbf{else } (\mathit{gas} := \bot \parallel (\textbf{frame } \mathit{spark} \cdot \mathit{ok}) \parallel t' \ge t{+}20) \;\wedge\; t' \le t{+}21.\; \mathit{GasIsOff}\ \textbf{fi}\end{aligned}$$

We are using the time variable to represent real time in seconds. The specification $t' \geq t{+}1$ represents the passage of at least 1 second; it can be implemented by **wait** (Section 5.3 and Exercise 332). The specification $t' \leq t{+}3$ represents the passage of at most 3 seconds; it requires the hardware executing the conjoined specification to be fast enough; in this case it is easy since instruction times are nanoseconds and the time bounds are seconds. The other time specifications are similar.

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 496 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.

$$GrowSlow \Leftarrow \textbf{if } t{=}2{\times}x \textbf{ then } alloc \parallel x{:=}\, t \textbf{ else } t{:=}\, t{+}1 \textbf{ fi}.\ GrowSlow$$

If the time is equal to $2{\times}x$, then one space is allocated, and concurrently 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.

$$GrowSlow = s < log\ t \Rightarrow (\forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < log\ t'')$$

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.

$$\begin{array}{ll} & \textbf{if } t{=}2{\times}x \textbf{ then } s{:=}\, s{+}1 \parallel x{:=}\, t \textbf{ else } t{:=}\, t{+}1 \textbf{ fi} \\ = & t{=}2{\times}x \wedge s'{=}s{+}1 \wedge x'{=}t \wedge t'{=}t{+}1 \ \vee\ t{\neq}2{\times}x \wedge s'{=}s \wedge x'{=}x \wedge t'{=}t{+}1 \end{array}$$

Now the refinement has the form

$$\begin{array}{lll} & (A{\Rightarrow}B \Leftarrow C{\vee}D.\ A{\Rightarrow}B) & \text{. distributes over } \vee \\ = & (A{\Rightarrow}B \Leftarrow (C.\ A{\Rightarrow}B) \vee (D.\ A{\Rightarrow}B)) & \text{antidistributive law} \\ = & (A{\Rightarrow}B \Leftarrow (C.\ A{\Rightarrow}B)) \wedge (A{\Rightarrow}B \Leftarrow (D.\ A{\Rightarrow}B)) & \text{portation twice} \\ = & (B \Leftarrow A \wedge (C.\ A{\Rightarrow}B)) \wedge (B \Leftarrow A \wedge (D.\ A{\Rightarrow}B)) & \end{array}$$

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:

$$s < \log t \ \wedge\ (t{=}2{\times}x \wedge s'{=}s{+}1 \wedge x'{=}t \wedge t'{=}t{+}1.\ s < \log t \Rightarrow \forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t'')$$

remove sequential composition, remembering that s is interactive

$$= \quad s < \log t \ \wedge\ (\exists x'', t'''{\cdot}\quad t{=}2{\times}x \wedge s'''{=}s{+}1 \wedge x''{=}t \wedge t'''{=}t{+}1 \wedge (s''' < \log t''' \Rightarrow \forall t''{\cdot}\ t''{\geq}t''' \Rightarrow s'' < \log t''))$$

Use $s'''{=}s{+}1$ and drop it. Use one-point to eliminate $\exists x'', t'''$.

$$\Rightarrow \quad s < \log t \ \wedge\ t{=}2{\times}x \ \wedge\ (s{+}1 < \log(t{+}1) \Rightarrow \forall t''{\cdot}\ t''{\geq}t{+}1 \Rightarrow s'' < \log t'')$$

The next step should be discharge. We need

$$s < \log t \ \wedge\ t{=}2{\times}x \Rightarrow s{+}1 < \log(t{+}1)$$
$$= \quad 2^s < t = 2{\times}x \Rightarrow 2^{s+1} < t{+}1$$
$$= \quad 2^s < t = 2{\times}x \Rightarrow 2^{s+1} \leq t$$
$$= \quad 2^s < t = 2{\times}x \Rightarrow 2^{s+1} \leq 2{\times}x$$
$$= \quad 2^s < t = 2{\times}x \Rightarrow 2^s \leq x$$
$$\Leftarrow \quad 2^s \leq x$$

This is the missing initialization of x. So we go back and redefine *GrowSlow*.

$$\mathit{GrowSlow} \ = \ s < \log t \ \wedge\ x{\geq}2^s \ \Rightarrow\ (\forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t'')$$

Now we redo the proof. First case:

$$s < \log t \ \wedge\ x{\geq}2^s \ \wedge\ (\ t{=}2{\times}x \wedge s'{=}s{+}1 \wedge x'{=}t \wedge t'{=}t{+}1.\ s < \log t \ \wedge\ x{\geq}2^s \Rightarrow \forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t'')$$

remove sequential composition, remembering that s is interactive

$$= \quad s < \log t \ \wedge\ x{\geq}2^s \ \wedge\ (\exists x'', t'''{\cdot}\quad t{=}2{\times}x \wedge s'''{=}s{+}1 \wedge x''{=}t \wedge t'''{=}t{+}1 \wedge (s''' < \log t''' \ \wedge\ x''{\geq}2^{s'''} \Rightarrow \forall t''{\cdot}\ t''{\geq}t''' \Rightarrow s'' < \log t''))$$

Use $s'''{=}s{+}1$ and drop it. Use one-point to eliminate $\exists x'', t'''$.

$$\Rightarrow \quad s < \log t \ \wedge\ x{\geq}2^s \ \wedge\ t{=}2{\times}x \ \wedge\ (s{+}1 < \log(t{+}1) \ \wedge\ t{\geq}2^{s+1} \ \Rightarrow\ \forall t''{\cdot}\ t''{\geq}t{+}1 \Rightarrow s'' < \log t'')$$

discharge, as calculated earlier

$$= \quad s < \log t \ \wedge\ x{\geq}2^s \ \wedge\ t{=}2{\times}x \ \wedge\ \forall t''{\cdot}\ t''{\geq}t{+}1 \Rightarrow s'' < \log t''$$

when $t''{=}t$, then $s''{=}s$ and since $s < \log t$, the domain of t'' can be increased

$$\Rightarrow \quad \forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t''$$

The second case is easier than the first.

$$s < \log t \ \wedge\ x{\geq}2^s \ \wedge\ (\ t{\neq}2{\times}x \wedge s'{=}s \wedge x'{=}x \wedge t'{=}t{+}1.\ s < \log t \ \wedge\ x{\geq}2^s \Rightarrow \forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t'')$$

remove sequential composition, remembering that s is interactive

$$= \quad s < \log t \ \wedge\ x{\geq}2^s \ \wedge\ (\exists x'', t'''{\cdot}\quad t{\neq}2{\times}x \wedge s'''{=}s \wedge x''{=}x \wedge t'''{=}t{+}1 \wedge (s''' < \log t''' \ \wedge\ x''{\geq}2^{s'''} \Rightarrow \forall t''{\cdot}\ t''{\geq}t''' \Rightarrow s'' < \log t''))$$

Use $s'''{=}s$ and drop it. Use one-point to eliminate $\exists x'', t'''$.

$$\Rightarrow \quad s < \log t \ \wedge\ x{\geq}2^s \ \wedge\ t{\neq}2{\times}x \ \wedge\ (s < \log t \ \wedge\ x{\geq}2^s \ \Rightarrow\ \forall t''{\cdot}\ t''{\geq}t{+}1 \Rightarrow s'' < \log t'')$$

discharge

$$= \quad s < \log t \ \wedge\ x{\geq}2^s \ \wedge\ t{\neq}2{\times}x \ \wedge\ \forall t''{\cdot}\ t''{\geq}t{+}1 \Rightarrow s'' < \log t''$$

when $t''{=}t$, then $s''{=}s$ and since $s < \log t$, the domain of t'' can be increased

$$\Rightarrow \quad \forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t''$$

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 concurrently. 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 $\mathcal{M}c$ and $\mathcal{T}c$ called the message script and the time script, and two extended natural variables $\boldsymbol{r}c$ and $\boldsymbol{w}c$ 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, from the channel. The write cursor is a state variable saying how many messages have been written, or output, to 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 $\mathcal{M}$, $\mathcal{T}$, $\boldsymbol{r}$, and $\boldsymbol{w}$.

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 will be sent on a channel may be unknown, but they can be referred to as items in the scripts. For example, after 3 more reads the next input on channel c will be $\mathcal{M}c_{\boldsymbol{r}c+3}$, and after 5 more writes the next output will be $\mathcal{M}c_{\boldsymbol{w}c+5}$ and it will occur at time $\mathcal{T}c_{\boldsymbol{w}c+5}$. Omitting the channel name from the script and cursor names, after 3 more reads the next input will be $\mathcal{M}_{\boldsymbol{r}+3}$, and after 5 more writes the next output will be $\mathcal{M}_{\boldsymbol{w}+5}$ at time $\mathcal{T}_{\boldsymbol{w}+5}$.

$$\begin{array}{lll} \mathcal{M} & = & 6\ ;\ 4\ ;\ 7\ ;\ 1\ ;\ 0\ ;\ 3\ ;\ 8\ ;\ 9\ ;\ 2\ ;\ 5\ ;\ \ldots \\ \mathcal{T} & = & 3\ ;\ 5\ ;\ 5\ ;\ 20\ ;\ 25\ ;\ 28\ ;\ 31\ ;\ 31\ ;\ 45\ ;\ 48\ ;\ \ldots \\ & & \quad\ \uparrow \quad\ \ \uparrow \\ & & \quad\ r \quad\ \ w \end{array}$$

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

if *even* $(\mathcal{M}c_r)$ **then** $\mathcal{M}d_w = \top$ **else** $\mathcal{M}d_w = \bot$ **fi**

or, more briefly,

$\mathcal{M}d_w$ = *even* $(\mathcal{M}c_r)$

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 way, 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

$t' \geq t \ \wedge\ r' \geq r \ \wedge\ w' \geq w$

An implementable specification can say what the scripts are in the segment written by a computation, that is the segment $\mathcal{M}_{w;..w'}$ and $\mathcal{T}_{w;..w'}$ 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 $\mathcal{M}$, and time script $\mathcal{T}$) is implementable if and only if

$$\begin{array}{ll} \forall \sigma, M, T \cdot \exists \sigma', \mathcal{M}, \mathcal{T} & S \ \wedge\ t' \geq t \ \wedge\ r' \geq r \ \wedge\ w' \geq w \\ & \wedge\ \mathcal{M}_{(0;..w);\,(w';..\infty)} = M_{(0;..w);\,(w';..\infty)} \\ & \wedge\ \mathcal{T}_{(0;..w);\,(w';..\infty)} = T_{(0;..w);\,(w';..\infty)} \\ & \wedge\ \forall i, j : w,..w' \cdot i \leq j \ \Rightarrow\ t \leq \mathcal{T}_i \leq \mathcal{T}_j \leq t' \end{array}$$

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.

End of Implementability

9.1.1 Input and Output

Here are four 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 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 binary expression meaning "there is unread input available on channel c". Here are the formal definitions.

$c!\ e \quad = \quad \mathcal{M}_w = e \ \wedge\ \mathcal{T}_w = t \ \wedge\ (w := w+1)$ "c output e"
$c? \quad = \quad r := r+1$ "c input"
$c \quad = \quad \mathcal{M}_{r-1}$
$\surd c \quad = \quad \mathcal{T}_r \le t$ "check c"

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." **fi**
else *screen*! "Are you there?" **fi**

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 $\mathcal{M}d_{wd} = even\ (\mathcal{M}c_{rc})$ given earlier.

$\mathcal{M}d_{wd} = even\ (\mathcal{M}c_{rc}) \quad \Leftarrow \quad c?.\ d!\ even\ c$

To prove the refinement, starting with the right side,

$c?.\ d!\ even\ c$
$= \quad rc := rc+1.\ \mathcal{M}d_{wd} = even\ (\mathcal{M}c_{rc-1}) \ \wedge\ \mathcal{T}d_{wd} = t \ \wedge\ (wd := wd+1)$
$= \quad \mathcal{M}d_{wd} = even\ (\mathcal{M}c_{rc}) \wedge \mathcal{T}d_{wd}=t \wedge rc'=rc+1 \wedge wc'=wc \wedge rd'=rd \wedge wd'=wd+1$
$\Longrightarrow \quad \mathcal{M}d_{wd} = even\ (\mathcal{M}c_{rc})$

A specification should be written as clearly, as understandably, as possible. A programmer refines the specification to obtain a program, which a computer can execute. In our example, the program seems more understandable than the specification! Whenever that is the case, we should consider using the program as the 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

$S \quad = \quad \forall n\colon nat\cdot \mathcal{M}d_{wd+n} = 2 \times \mathcal{M}c_{rc+n}$

The input and output may not be the first input and output ever on channels c and d. But 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.

$S \quad \Leftarrow \quad c?.\ d!\ 2{\times}c.\ S$

The proof is:

$c?.\ d!\ 2{\times}c.\ S$
$= \quad rc := rc+1.\ \mathcal{M}d_{wd} = 2 \times \mathcal{M}c_{rc-1} \ \wedge\ (wd := wd+1).\ \forall n\colon nat\cdot \mathcal{M}d_{wd+n} = 2 \times \mathcal{M}c_{rc+n}$
$= \quad rc := rc+1.\ \mathcal{M}d_{wd} = 2 \times \mathcal{M}c_{rc-1} \ \wedge\ \forall n\colon nat\cdot \mathcal{M}d_{wd+1+n} = 2 \times \mathcal{M}c_{rc+n}$
$= \quad \mathcal{M}d_{wd} = 2 \times \mathcal{M}c_{rc} \ \wedge\ \forall n\colon nat\cdot \mathcal{M}d_{wd+1+n} = 2 \times \mathcal{M}c_{rc+1+n}$
$= \quad \forall n\colon nat\cdot \mathcal{M}d_{wd+n} = 2 \times \mathcal{M}c_{rc+n}$
$= \quad S$

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 $\mathcal{M}c_{\mathcal{R}c}$. This message was or is or will be sent at time $\mathcal{T}c_{\mathcal{R}c}$. Its arrival time, according to the transit time measure, is $\mathcal{T}c_{\mathcal{R}c}+1$. So input becomes

$t := t \uparrow (\mathcal{T}c_{\mathcal{R}c} + 1).\ c?$

If the input has already arrived, $\mathcal{T}c_{\mathcal{R}c} + 1 \le t$, 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

$\surd c \ = \ \mathcal{T}c_{\mathcal{R}c} + 1 \le t$

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 515(a): Let W be "wait for input on channel c and then read it". Formally,

$W \ = \ t := t \uparrow (\mathcal{T}_{\mathcal{R}} + 1).\ c?$

Prove $W \ \Leftarrow$ **if** $\surd c$ **then** $c?$ **else** $t := t+1.\ W$ **fi** where time is an extended natural. 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:

	if $\surd c$ **then** $c?$ **else** $t := t+1.\ W$ **fi**	replace $\surd c$ and W
$=$	**if** $\mathcal{T}_{\mathcal{R}} + 1 \le t$ **then** $c?$ **else** $t := t+1.\ t := t \uparrow (\mathcal{T}_{\mathcal{R}} + 1).\ c?$ **fi**	
$=$	**if** $\mathcal{T}_{\mathcal{R}} + 1 \le t$ **then** $t := t.\ c?$ **else** $t := (t+1) \uparrow (\mathcal{T}_{\mathcal{R}} + 1).\ c?$ **fi**	
		If $\mathcal{T}_{\mathcal{R}} + 1 \le t$, then $t = t \uparrow (\mathcal{T}_{\mathcal{R}} + 1)$.
		If $\mathcal{T}_{\mathcal{R}} + 1 > t$ then $(t+1) \uparrow (\mathcal{T}_{\mathcal{R}} + 1) = \mathcal{T}_{\mathcal{R}} + 1 = t \uparrow (\mathcal{T}_{\mathcal{R}} + 1)$.
$=$	**if** $\mathcal{T}_{\mathcal{R}} + 1 \le t$ **then** $t := t \uparrow (\mathcal{T}_{\mathcal{R}} + 1).\ c?$ **else** $t := t \uparrow (\mathcal{T}_{\mathcal{R}} + 1).\ c?$ **fi**	
$=$	W	

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

$\forall n: nat \cdot \mathcal{M}d_{\mathcal{W}d+n} = 2 \times \mathcal{M}c_{\mathcal{R}c+n} \ \wedge \ \mathcal{T}d_{\mathcal{W}d+n} = t+n$

The strongest implementable solution is

$(\forall n: nat \cdot \mathcal{M}d_{\mathcal{W}d+n} = 2 \times \mathcal{M}c_{\mathcal{R}c+n} \ \wedge \ \mathcal{T}d_{\mathcal{W}d+n} = t+n) \ \wedge \ \mathcal{R}c' = \mathcal{W}d' = t' = \infty \ \wedge \ \mathcal{W}c' = \mathcal{W}c \ \wedge \ \mathcal{R}d' = \mathcal{R}d$

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

$\forall n: nat \cdot \mathcal{M}d_{\mathcal{W}d+n} = 2 \times \mathcal{M}c_{\mathcal{R}c+n} \ \wedge \ \mathcal{T}d_{\mathcal{W}d+n} = t+n \ \Leftarrow \ dbl$

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

$$\begin{aligned} dbl_1 &= c?.\ d!\ 2{\times}c.\ t{:=}\,t{+}1.\ dbl_0 \\ &= \mathbf{r}c{:=}\,\mathbf{r}c{+}1.\ \mathcal{M}d_{\mathbf{w}d} = 2 \times \mathcal{M}c_{\mathbf{r}c-1} \ \wedge\ \mathcal{T}d_{\mathbf{w}d} = t \ \wedge\ (\mathbf{w}d{:=}\,\mathbf{w}d{+}1).\ t{:=}\,t{+}1.\ \top \\ &= \mathcal{M}d_{\mathbf{w}d} = 2 \times \mathcal{M}c_{\mathbf{r}c} \ \wedge\ \mathcal{T}d_{\mathbf{w}d} = t \\ dbl_2 &= c?.\ d!\ 2{\times}c.\ t{:=}\,t{+}1.\ dbl_1 \\ &= \mathbf{r}c{:=}\,\mathbf{r}c{+}1.\ \mathcal{M}d_{\mathbf{w}d} = 2 \times \mathcal{M}c_{\mathbf{r}c-1} \ \wedge\ \mathcal{T}d_{\mathbf{w}d} = t \ \wedge\ (\mathbf{w}d{:=}\,\mathbf{w}d{+}1). \\ &\quad\ t{:=}\,t{+}1.\ \mathcal{M}d_{\mathbf{w}d} = 2 \times \mathcal{M}c_{\mathbf{r}c} \ \wedge\ \mathcal{T}d_{\mathbf{w}d} = t \\ &= \mathcal{M}d_{\mathbf{w}d} = 2 \times \mathcal{M}c_{\mathbf{r}c} \ \wedge\ \mathcal{T}d_{\mathbf{w}d} = t \ \wedge\ \mathcal{M}d_{\mathbf{w}d+1} = 2{\times}\mathcal{M}c_{\mathbf{r}c+1} \ \wedge\ \mathcal{T}d_{\mathbf{w}d+1} = t{+}1 \end{aligned}$$

and so on. The result of the construction

$$dbl_\infty = \forall n{:}\,nat{\cdot}\ \mathcal{M}d_{\mathbf{w}d+n} = 2 \times \mathcal{M}c_{\mathbf{r}c+n} \ \wedge\ \mathcal{T}d_{\mathbf{w}d+n} = t{+}n$$

is the weakest solution of the dbl fixed-point construction. If we begin recursive construction with $t'{\geq}t \wedge \mathbf{r}c'{\geq}\mathbf{r}c \wedge \mathbf{w}c'{\geq}\mathbf{w}c \wedge \mathbf{r}d'{\geq}\mathbf{r}d \wedge \mathbf{w}d'{\geq}\mathbf{w}d$ we get the 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 an output 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

$$merge = (c?.\ e!\ c) \vee (d?.\ e!\ d).\ merge$$

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 520(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.

$$\begin{aligned} timemerge = \quad & (\surd c \ \vee\ \mathcal{T}c_{\mathbf{r}c} \leq \mathcal{T}d_{\mathbf{r}d}) \ \wedge\ (c?.\ e!\ c) \\ \vee\ & (\surd d \ \vee\ \mathcal{T}c_{\mathbf{r}c} \geq \mathcal{T}d_{\mathbf{r}d}) \ \wedge\ (d?.\ e!\ d). \\ & timemerge \end{aligned}$$

To account for the time spent waiting for input, we should insert $t{:=}\,t{\uparrow}(\mathcal{T}_{\mathbf{r}} + 1)$ 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.

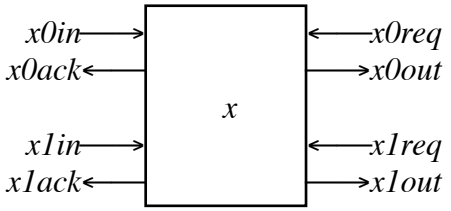
$$\begin{aligned} timemerge \Leftarrow\ & \textbf{if}\ \surd c\ \textbf{then}\ c?.\ e!\ c\ \textbf{else}\ ok\ \textbf{fi}. \\ & \textbf{if}\ \surd d\ \textbf{then}\ d?.\ e!\ d\ \textbf{else}\ ok\ \textbf{fi}. \\ & t{:=}\,t{+}1.\ timemerge \end{aligned}$$

where time is an extended natural.

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 of the next input on each of the input channels.

$$m = \Downarrow[\, \mathcal{T}x0in_{\mathbf{r}x0in} \,;\, \mathcal{T}x1in_{\mathbf{r}x1in} \,;\, \mathcal{T}x0req_{\mathbf{r}x0req} \,;\, \mathcal{T}x1req_{\mathbf{r}x1req} \,]$$

Then

$$\begin{aligned} monitor \;=\; & (\surd x0in \;\vee\; \mathcal{T}x0in_{\mathbf{r}x0in} = m) \;\wedge\; (x0in?.\; x{:=}\, x0in.\; x0ack!\, \top) \\ \vee\; & (\surd x1in \;\vee\; \mathcal{T}x1in_{\mathbf{r}x1in} = m) \;\wedge\; (x1in?.\; x{:=}\, x1in.\; x1ack!\, \top) \\ \vee\; & (\surd x0req \;\vee\; \mathcal{T}x0req_{\mathbf{r}x0req} = m) \;\wedge\; (x0req?.\; x0out!\, x) \\ \vee\; & (\surd x1req \;\vee\; \mathcal{T}x1req_{\mathbf{r}x1req} = m) \;\wedge\; (x1req?.\; x1out!\, x). \\ & monitor \end{aligned}$$

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 natural:

$$\begin{aligned} monitor \;\Leftarrow\; & \textbf{if}\ \surd x0in\ \textbf{then}\ x0in?.\; x{:=}\, x0in.\; x0ack!\, \top\ \textbf{else}\ ok\ \textbf{fi}. \\ & \textbf{if}\ \surd x1in\ \textbf{then}\ x1in?.\; x{:=}\, x1in.\; x1ack!\, \top\ \textbf{else}\ ok\ \textbf{fi}. \\ & \textbf{if}\ \surd x0req\ \textbf{then}\ x0req?.\; x0out!\, x\ \textbf{else}\ ok\ \textbf{fi}. \\ & \textbf{if}\ \surd x1req\ \textbf{then}\ x1req?.\; x1out!\, x\ \textbf{else}\ ok\ \textbf{fi}. \\ & t{:=}\, t{+}1.\; monitor \end{aligned}$$

We earlier solved Exercise 500 to specify a thermostat for a gas burner using interactive variables gas, $temperature$, $desired$, $flame$, and $spark$, as follows.

$$thermostat \;=\; (gas{:=}\, \bot \parallel spark{:=}\, \bot).\; GasIsOff$$

$$\begin{aligned} GasIsOff \;=\; & \textbf{if}\ temperature < desired - \varepsilon \\ & \textbf{then}\ (gas{:=}\, \top \parallel spark{:=}\, \top \parallel t' \geq t{+}1) \;\wedge\; t' \leq t{+}3.\; spark{:=}\, \bot.\; GasIsOn \\ & \textbf{else}\ ((\textbf{frame}\ gas, spark\cdot\ ok) \parallel t' \geq t) \;\wedge\; t' \leq t{+}1.\; GasIsOff\ \textbf{fi} \end{aligned}$$

$$\begin{aligned} GasIsOn \;=\; & \textbf{if}\ temperature < desired + \varepsilon \;\wedge\; flame \\ & \textbf{then}\ ((\textbf{frame}\ gas, spark\cdot\ ok) \parallel t' \geq t) \;\wedge\; t' \leq t{+}1.\; GasIsOn \\ & \textbf{else}\ (gas{:=}\, \bot \parallel (\textbf{frame}\ spark\cdot\ ok) \parallel t' \geq t{+}20) \;\wedge\; t' \leq t{+}21.\; GasIsOff\ \textbf{fi} \end{aligned}$$

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.

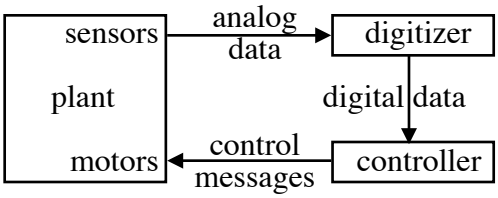
$$
\begin{array}{lll}
\mathit{thermostat} & = & ((\mathit{gasin}!\ \bot.\ \mathit{gasack}?) \parallel (\mathit{sparkin}!\ \bot.\ \mathit{sparkack}?)).\ \mathit{GasIsOff} \\
\\
\mathit{GasIsOff} & = & ((\mathit{temperaturereq}!\ \top.\ \mathit{temperature}?) \parallel (\mathit{desiredreq}!\ \top.\ \mathit{desired}?)). \\
 & & \textbf{if}\ \mathit{temperature} < \mathit{desired} - \varepsilon \\
 & & \textbf{then}\quad ((\mathit{gasin}!\ \top.\ \mathit{gasack}?) \parallel (\mathit{sparkin}!\ \top.\ \mathit{sparkack}?) \parallel t' \geq t{+}1) \\
 & & \qquad \wedge\ t' \leq t{+}3.\ \mathit{sparkin}!\ \bot.\ \mathit{sparkack}?.\ \mathit{GasIsOn} \\
 & & \textbf{else}\ t < t' \leq t{+}1.\ \mathit{GasIsOff}\ \textbf{fi} \\
\\
\mathit{GasIsOn} & = & (\ (\mathit{temperaturereq}!\ \top.\ \mathit{temperature}?) \parallel (\mathit{desiredreq}!\ \top.\ \mathit{desired}?) \\
 & & \parallel (\mathit{flamereq}!\ \top.\ \mathit{flame}?)\ ). \\
 & & \textbf{if}\ \mathit{temperature} < \mathit{desired} + \varepsilon\ \wedge\ \mathit{flame} \\
 & & \textbf{then}\ t < t' \leq t{+}1.\ \mathit{GasIsOn} \\
 & & \textbf{else}\ ((\mathit{gasin}!\ \bot.\ \mathit{gasack}?) \parallel t' \geq t{+}20)\ \wedge\ t' \leq t{+}21.\ \mathit{GasIsOff}\ \textbf{fi}
\end{array}
$$

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.

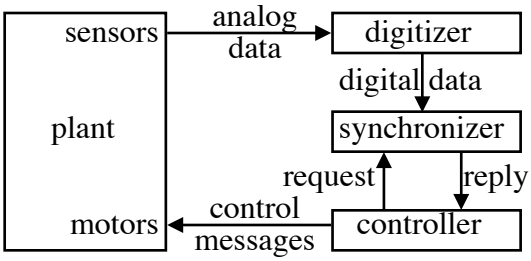
9.1.6 Reaction Controller

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 messages 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 messages 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 messages be produced as rapidly as digital data. But it is essential that each control message 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 three respects: first, there is only one writing process (digitizer) and one reading process (controller); second, the writing process does not need an acknowledgement; and last, the writing process is uniformly faster than the reading process. Here is its definition.

synchronizer = *digitaldata*?.
if √*request* **then** *request*? ‖ *reply*! *digitaldata* **else** *ok* **fi**.
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),

$c!\ 2.\ c?.\ x := c$
$= \quad \mathcal{M}_w = 2 \ \wedge\ w' = w+1 \ \wedge\ r' = r+1 \ \wedge\ x' = \mathcal{M}_r$

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 concurrent composition.

$c!\ 2 \parallel (c?.\ x := c)$
$= \quad \mathcal{M}_w = 2 \ \wedge\ w' = w+1 \ \wedge\ r' = r+1 \ \wedge\ x' = \mathcal{M}_r$

Again we cannot say $x'=2$ because there may be a previous unread output

$c!\ 1.\ (c!\ 2 \parallel (c?.\ x := c)).\ c?$

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. Here is a syntax and equivalent specification.

new $c?\ T\cdot S$
$= \quad \exists \mathcal{M}c\colon \infty^* T\cdot \exists \mathcal{T}c\colon \infty^* xnat\cdot$ **new** $rc, wc\colon xnat := 0\cdot S$
$= \quad \exists \mathcal{M}c\colon \infty^* T\cdot \exists \mathcal{T}c\colon \infty^* xnat\cdot \exists rc, rc', wc, wc'\colon xnat\cdot rc=wc=0 \ \wedge\ S$

This declaration introduces new channel c whose communications are of type T. The scope of the declaration is specification S. The declaration introduces two scripts, $\mathcal{M}c$ and $\mathcal{T}c$, which are infinite strings; they are not state variables, but state constants of unknown value (mathematical variables). The time script $\mathcal{T}c$ is a string of extended naturals ∞^*xnat, but it could be a string of nonnegative extended reals $\infty^*(\S r{:}\ xreal{\cdot}\ r{\geq}0)$. The channel declaration also introduces a read cursor $\boldsymbol{r}c$ with initial value 0 to say that initially there is no input on this channel, and a write cursor $\boldsymbol{w}c$ with initial value 0 to say that initially there is no output on this channel.

A local channel can be used without concurrency as a queue, or buffer. For example,

new $c?\ int{\cdot}\ c!\ 3.\ c!\ 4.\ c?.\ x{:=}\ c.\ c?.\ x{:=}\ x{+}c$

assigns 7 to x. Here is the proof, including time.

new $c?\ int{\cdot}\ c!\ 3.\ c!\ 4.\ t{:=}\ t{\uparrow}(\mathcal{T}_{\boldsymbol{r}}+1).\ c?.\ x{:=}\ c.\ t{:=}\ t{\uparrow}(\mathcal{T}_{\boldsymbol{r}}+1).\ c?.\ x{:=}\ x{+}c$

$=$ $\exists \mathcal{M}{:}\ \infty^*int{\cdot}\ \exists \mathcal{T}{:}\ \infty^*xnat{\cdot}\ \exists \boldsymbol{r}, \boldsymbol{r}', \boldsymbol{w}, \boldsymbol{w}'{:}\ xnat{\cdot}$
$\boldsymbol{r}{=}\boldsymbol{w}{=}0$
$\wedge\ (\mathcal{M}_{\boldsymbol{w}}=3\ \wedge\ \mathcal{T}_{\boldsymbol{w}}=t\ \wedge\ (\boldsymbol{w}{:=}\ \boldsymbol{w}{+}1).$
$\mathcal{M}_{\boldsymbol{w}}=4\ \wedge\ \mathcal{T}_{\boldsymbol{w}}=t\ \wedge\ (\boldsymbol{w}{:=}\ \boldsymbol{w}{+}1).$
$t{:=}\ t{\uparrow}(\mathcal{T}_{\boldsymbol{r}}+1).\ \boldsymbol{r}{:=}\ \boldsymbol{r}{+}1.$
$x{:=}\ \mathcal{M}_{\boldsymbol{r}-1}.$
$t{:=}\ t{\uparrow}(\mathcal{T}_{\boldsymbol{r}}+1).\ \boldsymbol{r}{:=}\ \boldsymbol{r}{+}1.$
$x{:=}\ x+\mathcal{M}_{\boldsymbol{r}-1})$

now use one-point laws and the Substitution Law several times

$=$ $\exists \mathcal{M}{:}\ \infty^*int{\cdot}\ \exists \mathcal{T}{:}\ \infty^*xnat{\cdot}$
$\mathcal{M}_0{=}3\ \wedge\ \mathcal{T}_0{=}t\ \wedge\ \mathcal{M}_1{=}4\ \wedge\ \mathcal{T}_1{=}t\ \wedge\ x'=\mathcal{M}_0+\mathcal{M}_1$
$\wedge\ t'=t{\uparrow}(\mathcal{T}_0+1){\uparrow}(\mathcal{T}_1+1)\ \wedge$ (other variables unchanged)

$=$ $x'{=}7\ \wedge\ t'=t{+}1\ \wedge$ (other variables unchanged)

Here are two processes with a communication between them. Ignoring time,

new $c?\ int{\cdot}\ c!\ 2\ \|\ (c?.\ x{:=}\ c)$ Use the definition of local channel declaration, and use the previous result for the concurrent composition

$=$ $\exists \mathcal{M}{:}\ \infty^*int{\cdot}\ \exists \boldsymbol{r}, \boldsymbol{r}', \boldsymbol{w}, \boldsymbol{w}'{:}\ xnat{\cdot}\ \boldsymbol{r}{=}\boldsymbol{w}{=}0$
$\wedge\ \mathcal{M}_{\boldsymbol{w}}{=}2\ \wedge\ \boldsymbol{w}'=\boldsymbol{w}{+}1\ \wedge\ \boldsymbol{r}'=\boldsymbol{r}{+}1\ \wedge\ x'{=}\mathcal{M}_{\boldsymbol{r}}\ \wedge$ (other variables unchanged)

use one-point law four times

$=$ $\exists \mathcal{M}{:}\ \infty^*int{\cdot}\ \mathcal{M}_0{=}2\ \wedge\ x'{=}\mathcal{M}_0\ \wedge$ (other variables unchanged)
$=$ $x'{=}2\ \wedge$ (other variables unchanged)
$=$ $x{:=}\ 2$

Replacing 2 by an arbitrary expression, we have a general theorem equating communication on a local channel with assignment. If we had included transit time, the result would have been

$x'{=}2\ \wedge\ t'=t{+}1\ \wedge$ (other variables unchanged)
$=$ $x{:=}\ 2\ \|\ t{:=}\ t{+}1$

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 (Exercise 528(a)). Inserting the input wait into

new $c?\ int{\cdot}\ c?.\ c!\ 5$

gives us

new c? int· $t := t\uparrow(\mathcal{T}_r + 1)$. $c?$. $c!\ 5$

$=$ $\exists \mathcal{M}$: ∞*int· $\exists \mathcal{T}$: ∞*$xnat$· $\exists r, r', w, w'$: $xnat$·
$r = w = 0 \wedge (t := t\uparrow(\mathcal{T}_r + 1).\ r := r+1.\ \mathcal{M}_w = 5 \wedge \mathcal{T}_w = t \wedge (w := w+1))$

We'll do this one slowly. First, expand $w := w+1$, taking r, w, x, and t as the state variables.

$=$ $\exists \mathcal{M}$: ∞*int· $\exists \mathcal{T}$: ∞*$xnat$· $\exists r, r', w, w'$: $xnat$·
$r = w = 0$
$\wedge\ (t := t\uparrow(\mathcal{T}_r+1).\ r := r+1.\ \mathcal{M}_w = 5 \wedge \mathcal{T}_w = t \wedge r' = r \wedge w' = w+1 \wedge x' = x \wedge t' = t)$

Now use the Substitution and one-point Laws.

$=$ $\exists \mathcal{M}$: ∞*int· $\exists \mathcal{T}$: ∞*$xnat$· $\mathcal{M}_0 = 5 \wedge \mathcal{T}_0 = t\uparrow(\mathcal{T}_0 + 1) \wedge x' = x \wedge t' = t\uparrow(\mathcal{T}_0 + 1)$

Look at the conjunct $\mathcal{T}_0 = t\uparrow(\mathcal{T}_0 + 1)$. It says $\mathcal{T}_0 = \infty$.

$=$ $x' = x \wedge t' = \infty$

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 Subsection 8.1.2. 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 and two local channels (Exercise 528(b)).

new c, d? int· $(c?.\ d!\ 6) \parallel (d?.\ c!\ 7)$

Inserting the input waits, we get

new c, d? int· $(t := t\uparrow(\mathcal{T}c_{rc} + 1).\ c?.\ d!\ 6) \parallel (t := t\uparrow(\mathcal{T}d_{rd} + 1).\ d?.\ c!\ 7)$

after a little work, we obtain

$=$ $\exists \mathcal{M}c, \mathcal{M}d$: ∞*int· $\exists \mathcal{T}c, \mathcal{T}d$: ∞*$xnat$· $\exists rc, rc', wc, wc', rd, rd', wd, wd'$: $xnat$·
$\mathcal{M}d_0 = 6 \wedge \mathcal{T}d_0 = t\uparrow(\mathcal{T}c_0 + 1) \wedge \mathcal{M}c_0 = 7 \wedge \mathcal{T}c_0 = t\uparrow(\mathcal{T}d_0 + 1)$
$\wedge\ rc' = wc' = rd' = wd' = 1 \wedge x' = x \wedge t' = t\uparrow(\mathcal{T}c_0 + 1)\uparrow(\mathcal{T}d_0 + 1))$

The conjuncts $\mathcal{T}d_0 = t\uparrow(\mathcal{T}c_0 + 1)$ and $\mathcal{T}c_0 = t\uparrow(\mathcal{T}d_0 + 1)$ imply $\mathcal{T}d_0 = \mathcal{T}c_0 = \infty$.

$=$ $x' = x \wedge t' = \infty$

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 concurrent 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

$P.\ (Q \parallel R).\ S$

and suppose P, Q, and S write to channel c, and P, R, and S read from channel c. The messages written by Q follow those written by P, and those written by S follow those written by Q. The messages read by R follow those read by P, and those read by S 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 concurrent processes. In the program structure

$P.\ (Q \parallel R \parallel S).\ T$

we might want Q to write, and both of R and S to read, all 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: what is the final value of the read cursor for the concurrent composition? The read cursors for R and S both start with the same value, but they may not end with the same value. The final value of the read cursor for $(Q \parallel R \parallel S)$ is needed because it becomes the initial value of the read cursor for T. The problem is similar to what happens to the time variable; it starts the same for Q, R, and S, but it may not end the same. The solution is to say that the final time for the concurrent composition is the maximum of the final times of the processes. The same solution works for the read cursor. For each channel, the final value of the read cursor in a concurrent composition is the maximum of the final values of the read cursors of the processes.

End of Broadcast

9.1.10 Power Series Multiplication

To end, we present an 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 526 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 concurrently 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 concurrently 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: $a_n = \mathcal{M}a_{\mathbf{r}a+n}$, $b_n = \mathcal{M}b_{\mathbf{r}b+n}$, and $c_n = \mathcal{M}c_{\mathbf{r}c+n}$. The question relieves us from concern with input times, but we are still concerned with output times. Let us use A, B, and C for the power series, so we can express our desired result as

$C = A{\times}B \ \wedge\ \forall n{\cdot}\ \mathcal{T}c_{\mathbf{w}c+n} = t{+}n$

We can calculate the output coefficients as follows.

$$\begin{aligned} & A{\times}B \\ = \quad & (a_0 + a_1{\times}x + a_2{\times}x^2 + a_3{\times}x^3 + ...) \times (b_0 + b_1{\times}x + b_2{\times}x^2 + b_3{\times}x^3 + ...) \\ = \quad & a_0{\times}b_0 + (a_0{\times}b_1 + a_1{\times}b_0){\times}x + (a_0{\times}b_2 + a_1{\times}b_1 + a_2{\times}b_0){\times}x^2 \\ & + (a_0{\times}b_3 + a_1{\times}b_2 + a_2{\times}b_1 + a_3{\times}b_0){\times}x^3 + ... \end{aligned}$$

from which we see $c_n = \Sigma i{:}\ 0{,}..n{+}1{\cdot}\ a_i{\times}b_{n-i}$.

Output coefficient n requires $n{+}1$ multiplications and n additions from $2{\times}(n{+}1)$ input coefficients, and must be produced 1 time unit after the previous coefficient. To accomplish this requires more and more data storage, and more and more concurrency, 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, $a_0{\times}b_0$, 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 $A{\times}B$,

$$\begin{aligned} = \quad & a_0{\times}b_0 + (a_0{\times}(b_1 + B_2{\times}x) + (a_1 + A_2{\times}x){\times}b_0){\times}x + A_1{\times}B_1{\times}x^2 \\ = \quad & a_0{\times}b_0 + (a_0{\times}b_1 + a_1{\times}b_0){\times}x + (a_0{\times}B_2 + A_1{\times}B_1 + A_2{\times}b_0){\times}x^2 \end{aligned}$$

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 $\langle c?\ T{\cdot}\ S\rangle$. 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 variable 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.

$$P \;{=}\!{=}\; \langle c?\ rat{\cdot}\ C = A{\times}B\rangle$$

We refine $P\,c$ as follows.

$$\begin{aligned} P\,c \quad {\Longleftarrow} \quad & (a? \parallel b?).\ c!\ a{\times}b. \\ & \textbf{new}\ a0{:}\ rat := a{\cdot}\ \ \textbf{new}\ b0{:}\ rat := b{\cdot}\ \ \textbf{new}\ d?\ rat{\cdot} \\ & P\,d \parallel ((a? \parallel b?).\ c!\ a0{\times}b + a{\times}b0.\ C = a0{\times}B + D + A{\times}b0) \end{aligned}$$

$$C = a0{\times}B + D + A{\times}b0 \quad {\Longleftarrow} \quad (a? \parallel b? \parallel d?).\ c!\ a0{\times}b + d + a{\times}b0.\ C = a0{\times}B + D + A{\times}b0$$

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, concurrently with the remainder of the program. Both $P\,d$ and its concurrent process will be reading from channels a and b using separate read cursors. Concurrent 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, leaving out time.

$(a? \parallel b?).\ c!\ a{\times}b.$
new $a0{:}\ rat := a{\cdot}$ **new** $b0{:}\ rat := b{\cdot}$ **new** $d?\ rat{\cdot}$
$P\ d \parallel ((a? \parallel b?).\ c!\ a0{\times}b + a{\times}b0.\ C = a0{\times}B + D + A{\times}b0)$

$=$ $(\mathbf{r}a := \mathbf{r}a{+}1 \parallel \mathbf{r}b := \mathbf{r}b{+}1).\ \mathcal{M}c_{\mathbf{w}c} = \mathcal{M}a_{\mathbf{r}a-1} \times \mathcal{M}b_{\mathbf{r}b-1} \ \wedge\ (\mathbf{w}c := \mathbf{w}c{+}1).$
$\exists a0, a0', b0, b0', \mathcal{M}d, \mathbf{r}d, \mathbf{r}d', \mathbf{w}d, \mathbf{w}d'{\cdot}$
$a0 := \mathcal{M}a_{\mathbf{r}a-1}.\ b0 := \mathcal{M}b_{\mathbf{r}b-1}.\ \mathbf{r}d := 0.\ \mathbf{w}d := 0.$
$(\forall n{\cdot}\ \mathcal{M}d_{\mathbf{w}d+n} = (\Sigma i{:}\ 0,..n{+}1{\cdot}\ \mathcal{M}a_{\mathbf{r}a+i} \times \mathcal{M}b_{\mathbf{r}b+n-i}))$
$\wedge\ ((\mathbf{r}a := \mathbf{r}a{+}1 \parallel \mathbf{r}b := \mathbf{r}b{+}1).\ \mathcal{M}c_{\mathbf{w}c} = a0{\times}\mathcal{M}b_{\mathbf{r}b-1} + \mathcal{M}a_{\mathbf{r}a-1}{\times}b0 \wedge (\mathbf{w}c := \mathbf{w}c{+}1).$
$\forall n{\cdot}\ \mathcal{M}c_{\mathbf{w}c+n} = a0 \times \mathcal{M}b_{\mathbf{r}b+n} + \mathcal{M}d_{\mathbf{r}d+n} + \mathcal{M}a_{\mathbf{r}a+n} \times b0)$
Make all substitutions indicated by assignments.

$=$ $\mathcal{M}c_{\mathbf{w}c} = \mathcal{M}a_{\mathbf{r}a} \times \mathcal{M}b_{\mathbf{r}b}$
$\wedge\ \exists a0, a0', b0, b0', \mathcal{M}d, \mathbf{r}d, \mathbf{r}d', \mathbf{w}d, \mathbf{w}d'{\cdot}$
$(\forall n{\cdot}\ \mathcal{M}d_n = \Sigma i{:}\ 0,..n{+}1{\cdot}\ \mathcal{M}a_{\mathbf{r}a+1+i} \times \mathcal{M}b_{\mathbf{r}b+1+n-i})$
$\wedge\ \mathcal{M}c_{\mathbf{w}c+1} = \mathcal{M}a_{\mathbf{r}a} \times \mathcal{M}b_{\mathbf{r}b+1} + \mathcal{M}a_{\mathbf{r}a+1} \times \mathcal{M}b_{\mathbf{r}b}$
$\wedge\ (\forall n{\cdot}\ \mathcal{M}c_{\mathbf{w}c+2+n} = \mathcal{M}a_{\mathbf{r}a} \times \mathcal{M}b_{\mathbf{r}b+2+n} + \mathcal{M}d_n + \mathcal{M}a_{\mathbf{r}a+2+n} \times \mathcal{M}b_{\mathbf{r}b})$
Use the first universal quantification to replace $\mathcal{M}d_n$ in the second.
Then throw away the first universal quantification (weakening our expression).
Now all existential quantifications are unused, and can be thrown away.

$\Longrightarrow$ $\mathcal{M}c_{\mathbf{w}c} = \mathcal{M}a_{\mathbf{r}a} \times \mathcal{M}b_{\mathbf{r}b}$
$\wedge\ \mathcal{M}c_{\mathbf{w}c+1} = \mathcal{M}a_{\mathbf{r}a} \times \mathcal{M}b_{\mathbf{r}b+1} + \mathcal{M}a_{\mathbf{r}a+1} \times \mathcal{M}b_{\mathbf{r}b}$
$\wedge\ \forall n{\cdot}\ \mathcal{M}c_{\mathbf{w}c+2+n} = \mathcal{M}a_{\mathbf{r}a} \times \mathcal{M}b_{\mathbf{r}b+2+n} + (\Sigma i{:}\ 0,..n{+}1{\cdot}\ \mathcal{M}a_{\mathbf{r}a+1+i} \times \mathcal{M}b_{\mathbf{r}b+1+n-i}) + \mathcal{M}a_{\mathbf{r}a+2+n} \times \mathcal{M}b_{\mathbf{r}b}$
Now put the three conjuncts together.

$=$ $\forall n{\cdot}\ \mathcal{M}c_{\mathbf{w}c+n} = \Sigma i{:}\ 0,..n{+}1{\cdot}\ \mathcal{M}a_{\mathbf{r}a+i} \times \mathcal{M}b_{\mathbf{r}b+n-i}$
$=$ $P\ c$

We still have to prove the loop refinement.

$(a? \parallel b? \parallel d?).\ c!\ a0{\times}b + d + a{\times}b0.\ C = a0{\times}B + D + A{\times}b0$

$=$ $(ra := ra{+}1 \parallel rb := rb{+}1 \parallel rd := rd{+}1).$
$\mathcal{M}c_{\mathbf{w}c} = a0 \times \mathcal{M}b_{\mathbf{r}b-1} + \mathcal{M}d_{\mathbf{r}d-1} + \mathcal{M}a_{\mathbf{r}a-1} \times b0 \ \wedge\ (\mathbf{w}c := \mathbf{w}c{+}1).$
$\forall n{\cdot}\ \mathcal{M}c_{\mathbf{w}c+n} = a0 \times \mathcal{M}b_{\mathbf{r}b+n} + \mathcal{M}d_{\mathbf{r}d+n} + \mathcal{M}a_{\mathbf{r}a+n} \times b0$
Make all substitutions indicated by assignments.

$=$ $\mathcal{M}c_{\mathbf{w}c} = a0 \times \mathcal{M}b_{\mathbf{r}b} + \mathcal{M}d_{\mathbf{r}d} + \mathcal{M}a_{\mathbf{r}a} \times b0$
$\wedge\ \forall n{\cdot}\ \mathcal{M}c_{\mathbf{w}c+1+n} = a0 \times \mathcal{M}b_{\mathbf{r}b+1+n} + \mathcal{M}d_{\mathbf{r}d+1+n} + \mathcal{M}a_{\mathbf{r}a+1+n} \times b0$
Put the two conjuncts together.

$=$ $\forall n{\cdot}\ \mathcal{M}c_{\mathbf{w}c+n} = a0 \times \mathcal{M}b_{\mathbf{r}b+n} + \mathcal{M}d_{\mathbf{r}d+n} + \mathcal{M}a_{\mathbf{r}a+n} \times b0$
$=$ $C = a0{\times}B + D + A{\times}b$

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.

$$\begin{aligned} Q\ c \ \Longleftarrow\ & (a? \parallel b?).\ c!\ a{\times}b. \\ & \textbf{new}\ a0{:}\ rat := a{\cdot}\ \ \textbf{new}\ b0{:}\ rat := b{\cdot}\ \ \textbf{new}\ d?\ rat{\cdot} \\ & (t{:=}\ t{+}1.\ \ Q\ d)\ \parallel\ ((a? \parallel b?).\ \ t{:=}\ t{+}1.\ \ c!\ a0{\times}b + a{\times}b0.\ \ R) \end{aligned}$$

$$R \ \Longleftarrow\ (a? \parallel b? \parallel (t{:=}\ t{\uparrow}(\mathcal{T}d_{\boldsymbol{r}d} + 1).\ \ d?)).\ \ c!\ a0{\times}b + d + a{\times}b0.\ \ t{:=}\ t{+}1.\ \ R$$

where Q and R are defined as follows:

$$\begin{aligned} Q\ c\ &=\ \forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}n \\ Q\ d\ &=\ \forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{w}d+n} = t{+}n \\ R\ &=\ (\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n)\ \Rightarrow\ (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}1{+}n) \end{aligned}$$

Within loop R, the assignment $t{:=}\ t{\uparrow}(\mathcal{T}d_{\boldsymbol{r}d} + 1)$ represents a delay of 1 time unit the first iteration (because $t = \mathcal{T}d_{\boldsymbol{r}d}$), and a delay of 0 time units each subsequent iteration (because $t = \mathcal{T}d_{\boldsymbol{r}d} + 1$). This makes the proof very ugly. To make the proof pretty, we can replace $t{:=}\ t{\uparrow}(\mathcal{T}d_{\boldsymbol{r}d} + 1)$ by $t{:=}\ (t{+}1){\uparrow}(\mathcal{T}d_{\boldsymbol{r}d} + 1)$ 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 $t{:=}\ (t{+}1){\uparrow}(\mathcal{T}d_{\boldsymbol{r}d} + 1)$ 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

$$\begin{aligned} Q\ c \ \Longleftarrow\ & (a? \parallel b?).\ c!\ a{\times}b. \\ & \textbf{new}\ a0{:}\ rat := a{\cdot}\ \ \textbf{new}\ b0{:}\ rat := b{\cdot}\ \ \textbf{new}\ d?\ rat{\cdot} \\ & (t{:=}\ t{+}1.\ \ Q\ d)\ \parallel\ ((a? \parallel b?).\ \ t{:=}\ t{+}1.\ \ c!\ a0{\times}b + a{\times}b0.\ \ R) \end{aligned}$$

$$R \ \Longleftarrow\ (a? \parallel b? \parallel (t{:=}\ (t{+}1){\uparrow}(\mathcal{T}d_{\boldsymbol{r}d} + 1).\ \ d?)).\ \ c!\ a0{\times}b + d + a{\times}b0.\ \ R$$

Here is the proof of the first of these refinements, beginning with the right side

$$\begin{aligned} & (a? \parallel b?).\ c!\ a{\times}b. \\ & \textbf{new}\ a0{:}\ rat := a{\cdot}\ \ \textbf{new}\ b0{:}\ rat := b{\cdot}\ \ \textbf{new}\ d?\ rat{\cdot} \\ & (t{:=}\ t{+}1.\ \ Q\ d)\ \parallel\ ((a? \parallel b?).\ \ t{:=}\ t{+}1.\ \ c!\ a0{\times}b + a{\times}b0.\ \ R) \end{aligned}$$

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 $b0$.

$$\begin{aligned} \Longrightarrow\ & \mathcal{T}c_{\boldsymbol{w}c} = t\ \wedge\ (\boldsymbol{w}c{:=}\ \boldsymbol{w}c{+}1). \\ & \exists \mathcal{T}d, \boldsymbol{r}d, \boldsymbol{r}d', \boldsymbol{w}d, \boldsymbol{w}d'{\cdot}\ \ \boldsymbol{r}d{=}\boldsymbol{w}d{=}\ 0 \\ \wedge\ & (\quad (t{:=}\ t{+}1.\ \ \forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{w}d+n} = t{+}n) \\ & \quad \wedge\ (\ t{:=}\ t{+}1.\ \ \mathcal{T}c_{\boldsymbol{w}c} = t\ \wedge\ (\boldsymbol{w}c{:=}\ \boldsymbol{w}c{+}1). \\ & \qquad (\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n) \Rightarrow (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}1{+}n)\ )\ ) \end{aligned}$$

Make all substitutions indicated by assignments and one-point.

$$\begin{aligned} \Longrightarrow\ & \mathcal{T}c_{\boldsymbol{w}c} = t \\ \wedge\ & \exists \mathcal{T}d{\cdot}\quad (\forall n{\cdot}\ \mathcal{T}d_n = t{+}1{+}n) \\ & \qquad \wedge\ \ \mathcal{T}c_{\boldsymbol{w}c+1} = t{+}1 \\ & \qquad \wedge\ \ ((\forall n{\cdot}\ \mathcal{T}d_n = t{+}1{+}n) \Rightarrow (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+2+n} = t{+}2{+}n)) \end{aligned}$$

Use the first universal quantification to discharge the antecedent.

Then throw away the first universal quantification (weakening our expression).

Now the existential quantification is unused, and can be thrown away.

$$\Longrightarrow\quad \mathcal{T}c_{\boldsymbol{w}c} = t\ \wedge\ \mathcal{T}c_{\boldsymbol{w}c+1} = t{+}1\ \wedge\ \forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+2+n} = t{+}2{+}n$$

Now put the three conjuncts together.

$$\begin{aligned} =\quad & \forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}n \\ =\quad & Q\ c \end{aligned}$$

We still have to prove the loop refinement.

$$(R \Leftarrow (a? \parallel b? \parallel (t{:=}(t{+}1)\uparrow(\mathcal{T}d_{\boldsymbol{r}d}+1).\ d?)).\ c!\ a0{\times}b + d + a{\times}b0.\ R)$$

Ignore $a?$ and $b?$ and the output message.

$$\Leftarrow \quad ((\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n) \Rightarrow (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}1{+}n))$$

$$\Leftarrow\ (\ t{:=}(t{+}1)\uparrow(\mathcal{T}d_{\boldsymbol{r}d}+1).\ \boldsymbol{r}d{:=}\boldsymbol{r}d{+}1.\ \mathcal{T}c_{\boldsymbol{w}c} = t\ \wedge (\boldsymbol{w}c{:=}\boldsymbol{w}c{+}1).$$
$$(\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n) \Rightarrow (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}1{+}n)\ )$$

Use the Law of Portation to move the first antecedent to the right side, where it becomes a conjunct.

$$= \quad (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}1{+}n)$$
$$\Leftarrow \quad (\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n)$$
$$\wedge\ (\ t{:=}(t{+}1)\uparrow(\mathcal{T}d_{\boldsymbol{r}d}+1).\ \boldsymbol{r}d{:=}\boldsymbol{r}d{+}1.\ \mathcal{T}c_{\boldsymbol{w}c} = t\ \wedge (\boldsymbol{w}c{:=}\boldsymbol{w}c{+}1).$$
$$(\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n) \Rightarrow (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}1{+}n)\ )$$

Specializing $\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n$ to the case $n{=}0$, we use $\mathcal{T}d_{\boldsymbol{r}d}{=}t$ to simplify $(t{+}1)\uparrow(\mathcal{T}d_{\boldsymbol{r}d}+1)$.

$$= \quad (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}1{+}n)$$
$$\Leftarrow \quad (\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n)$$
$$\wedge\ (\ t{:=}t{+}1.\ \boldsymbol{r}d{:=}\boldsymbol{r}d{+}1.\ \mathcal{T}c_{\boldsymbol{w}c} = t\ \wedge (\boldsymbol{w}c{:=}\boldsymbol{w}c{+}1).$$
$$(\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n) \Rightarrow (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}1{+}n)\ )$$

Make all substitutions indicated by assignments.

$$= \quad (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}1{+}n)$$
$$\Leftarrow \quad (\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n)$$
$$\wedge\ \mathcal{T}c_{\boldsymbol{w}c} = t{+}1$$
$$\wedge\ ((\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+1+n} = t{+}1{+}n) \Rightarrow (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+1+n} = t{+}2{+}n))$$

The conjunct $\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+n} = t{+}n$ discharges the antecedent $\forall n{\cdot}\ \mathcal{T}d_{\boldsymbol{r}d+1+n} = t{+}1{+}n$ which can be dropped.

$$\Leftarrow \quad (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+n} = t{+}1{+}n)$$
$$\Leftarrow\ \mathcal{T}c_{\boldsymbol{w}c} = t{+}1\ \wedge\ (\forall n{\cdot}\ \mathcal{T}c_{\boldsymbol{w}c+1+n} = t{+}2{+}n)$$

$$= \quad \top$$

End of Power Series Multiplication

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. First problem: 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 if the answers are at least reasonable. Second problem: 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 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 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 proved. 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. Solutions to exercises are at hehner.ca/aPToP/solutions , or click on the exercise number.

10.0 Introduction

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?

1 Jack is looking at Anne. Anne is looking at George. Jack is married. George is single. Is a married person looking at a single person? (yes) (no) (cannot be determined)

2 Here are three statements.
(i) Exactly one of these three statements is false.
(ii) Exactly two of these three statements are false.
(iii) All three of these three statements are false.
Which of these statements are true, and which are false?

3 Here are five statements.
(i) This statement is true.
(ii) This statement is false.
(iii) This statement is either true or false.
(iv) This statement is neither true nor false.
(v) This statement is both true and false.
Which of these statements are
(a) true?
(b) false?
(c) either true or false?
(d) neither true nor false?
(e) both true and false?

End of Introduction

10.1 Basic Theories

4 Value tables and the Evaluation Rule can be replaced by some new axioms and antiaxioms. For example, one value 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 value tables as axioms and antiaxioms as succinctly as possible.

5 Simplify each of the following binary expressions.
(a) $x \wedge \neg x$
(b) $x \vee \neg x$
(c) $x \Rightarrow \neg x$
(d) $x \Leftarrow \neg x$
(e) $x = \neg x$
(f) $x \neq \neg x$

6 Prove each of the following laws of Binary Theory using the proof format given in Subsection 1.0.1, and any laws listed in Section 11.3. Do not use the Completion Rule.

(a) $a \wedge b \Rightarrow a \vee b$
(b) $(a \wedge b) \vee (b \wedge c) \vee (c \wedge a) = (a \vee b) \wedge (b \vee c) \wedge (c \vee a)$
(c) $\neg a \Rightarrow (a \Rightarrow b)$
(d) $a = (b \Rightarrow a) = a \vee b$
(e) $a = (a \Rightarrow b) = a \wedge b$
(f) $(a \Rightarrow c) \wedge (b \Rightarrow \neg c) \Rightarrow \neg(a \wedge b)$
(g) $a \wedge \neg b \Rightarrow a \vee b$
(h) $(a \Rightarrow b) \wedge (c \Rightarrow d) \wedge (a \vee c) \Rightarrow (b \vee d)$
(i) $a \wedge \neg a \Rightarrow b$
(j) $(a \Rightarrow b) \vee (b \Rightarrow a)$
(k)√ $\neg(a \wedge \neg(a \vee b))$
(l) $(\neg a \Rightarrow \neg b) \wedge (a \neq b) \vee (a \wedge c \Rightarrow b \wedge c)$
(m) $(a \Rightarrow \neg a) \Rightarrow \neg a$
(n) $(a \Rightarrow b) \wedge (\neg a \Rightarrow b) = b$
(o) $(a \Rightarrow b) \Rightarrow a = a$
(p) $a = b \vee a = c \vee b = c$
(q) $a \wedge b \vee a \wedge \neg b = a$
(r) $a \Rightarrow (b \Rightarrow a)$
(s) $a \Rightarrow a \wedge b = a \Rightarrow b = a \vee b \Rightarrow b$
(t) $(a \Rightarrow a \wedge b) \vee (b \Rightarrow a \wedge b)$
(u) $(a \Rightarrow (p = x)) \wedge (\neg a \Rightarrow p) = p = (x \vee \neg a)$
(v) $(a \Rightarrow b \Rightarrow \neg a) \vee (b \wedge c \Rightarrow a \wedge c)$
(w) $a = (b \wedge c) \wedge d = (\neg b \wedge \neg c) \wedge e = ((a \vee d) = c) \Rightarrow e = b$

7 Prove each of the following laws of Binary Theory using the proof format given in Subsection 1.0.1, and any laws listed in Section 11.3. Do not use the Completion Rule.

(a) **if** a **then** a **else** $\neg a$ **fi**
(b) **if** b **then** c **else** $\neg c$ **fi** = **if** c **then** b **else** $\neg b$ **fi**
(c) **if** $b \wedge c$ **then** P **else** Q **fi** = **if** b **then if** c **then** P **else** Q **fi else** Q **fi**
(d) **if** $b \vee c$ **then** P **else** Q **fi** = **if** b **then** P **else if** c **then** P **else** Q **fi fi**
(e) **if** b **then** P **else if** b **then** Q **else** R **fi fi** = **if** b **then** P **else** R **fi**
(f) **if if** b **then** c **else** d **fi then** P **else** Q **fi**
= **if** b **then if** c **then** P **else** Q **fi else if** d **then** P **else** Q **fi fi**
(g) **if** b **then if** c **then** P **else** R **fi else if** c **then** Q **else** R **fi fi**
= **if** c **then if** b **then** P **else** Q **fi else** R **fi**
(h) **if** b **then if** c **then** P **else** R **fi else if** d **then** Q **else** R **fi fi**
= **if if** b **then** c **else** d **fi then if** b **then** P **else** Q **fi else** R **fi**

8 The Case Analysis Laws equate the three-operand operator **if** a **then** b **else** c **fi** 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.

9 Consider a fully bracketed expression containing only the symbols $\top$ $\bot$ $=$ $\neq$ ( ) 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 $\bot$, $\bot$ for $\top$, $=$ for $\neq$, $\neq$ for $=$.

10 Express formally and succinctly that exactly one of three statements is true.

11 (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 binary operators, there is 1 pair of duals, and 2 operators that are their own duals. Find them.

(b) Of the 16 two-operand binary 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 fi** ? Express it using only the operator **if then else fi** .

(d) The dual of a binary expression without variables is formed as follows: replace each operator with its dual, adding brackets if necessary to maintain the precedence. Explain why the dual of a theorem is an antitheorem, and vice versa.

(e) Let P be a binary expression without variables. From part (d) we know that every binary expression without variables of the form

$$(\text{dual of } P) = \neg P$$

is a theorem. Therefore, to find the dual of a binary expression with variables, we must replace each operator by its dual and negate each variable. For example, if a and b are binary 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 value tables that are each other's vertical mirror reflections. For example, the value table for $\wedge$ (below left) is the vertical mirror reflection of the value table for $\vee$ (below right).

$\wedge$:		
	$\top\top$	$\top$
	$\top\bot$	$\bot$
	$\bot\top$	$\bot$
	$\bot\bot$	$\bot$

$\vee$:		
	$\top\top$	$\top$
	$\top\bot$	$\top$
	$\bot\top$	$\top$
	$\bot\bot$	$\bot$

Design symbols (you may redesign existing symbols where necessary) for the 4 one-operand and 16 two-operand binary 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.

12 Formalize each of the following statements. For each of the ten pairs of statements, 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.

13 Design symbols for the 10 two-operand binary operators that are not presented in Section 1.0, and find laws about these operators.

14 Complete the following laws of Binary 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 fi** $a\ b$ ( )

That's $8 \times 5 = 40$ questions.

15 (BDD) A BDD (Binary Decision Diagram) is a binary expression that has one of the following 3 forms: $\top$, $\bot$, **if** variable **then** BDD **else** BDD **fi** . For example,

if x **then if** a **then** $\top$ **else** $\bot$ **fi else if** y **then if** b **then** $\top$ **else** $\bot$ **fi else** $\bot$ **fi fi**

is a BDD. An OBDD (Ordered BDD) is a BDD with an ordering on the variables, and in each **if then else fi** , 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** $\bot$ **fi else if** b **then if** c **then** $\top$ **else** $\bot$ **fi else** $\bot$ **fi fi**

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 **fi**

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 **fi**

bob = **if** a **then** alice **else** false **fi**

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 fi** , the **then**- and **else**-parts differ. 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 **fi** 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 $\top$)?

16 (degenerate) There are 256 operators with 3 binary operands and a binary result. How many of them are degenerate? An operator is degenerate if its result can be expressed without using all the operands.

17 Formalize each of the following statements as a binary 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 binary expressions like (the door can be opened) for the parts that cannot make use of binary operators, but for words like "only if" you should use binary operators. You translate meanings from words to binary symbols; the meaning of the words may depend on their context and 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.

18 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.

19 (pigs) Here is a sentence: "If this sentence is true, then pigs can fly.". What are the consequences of the fact that this sentence has been written?

20 A sign says:

Using variables like t for time of day and d for day of week, write a binary expression that says when there is no parking.

21 (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)

22 (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?

23 Let p and q be binary 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?

24 (caskets) There are two caskets; one is gold and one is silver. In one casket there is a million dollars, and the other is empty. On the gold casket there is an inscription: the money is not in here. On the silver casket there is an inscription: exactly one of these inscriptions is true. Each inscription is either true or false (not both). On the basis of the inscriptions, find the money.

25 (the unexpected egg) There are two boxes; one is red and the other is 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.

26 (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) P says: “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: “Q is 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: “Q is 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?

(g) P says that Q and R are the same (both knaves or both knights). Someone asks R whether P and Q are the same. What is R's answer?

(h) P, Q, and R each say: “The other two are knaves.”. How many knaves are there?

27 (pirate gold) Islands X and Y contain knights who always tell the truth, knaves who always lie, and possibly 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?

28 (doorway to heaven) There is a door that leads either to heaven or to hell. There is a guard who knows where the door leads. If you ask the guard a question, the guard may tell the truth, or the guard may lie. Ask the guard one question to determine whether the door leads to heaven or hell.

29 (bracket algebra) Here is a new way to write binary expressions. An expression can be empty; in other words, nothing is already an expression. If you put a pair of brackets 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 brackets around an expression is bracket algebra's way of negating it; 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 proved.

(a) Rewrite the binary 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 binary expression

$(\neg a\Rightarrow\neg b)\ \wedge\ (a\neq b)\ \vee\ (a\wedge c\Rightarrow b\wedge c)$

as a bracket expression, and then prove it following the rules of bracket algebra.

(c) Can all binary 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 Binary Theory, rewritten reasonably directly as bracket expressions, be proved using the rules of bracket algebra?

(f) We interpret empty as $\top$, brackets as negation, and adjacency as conjunction. Is there any other consistent way to interpret the symbols and rules of bracket algebra?

30 (hats) There are three people, named Front, Middle, and Back, standing in a queue. Each is wearing a hat, which is either red or blue. Back can see Middle's hat and Front's hat, but cannot see Back's hat. Middle can see Front's hat, but cannot see Back's hat or Middle's hat. Front cannot see anyone's hat. A passerby tells them that at least one of them is wearing a red hat. Back then says: I don't know what color hat I'm wearing. Middle then says: I don't know what color hat I'm wearing. Front then says: I do know what color hat I'm wearing. Formalize Front's reasoning, and determine what color hat Front is wearing.

31 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.

32 A number can be written as a sequence of decimal digits. Consider using the sequence notation with arbitrary expressions, not just digits. For example, $1(2+3)4 = 154$. What changes are needed to the number axioms?

33 (scale) There is a tradition in programming languages to use a scale operator, e , in the limited context of digit sequences. Thus $12e3 = 12{\times}10^3$. Consider using the scale notation with arbitrary expressions, not just digits. For example, $(6+6)e(5-2) = 12e3$. What changes are needed to the number axioms?

34 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 11.3.2 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 ∞ ?

35 Prove $-\infty<y<\infty \land y\neq 0 \Rightarrow (x/y=z \;=\; x=z{\times}y)$.

36 Show that the number axioms become inconsistent when we add the axiom

$$-\infty<y<\infty \;\Rightarrow\; x/y{\times}y = x$$

37 (circular numbers) Redesign the axioms for the extended number system to make it circular, so that $+\infty = -\infty$. Be careful with the transitivity of $<$.

38 Let $\bullet$ be a two-operand infix operator (precedence 3) whose operands and result are of some type T. Let $\lozenge$ be a two-operand infix operator (precedence 7) whose operands are of type T and whose result is binary, defined by the axiom

$$a \lozenge b \;=\; a \bullet b = a$$

(a) Prove if $\bullet$ is idempotent then $\lozenge$ is reflexive.

(b) Prove if $\bullet$ is associative then $\lozenge$ is transitive.

(c) Prove if $\bullet$ is symmetric then $\lozenge$ is antisymmetric.

(d) If T is the binary values and $\bullet$ is $\land$, what is $\lozenge$?

(e) If T is the binary values and $\bullet$ is $\lor$, what is $\lozenge$?

(f) If T is the natural numbers and $\lozenge$ is $\leq$, what is $\bullet$?

(g) The axiom defines $\lozenge$ in terms of $\bullet$. Can it be inverted, so that $\bullet$ is defined in terms of $\lozenge$?

39 Is there any harm in adding the axiom $0/0=5$ to Number Theory?

40 (family theory) Design a theory of personal relationships. Invent person expressions such as *Jack* , *Jill* , *father of p* , *mother of p* . Invent binary 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

41 Simplify
(a) $(1, 7–3) + 4 – (2, 6, 8)$
(b) $nat×nat$
(c) $nat–nat$
(d) $(nat+1)×(nat+1)$

42 Prove $\neg\ 7: null$.

43 We defined bunch *null* with the axiom $null: A$. Is there any harm in defining bunch *all* with the axiom $A: all$?

44 Let x be an element, and let A be any bunch. Prove $\neg\ x: A \Leftarrow \not\subset(A`x) = 0$ without using the law $\neg\ x: A = \not\subset(A`x) = 0$.

45 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?

46 (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.

47 Let $\otimes$ be a two-operand infix operator (precedence 3) with natural operands and an extended natural result. Informally, $n\otimes m$ means "the number of times that n is a factor of m ". It is defined by the following two axioms.

$$m: n×nat \ \vee\ n\otimes m = 0$$
$$n≠0 \ \Rightarrow\ n\otimes(m×n) = n\otimes m + 1$$

(a) Make a 3×3 chart of the values of $(0,..3)\otimes(0,..3)$.
(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 $\otimes$ to have extended natural operands?

48 Let A be a bunch of binary values such that $A = \neg A$. What is A?

49 For naturals n and m, we can express the statement "n is a factor of m" formally as

$m: n{\times}nat$

(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?

50 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.

51 Let $B = 1, 3, 5$. What is

(a) $¢(B + B)$
(b) $¢(B \times 2)$
(c) $¢(B \times B)$
(d) $¢(B^2)$

52 The compound axiom says

$x: A, B = x: A \vee x: B$

There are 16 two-operand binary operators that could sit where $\vee$ sits in this axiom if we just replace bunch union (,) by a corresponding bunch operator. Which of the 16 two-operand binary operators correspond to useful bunch operators?

53 (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

54 (Cantor's paradise) Show that $¢ǂB > ¢B$ is neither a theorem nor an antitheorem.

55 Prove

(a) $\$S = ¢{\sim}S$
(b) $A{\in}S = A{:}{\sim}S$

56 Evaluate (no proof)

(a) $¢\ null$
(b) $¢\ nil$
(c) $↔\ null$
(d) $↔\ nil$
(e) $ǂ\ null$
(f) $ǂ\ nil$

57 Strings defined in Section 2.2 have natural indexes and extended natural lengths. Add a new operator, the inverse of join, to obtain strings that have negative indexes and lengths.

58 (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.

59 In Section 2.2 there is a self-describing expression

“““0;0;(0;..29);28;28;(1;..28)”””0;0;(0;..29);28;28;(1;..28)

which evaluates to its own representation.

(a) Write an expression that evaluates to twice its own representation. In other words, it evaluates to its own representation followed by its own representation again.

(b) Make it into a self-printing program. Let's say that $!e$ prints the value of expression e.

60 Suppose we add laws to allow various operators to distribute over string join (semicolon). For example, if i and j are items and s and t are strings, then the laws

$$nil + nil = nil$$
$$(i; s) + (j; t) = i+j; s+t$$

say that strings are added item by item (a sum of strings is a string of sums). For example,

$$(2; 4; 7) + (3; 9; 1) = 5; 13; 8$$

What string f is defined by

$$f = 0; 1; f + f_{1;..\infty}$$

61 (string replacement) Let S and T be strings. Let n and m be such that

$$n, m: 0,..\leftrightarrow S+1 \wedge n \leq m$$

Design a notation and axiom for a string expression that means a string like S except that the substring of S from index n to index m is replaced by string T. If $n=m$ then it is insertion of T at index n. If $T=nil$ then it is deletion of the substring from n to m. If $n=m=\leftrightarrow S$ then it is appending T to the end of S. If $n=m=0$ then it is prepending T to the front of S.

62 Simplify (no proof)

(a) $null, nil$

(b) $null; nil$

(c) $*nil$

(d) $[null]$

(e) $[*null]$

63 What is the difference between $[0, 1, 2]$ and $[0; 1; 2]$?

64 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) $[0][0][0][0]$

(c) $((3;2) \rightarrow [10;..15] \mid 3 \rightarrow [5;..10] \mid [0;..5])\ 3$

(d) $([0;..5]\ [3; 4])\ 1$

(e) $(2;2) \rightarrow \text{“j”} \mid [[\text{“abc”}]; [\text{“de”}]; [\text{“fghi”}]]$

(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) $[3; (4, 5); 6; (7, 8, 9)]\ \text{`}\ [3; 4; (5, 6); (7, 8)]$

65 Simplify, assuming i: $\square L$
(a) $i \to L\, i \mid L$
(b) $(L\, i \to i \mid L)\, i$
(c) $L\, [0;..i] ;; [x] ;; L\, [i+1;..\#L]$

66 Let i and j be indexes of list L. Express $i \to L\, j \mid j \to L\, i \mid L$ without using $\mid$.

67 Which step(s) in this proof is(are) wrong?
(a) $2 = 2^1 = 2^{2 \times 1/2} = (2^2)^{1/2} = 4^{1/2} = ((-2)^2)^{1/2} = (-2)^{2 \times 1/2} = (-2)^1 = -2$
(b) $4 = 4^1 = 4^{1/2+1/2} = 4^{1/2} \times 4^{1/2} = (2,-2) \times (2,-2) = 4,-4$

68 (order; research) There is an ordering on extended real numbers. Using that, we defined an ordering on strings of extended real numbers. Using that, we defined an ordering on lists of extended real numbers. This exercise is to explore an extension that connects these orders, and extends them to sets. Let A be any bunch (anything), and let S be any string. Add the axioms
$A < \{A\}$
$S < [S]$
so that an increase in structure is an increase in the order. What correspondence can you make between this order and Cantor's order $¢A < ¢ǂA$ in the higher cardinals?

End of Basic Data Structures

10.3 Function Theory

69 In each of the following, replace p by
$\langle x{:}\ int{\cdot}\ \langle y{:}\ int{\cdot}\ \langle z{:}\ int{\cdot}\ x{\geq}0 \wedge x^2{\leq}y \wedge \forall z{:}\ int{\cdot}\ z^2{\leq}y \Rightarrow z{\leq}x\rangle\rangle\rangle$
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) $p\ (x+z)\ (y+y)\ (2+z)$

70 Some mathematicians like to use a notation like $\exists!x{:}\ D{\cdot}\ P\, x$ to mean "there is a unique x in D such that $P\, x$ holds". Define $\exists!$ formally.

71 Write an expression equivalent to each of the following without using §.
(a) $¢(§x{:}\ D{\cdot}\ P\, x) = 0$
(b) $¢(§x{:}\ D{\cdot}\ P\, x) = 1$
(c) $¢(§x{:}\ D{\cdot}\ P\, x) = 2$

72 Simplify without proof
(a) $§n{:}\ nat{\cdot}\ \exists m{:}\ nat{\cdot}\ n = m^2$
(b) $§n{:}\ nat{\cdot}\ \exists m{:}\ nat{\cdot}\ n^2 = m \Rightarrow n = m^2$

73 (join) Define function $join$ to apply to a list of lists and produce their join. For example,
$join\ [[0; 1; 2]; [nil]; [[3]]; [4; 5]] = [0; 1; 2; [3]; 4; 5]$

74 Express formally that L is a sublist (not necessarily consecutive items) of list M. For example, $[0; 2; 1]$ is, but $[2; 0; 1]$ is not, a sublist of $[0; 1; 2; 2; 1; 0]$.

75 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).

76 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.

77 (sick) John says "If one of us gets sick, then we all get sick.". Mary disagrees. She says "If one of us stays well, then we all stay well.". Settle the argument.

78 (liar) Epimenides was a Cretan who said "All Cretans are liars."; this has become known as the liar's paradox. Formalize and analyze this sentence.

79 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
That's $3\times3 = 9$ questions.

80 (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.
Therefore
(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 last statement.

81 (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).

82 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 by anyone without a key.
(c) The locksmith can open any door even without a key.

83 Prove that if variables i and j do not appear in predicates P and Q, then
$$(\forall i\cdot P\ i) \Rightarrow (\exists i\cdot Q\ i) \ = \ (\exists i, j\cdot P\ i \Rightarrow Q\ j)$$

84 There are four binary two-operand associative symmetric operators with an identity. We used two of them to define quantifiers. What happened to the other two?

85 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?

86 Define quantifier $\boxdot$ to give the deep domain of a list or function. Here are some examples.
$$\boxdot[[0; 1]; [0; 1]; [0; 1]] \ = \ 0;0,\ 0;1,\ 1;0,\ 1;1,\ 2;0\ \ 2;1$$
$$\boxdot[10; [11; 12]; 13] \ = \ 0,\ 1;0,\ 1;1,\ 2$$
$$\boxdot\langle x{:}\ nat\cdot \langle y{:}\ nat\cdot\ x{+}y\rangle\rangle \ = \ nat; nat$$
$$\boxdot\langle x{:}\ 0,..4\cdot \langle y{:}\ 0,..x\cdot\ x{+}y\rangle\rangle \ = \ 1;0,\ 2;0,\ 2;1,\ 3;0,\ 3;1,\ 3;2$$

87 Define $\approx$ to give the deep contents of a string or set or list or function. Here are some examples.
$$\approx(10; 11; 12; 13) \ = \ 10, 11, 12, 13$$
$$\approx\{10, \{11, 12\}, 13\} \ = \ 10, 11, 12, 13$$
$$\approx[10; [11; 12]; 13] \ = \ 10, 11, 12, 13$$
$$\approx\{10, [11; 12], 13\} \ = \ 10, 11, 12, 13$$
$$\approx[10; \{11, 12\}; 13] \ = \ 10, 11, 12, 13$$
$$\approx\langle x{:}\ nat\cdot \langle y{:}\ nat\cdot\ 2{\times}x{\times}y\rangle\rangle \ = \ 2{\times}nat$$
$$\approx\langle x{:}\ 0,..4\cdot \langle y{:}\ 0,..x\cdot\ x{+}y\rangle\rangle \ = \ 1, 2, 3, 4, 5$$
The contents operator $\sim$ removes one level of structure from a set or list. The deep contents operator $\approx$ removes all levels of structure.

88 Exercise 12 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?

89 Formalize each of the following statements as a binary 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 altitude 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.
(g) I don't agree with anything you say.
(h) I don't agree with everything you say.

90 (baby) Formalize the statements
Everyone loves my baby.
My baby loves only me.
I am my baby.
and prove that the first two statements imply the last statement.

91 (drink) There are some people in a bar. Formalize and prove the statement “There's a person in the bar such that, if that person drinks, then everyone in the bar drinks.”.

92 Simplify (all domains are *nat*)
(a) $\forall y\cdot\ y = x+2\ \Rightarrow\ y>5$
(b) $\forall y\cdot\ y = x+2\ \vee\ y = x+1\ \Rightarrow\ y>5$

93 Prove $((\exists x\cdot P\ x)\Rightarrow(\forall x\cdot R\ x \Rightarrow Q\ x))\wedge(\exists x\cdot P\ x \vee Q\ x)\wedge(\forall x\cdot Q\ x \Rightarrow P\ x) \Rightarrow (\forall x\cdot R\ x \Rightarrow P\ x)$

94(a) If P: $bin\to bin$ is monotonic, prove $(\exists x\cdot P\ x) = P\ \top$ and $(\forall x\cdot P\ x) = P\ \bot$.
(b) If P: $bin\to bin$ is antimonotonic, prove $(\exists x\cdot P\ x) = P\ \bot$ and $(\forall x\cdot P\ x) = P\ \top$.

95 (bitonic) 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.

96 Formalize and disprove the statement “There is a natural number that is not equal to any natural number.”.

97 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.

98 (friends) Formalize and prove the statement “The people you know are those known by all who know all whom you know.”.

99 (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.

100 Prove that the square of an odd natural number is odd, and the square of an even natural number is even.

101 Express $\forall$ and $\exists$ in terms of $\not\in$ and $\S$.

102 Simplify
(a) $\Sigma\,((0,..n) \rightarrow m)$
(b) $\Pi\,((0,..n) \rightarrow m)$
(c) $\forall\,((0,..n) \rightarrow b)$
(d) $\exists\,((0,..n) \rightarrow b)$

103 Are the binary expressions
$nil \rightarrow x \;=\; x$
$(S;T) \rightarrow x \;=\; S \rightarrow T \rightarrow x$
(a) consistent with the theory in Chapters 2 and 3?
(b) theorems according to the theory in Chapters 2 and 3?

104 (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?

105 (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.

106 (Russell's paradox) Define $rus \;=\; \langle f{:}\; null \rightarrow bin \cdot\; \neg\, f\, f \rangle$.
(a) Can we prove $rus\; rus \;=\; \neg\; rus\; rus$?
(b) Is this an inconsistency?
(c) Can we add the axiom $\neg\; f{:}\; \Box f$? Would it help?

107 (Cantor's diagonal) Prove $\neg \exists f{:}\; nat \rightarrow nat \rightarrow nat \cdot\; \forall g{:}\; nat \rightarrow nat \cdot\; \exists n{:}\; nat \cdot\; f\, n = g$.

108 (Gödel/Turing incompleteness) Prove that we cannot consistently and completely define a total deterministic interpreter. An interpreter is a predicate $\mathbb{I}$ that applies to texts; when applied to a text representing a binary expression, its result is equal to the represented expression. For example,
$\mathbb{I}$ "$\forall s{:}\; [{*}char] \cdot\; \#s \geq 0$" $=\; \forall s{:}\; [{*}char] \cdot\; \#s \geq 0$

109 Let f and g be functions from nat to nat .
(a) For what f do we have the theorem $f\, g = g$?
(b) For what f do we have the theorem $g\, f = g$?

110 What is the difference between $\#[n{*}\top]$ and $\not\in\S[n{*}\top]$?

111 Without using the Bounding Laws, prove
(a) $\forall i \cdot\; L\, i \leq m \;=\; \Uparrow L \leq m$
(b) $\exists i \cdot\; L\, i \leq m \;=\; \Downarrow L \leq m$

112 If $f{:}\; A \rightarrow B$ and $p{:}\; B \rightarrow bin$, prove
(a) $\exists b{:}\; f\, A \cdot\; p\, b \;=\; \exists a{:}\; A \cdot\; p\, f\, a$
(b) $\forall b{:}\; f\, A \cdot\; p\, b \;=\; \forall a{:}\; A \cdot\; p\, f\, a$

113 Let all variables range over the reals. Prove

$$n{=}m \;=\; \forall k{\cdot}\ (k{\le}n){=}(k{\le}m)$$

114 We can express "there is a smallest natural number" as follows:

$$\exists n{:}\ nat{\cdot}\ \forall m{:}\ nat{\cdot}\ n{\le}m$$

(a) Now how do we say "Denote that smallest natural number by 0 ." formally? In other words, how do we say "Let's call that smallest natural number 0 ." formally?

(b) Prove that there are not two different natural numbers that are tied for smallest.

115 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 $\langle v{\cdot}\ b\rangle$ where v is a variable and b is any expression (the body; no domain). There is an Application Law

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

and an Extension Law

$$f \;=\; \langle v{\cdot}\ f\ v\rangle$$

Let a be binary, and let b be any expression. Then $a \mathbin{»} b$ is an expression of the same type as b . The $»$ operator has precedence level 12 and is right-associating. Its axioms include:

$$\top \mathbin{»} b \;=\; b$$
$$a \mathbin{»} b \mathbin{»} c \;=\; a{\wedge}b \mathbin{»} c$$

The expression $a \mathbin{»} b$ 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 $\langle n{\cdot}\ n{:}\ nat \mathbin{»} n{+}1\rangle$ can be abbreviated $\langle n{:}\ nat{\cdot}\ n{+}1\rangle$. Applying this function to 3 , we find

$$\begin{aligned} & \langle n{\cdot}\ n{:}\ nat \mathbin{»} n{+}1\rangle\ 3 \\ = \quad & 3{:}\ nat \mathbin{»} 3{+}1 \\ = \quad & \top \mathbin{»} 4 \\ = \quad & 4 \end{aligned}$$

Applying it to -3 we find

$$\begin{aligned} & \langle n{\cdot}\ n{:}\ nat \mathbin{»} n{+}1\rangle\ (-3) \\ = \quad & -3{:}\ nat \mathbin{»} -3{+}1 \\ = \quad & \bot \mathbin{»} -2 \end{aligned}$$

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?

116 Is there any harm in defining relation R with the following axioms?

$\forall x{\cdot}\ \exists y{\cdot}\ R\ x\ y$	totality
$\forall x{\cdot}\ \neg\ R\ x\ x$	irreflexivity
$\forall x, y, z{\cdot}\ R\ x\ y \wedge R\ y\ z \Rightarrow R\ x\ z$	transitivity
$\exists u{\cdot}\ \forall x{\cdot}\ x{=}u \vee R\ x\ u$	unity

117 Let n be a natural number, and let R be a relation on $0,..n$. In other words,
$R: (0,..n) \to (0,..n) \to bin$
We say that from x we can reach x in zero steps. If $R\,x\,y$ we say that from x we can reach y in one step. If $R\,x\,y$ and $R\,y\,z$ 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.

118 Relation R is transitive if $\forall x, y, z \cdot R\,x\,y \wedge R\,y\,z \Rightarrow R\,x\,z$. Express formally that relation R is the transitive closure of relation Q (R is the strongest transitive relation that is implied by Q).

119 (pigeon-hole) Let n be natural and let L be a finite list of finite numbers. Prove
$\Sigma L > n \times \#L \;\Rightarrow\; \exists i: \Box L \cdot L\,i > n$

120 Most arithmetic expressions can be evaluated. For example, $2+3$ evaluates to 5. Our evaluation rules do not give us any answer for $1/0$. That's because there is no single answer that makes sense. We might look at $\Updownarrow n \cdot 1/(1/n)$, which is ∞. But we might equally well look at $\Updownarrow n \cdot 1/(-1/n)$, which is $-\infty$. We already have some arithmetic expressions that evaluate to a bunch of answers. For example, $4^{1/2}$ evaluates to $2, -2$. So perhaps it makes sense to say $1/0 = \infty, -\infty$. What sense can we make of $0/0$?

End of Function Theory

10.4 Program Theory

121√ 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) $x := y+1.\ y' = 2 \times x$
(d) $x := 1.\ x \geq 1 \Rightarrow \exists x \cdot y' = 2 \times x$
(e) $x := y.\ x \geq 1 \Rightarrow \exists y \cdot y' = x \times y$
(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** $x := y$ **fi**
(k) $x := 1.\ x' > x.\ x' = x+1$

122 Prove specification S is satisfiable for prestate σ if and only if $(S.\ \top)$. Note: $\top$ is the "true" or "top" binary.

123 Let x be an integer state variable. Is the following specification implementable?
(a) $x \geq 0 \;\Rightarrow\; x'^{\,2} = x$
(b) $x' \geq 0 \;\Rightarrow\; x = 0$
(c) $\neg(x \geq 0 \;\wedge\; x' = 0)$
(d) $\neg(x \geq 0 \;\vee\; x' = 0)$

124 Let n be a natural state variable. Is the following specification implementable?
(a) $n := n-1$
(b) $n > 0 \Rightarrow (n := n-1)$
(c) **if** $n > 0$ **then** $n := n-1$ **else** ok **fi**

125 Here are four specifications in integer variables x and y.
(i) $x{:=}\,2.\ y{:=}\,3$
(ii) $x'{=}2.\ y'{=}3$
(iii) $(x{:=}\,2) \wedge (y{:=}\,3)$
(iv) $x'{=}2 \wedge y'{=}3$
(a) Which of them make the final value of x be 2 and the final value of y be 3?
(b) Which of them are implementable, and which are unimplementable?
(c) Which of them are deterministic, and which are nondeterministic?
(d) If the state variables are x, y, and z, which of them are deterministic, and which are nondeterministic?

126 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 $(S.\,S)$.

127 Prove or disprove
(a) $R.$ **if** b **then** P **else** Q **fi** $=$ **if** b **then** $R.\,P$ **else** $R.\,Q$ **fi**
(b) **if** b **then** $P \Rightarrow Q$ **else** $R \Rightarrow S$ **fi** $=$ **if** b **then** P **else** R **fi** $\Rightarrow$ **if** b **then** Q **else** S **fi**
(c) **if** b **then** $P.\,Q$ **else** $R.\,S$ **fi** $=$ **if** b **then** P **else** R **fi**. **if** b **then** Q **else** S **fi**

128 Prove

$$(R \Leftarrow P.\ \textbf{if}\ b\ \textbf{then}\ ok\ \textbf{else}\ R\ \textbf{fi}) \wedge (W \Leftarrow \textbf{if}\ b\ \textbf{then}\ ok\ \textbf{else}\ P.\,W\ \textbf{fi})$$
$$\Leftarrow (R \Leftarrow P.\,W) \wedge (W \Leftarrow \textbf{if}\ b\ \textbf{then}\ ok\ \textbf{else}\ R\ \textbf{fi})$$

129 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 $P \wedge R$.

130 (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 $S \Leftarrow A.\,S.\,Z$?

131 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) $P.\,Q = \neg(P.\,\neg Q)$

132 What is wrong with the following proof:

	$(R \Leftarrow R.\,S)$	use context rule
$=$	$(R \Leftarrow \bot.\,S)$	$\bot$ is base for $.$
$=$	$(R \Leftarrow \bot)$	base law for $\Leftarrow$
$=$	$\top$	

133 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 **fi** and $E \Leftarrow$ **if** b **then** F **else** G **fi** are theorems,
then $A \vee E \Leftarrow$ **if** b **then** $C \vee F$ **else** $D \vee G$ **fi** 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.

134 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] .

135 Let x and n be natural variables. Find a specification P such that both the following refinements can be proved:

$x = x' \times 2^{n'} \Leftarrow n := 0.\ P$

$P \Leftarrow$ **if** *even* x **then** $x := x/2.\ n := n+1.\ P$ **else** *ok* **fi**

136 Let x and y be binary variables. Simplify

(a) $x := x=y.\ x := x=y$

(b) $x := x \neq y.\ y := x \neq y.\ x := x \neq y$

137 Let a , b , and c be integer variables. Express as simply as possible without using quantifiers, assignments, or sequential compositions

(a) $b := a-b.\ b := a-b$

(b) $a := a+b.\ b := a-b.\ a := a-b$

(c) $c := a-b-c.\ b := a-b-c.\ a := a-b-c.\ c := a+b+c$

(d) $a := a+b.\ b := a+b.\ c := a+b$

(e) $a := a+b.\ b' = a+b.\ c := a+b$

(f) $a := a+b+1.\ b := a-b-1.\ a := a-b-1$

(g) $a' = a+b+1.\ b' = a-b-1$

(h) $a := a-b.\ b := a-b.\ a := a+b$

138 (factorial) In natural variables n and f prove

$f := n! \Leftarrow$ **if** $n=0$ **then** $f := 1$ **else** $n := n-1.\ f := n!.\ n := n+1.\ f := f \times n$ **fi**

where $n! = 1 \times 2 \times 3 \times ... \times n$.

139 In natural variables n and m prove

$P \Leftarrow n := n+1.$ **if** $n=10$ **then** *ok* **else** $m := m-1.\ P$ **fi**

where $P = m := m+n-9.\ n := 10$.

140 Let n and s be natural variables. The program

$R \Leftarrow s := 0.\ Q$

$Q \Leftarrow$ **if** $n=0$ **then** *ok* **else** $n := n-1.\ s := s+n.\ Q$ **fi**

adds up the first n natural numbers. Define R and Q appropriately, and prove the two refinements.

141 Let s and n be number variables. Let Q be a specification defined as

$Q = s' = s + n \times (n-1)/2$

(a) Prove the refinement

$Q \Leftarrow n := n-1.\ s := s+n.\ Q$

(b) Add time according to the recursive measure, replace Q by an implementable timing specification, and reprove the refinement.

142 (square) Let s and n be natural variables. Find a specification P such that both the following refinements can be proved:

$s' = n^2 \Leftarrow s := n.\ P$

$P \Leftarrow$ **if** $n=0$ **then** *ok* **else** $n := n-1.\ s := s+n+n.\ P$ **fi**

This program squares using only addition, subtraction, and test for zero.

143 In natural variables s and n prove

$$P \Leftarrow \textbf{if } n{=}0 \textbf{ then } ok \textbf{ else } n{:=}\, n{-}1.\ s{:=}\, s{+}2^n{-}n.\ t{:=}\, t{+}1.\ P \textbf{ fi}$$

where $P = s' = s + 2^n - n{\times}(n{-}1)/2 - 1 \ \wedge\ n'{=}0 \ \wedge\ t' = t{+}n$.

144 Let x be an integer variable. Prove the refinement

(a) $x'{=}0 \Leftarrow \textbf{if } x{=}0 \textbf{ then } ok \textbf{ else } x{:=}\, x{-}1.\ x'{=}0 \textbf{ fi}$

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

where $P = x'{=}0 \ \wedge\ \textbf{if } x{\geq}0 \textbf{ then } t' = t{+}x \textbf{ else } t'{=}\infty \textbf{ fi}$

145 Let x be an integer variable and let $P = x'{=}0$. Prove the refinement

$$\begin{aligned} P \Leftarrow\ & \textbf{if } x{>}0 \textbf{ then } x{:=}\, x{-}1.\ P \\ & \textbf{else if } x{<}0 \textbf{ then } x{:=}\, x{+}1.\ P \\ & \quad \textbf{else } ok \textbf{ fi fi} \end{aligned}$$

146 Let n be a natural variable. Here is a refinement.

$$P \Leftarrow \textbf{if } n{=}0 \textbf{ then } ok \textbf{ else } n{:=}\, n{-}1.\ P.\ n{:=}\, n{+}1 \textbf{ fi}$$

(a) Ignoring time, prove this refinement where

$$P = ok$$

(b) Now add recursive time and prove this refinement where

$$P = t{:=}\, t{+}n$$

147 Let x be an integer variable, and let t be the time variable. Find the strongest implementable time specification P such that

$$P \Leftarrow \textbf{if } x{=}0 \textbf{ then } ok \textbf{ else } x{:=}\, x{+}1.\ t{:=}\, t{+}1.\ P \textbf{ fi}$$

and prove the refinement.

148 Let x be an integer variable, and let t be the time variable. Prove the refinement

(a) $x'{=}1 \Leftarrow \textbf{if } x{=}1 \textbf{ then } ok \textbf{ else } x{:=}\, div\ x\ 2.\ x'{=}1 \textbf{ fi}$

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

where $R = x'{=}1 \ \wedge\ \textbf{if } x{\geq}1 \textbf{ then } t' \leq t + log\ x \textbf{ else } t'{=}\infty \textbf{ fi}$

149 Let x be an integer variable. Is the refinement

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

a theorem when

$$P = x{<}0 \Rightarrow x'{=}1 \wedge t'{=}\infty$$

Is this reasonable? Explain.

150 Let x be an integer variable. Let P be a specification refined as follows.

$$\begin{aligned} P \Leftarrow\ & \textbf{if } x{>}0 \textbf{ then } x{:=}\, x{-}2.\ P \\ & \textbf{else if } x{<}0 \textbf{ then } x{:=}\, x{+}1.\ P \\ & \quad \textbf{else } ok \textbf{ fi fi} \end{aligned}$$

(a) Prove the refinement when $P = x'{=}0$.

(b) Add recursive time and find and prove an upper bound for the execution time.

151 Let x and y be natural variables. Here is a refinement.

$$A \Leftarrow \textbf{if } x{=}0 \vee y{=}0 \textbf{ then } ok \textbf{ else } x{:=}\, x{-}1.\ y{:=}\, y{-}1.\ A \textbf{ fi}$$

(a) Add recursive time.

(b) Find specification A that gives the exact execution time.

(c) Prove the execution time.

152 Let n be natural and let i and j be natural variables. Here are two refinements.

$A \Leftarrow i{:=}\,0.\ j{:=}\,n.\ B$

$B \Leftarrow$ **if** $i{\geq}j$ **then** ok **else** $i{:=}\,i{+}1.\ j{:=}\,j{-}1.\ B$ **fi**

(a) Add recursive time.

(b) Find specifications A and B that give good upper bounds on the time, and prove the refinements.

153 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** $i{:=}\,i{+}1$ **fi**.
if $i{=}1$ **then** ok **else** P **fi**

(b) $P \Leftarrow$ **if** *even* i **then** $i{:=}\,i/2$ **else** $i{:=}\,i{-}3$ **fi**.
if $i{=}0$ **then** ok **else** P **fi**

154 Let the variables be x, y: *int* . Write a program to refine specification $\neg ok$. Prove your refinement.

155 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

$t' \leq t + f\,i\,j \Leftarrow$ **if** $i{=}0 \wedge j{=}0$ **then** ok
else if $i{=}0$ **then** $i{:=}\,j{\times}j.\ j{:=}\,j{-}1.\ t{:=}\,t{+}1.\ t' \leq t + f\,i\,j$
else $i{:=}\,i{-}1.\ t{:=}\,t{+}1.\ t' \leq t + f\,i\,j$ **fi fi**

156 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 **fi fi**

(c) Find an upper bound for the execution time of the program in part (b).

157 (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.

158 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?

159 Let n and r be natural variables in the refinement

$P \Leftarrow$ **if** $n{=}1$ **then** $r{:=}\,0$ **else** $n{:=}\,div\ n\ 2.\ P.\ r{:=}\,r{+}1$ **fi**

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 .

160 You are given a predicate *prime* with domain *nat* such that *prime n* is $\top$ if n is a prime number, and $\bot$ if it is not. You are given natural variable n. Write a program to assign to n the smallest prime number that is bigger than or equal to the initial value of n. Write all specifications and refinements necessary to prove your program is correct, but you do not need to write the proof. You may ignore time.

161 (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) $\forall n{:}\ \Box L{\cdot}\ L'n\ =\ \Sigma\, L\,[0;..n]$

(b) $\forall n{:}\ \Box L{\cdot}\ L'n\ =\ \Sigma\, L\,[0;..n{+}1]$

162 (cube) Write a program that cubes using only addition, subtraction, and test for zero.

163 (cube test) Write a program to determine if a given natural number is a cube without using exponentiation.

164 (*mod* 2) Let n be a natural variable. The problem to reduce n modulo 2 can be solved as follows:

$n' = mod\ n\ 2 \ \Longleftarrow\ $ **if** $n{<}2$ **then** *ok* **else** $n{:=}\ n{-}2.\ n' = mod\ n\ 2$ **fi**

Using the recursive time measure, find and prove an upper time bound. Make it as small as you can.

165 (*mod* 4) Let n be a natural variable. Here is a refinement.

$n' = mod\ n\ 4 \ \Longleftarrow\ $ **if** $n{<}4$ **then** *ok* **else** $n{:=}\ n{-}4.\ n' = mod\ n\ 4$ **fi**

(a) Prove it.

(b) Insert time increments according to the recursive time measure, and write a timing specification.

(c) Prove the timing refinement.

166 Let n be a natural variable, and let b be a binary variable. Write a program to determine whether 3 is a factor of n (whether 3 divides evenly into n with no remainder), reporting the answer as the final value of b. Your program can use addition, subtraction, comparison, and binary operators, but not multiplication, division, *div*, *mod*, *floor*, or *ceil*. (Your non-program specifications can use anything.)

(a) Write a formal specification.

(b) Refine your specification to obtain a program. You do not need to prove the refinements.

167 Express formally that specification R is satisfied by any number (including 0) of repetitions of behavior satisfying specification S.

168 (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 \ \Longleftarrow\ $ **if** $n{<}2$ **then** *ok* **else** $even\ n' = even\ n.\ n' = mod\ n\ 2$ **fi**

$even\ n' = even\ n \ \Longleftarrow\ p{:=}\ 2.\ even\ p \ \Rightarrow\ even\ p' \ \wedge\ even\ n' = even\ n$

$even\ p \ \Rightarrow\ even\ p' \ \wedge\ even\ n' = even\ n \ \Longleftarrow$

$n{:=}\ n{-}p.\ p{:=}\ p{+}p.$

if $n{<}p$ **then** *ok* **else** $even\ p \ \Rightarrow\ even\ p' \ \wedge\ even\ n' = even\ n$ **fi**

(a) Prove these refinements.

(b) Using the recursive time measure, find and prove a sublinear upper time bound.

169 Let n and d be *nat* variables. Here is a refinement.

$n' = n + d\times(d-1)/2$ ⟸

if $d=0$ **then** *ok* **else** $d := d-1.\ \ n := n+d.\ \ n' = n + d\times(d-1)/2$ **fi**

(a) Prove it.

(b) Insert appropriate time increments according to the recursive measure, and write an appropriate timing specification and refinement.

(c) Prove the timing refinement.

170 Let s and n be *nat* variables. Here is a refinement.

$s' = s + 2^n - 1$ ⟸ **if** $n=0$ **then** *ok* **else** $n := n-1.\ \ s := s + 2^n.\ \ s' = s + 2^n - 1$ **fi**

(a) Prove it.

(b) Insert appropriate time increments according to the recursive measure, and write appropriate timing specifications.

(c) Prove the timing refinement.

171 (termination) Each of the following formulas attempts to say that specification S requires termination from prestate σ :

(i) $\forall\sigma' \cdot\ S \wedge t<\infty \Rightarrow t'<\infty$

(ii) $\forall\sigma' \cdot\ S \Rightarrow \exists n\text{: } nat \cdot\ t' \le t+n$

According to each formula, for what prestates does the following specification require termination? Comment on whether it is reasonable. That's $2\times3 = 6$ questions.

(a) $x\ge0 \Rightarrow x'=0 \wedge t' = t+x$ where x is an integer variable

(b) $t<\infty \Rightarrow t'<\infty$

(c) $\exists n\text{: } nat \cdot\ t' \le t+n$

172√ (maximum item) Write a program to find the maximum item in a list.

173 (list comparison) Using item comparison but not list comparison, write a program to determine whether one list comes before another in the list order.

174√ (list summation) Write a program to find the sum of a list of numbers.

175 (alternating sum) Write a program to find the alternating sum $L\,0 - L\,1 + L\,2 - L\,3 + \ldots$ of a list L of numbers.

176 Let L be a variable, L: [**int*] . Write a program that changes all the negative items of L to 0 , and otherwise leaves L unchanged.

177 (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.

178 (polynomial) You are given n: *nat* , c: *n*rat* , x: *rat* and variable y: *rat* . c is a string of coefficients of a polynomial ("of degree $n-1$") to be evaluated at x . Write a program for

$y' = \Sigma i\text{: } 0,..n \cdot\ c_i \times x^i$

179√ (binary exponentiation) Given natural variables x and y , write a program for $y' = 2^x$ without using exponentiation.

180√ (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.

181 (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]]

182 (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.

183 Write a program to find the smallest power of 2 that is bigger than or equal to a given positive integer without using exponentiation.

184 Let L be a list-of-integers state variable (program variable) L: [**int*] . Formalize the specification "sort list L ".

185 (sort test) Write a program to assign a binary variable to indicate whether a given list is sorted.

186√ (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.

187√ (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.

188 (binary search again) The problem is binary search (Exercise 187), 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.

189 (ternary search) The problem is similar to binary search (Exercise 187). 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.

190 (approximate search) Given a nonempty sorted list of numbers and a number, write a program to determine the index of an item that is closest in value to the given number.

191√ (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.

192 (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.

193 (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.

194 (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 string operators given in Section 2.2.

(b) using string indexing and string length, but no other string operators.

195 (fixed point) Let L be a nonempty sorted list of different integers. Write a program to find a fixed-point of L, that is an index i such that $L\,i = i$, or to report that no such index exists. Execution time should be at most $log\,(\#L)$.

196 (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.)

197 (all present) Given a natural number and a list of natural numbers, write a program to determine if every natural number up to the given number is an item in the list.

198 (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.

199 (duplicate)Write a program to find whether a given nonempty list has any duplicate items.

200 (item count) Write a program to find the number of occurrences of a given item in a given list.

201 (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.

202 (space-free subtext) Given a text, write a program to find the longest subtext that does not include a space character " ".

203 (merge) Given two sorted lists, write a program to merge them into one sorted list.

204 (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.

205 (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.

206 (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.

207 Define a partial order $\ll$ 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: $\square L$ such that $\neg\exists i{\cdot}\; L\, i \ll L\, j$. The execution time should be n.

208 (n sort) Given a list L such that $L\,(\square L) = \square L$, write a program to sort L in linear time and constant space. The only change permitted to L is to swap two items.

209√ (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.

210 ($n \times log\ n$ 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.

211 (reverse) Write a program to reverse the order of the items of a list.

212 (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.

213 (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 $0,..n$.

214 (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 $0,..n$.

215 (permutation inverse) You are given a list variable P of different items in $\square P$. Write a program for $P\,P' = [0;..\#P]$.

216 (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.

217 (idempotent permutation) You are given a list variable L of items in $\square L$ (not necessarily all different). Write a program to permute the list so that finally $L'\,L' = L'$.

218 (local minimum) You are given a list L of at least 3 numbers such that $L\,0 \geq L\,1$ and $L\,(\#L–2) \leq L\,(\#L–1)$. A local minimum is an interior index i: $1,..\#L–1$ such that

$$L\,(i–1) \geq L\,i \leq L\,(i+1)$$

Write a program to find a local minimum of L.

219 (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 functions *div* , *mod* , *floor* , or *ceil* .

220 (remainder) Write a program to find the remainder after natural division (Exercise 219), using only comparison, addition, and subtraction (not multiplication, division, *div*, *mod*).

221 (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.

222 (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).

223 (factor count) Write a program to find the number of distinct factors (not necessarily prime factors) of a given natural number.

224 (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.

225 (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 $L\ i$: *int* to see if an item is an integer or a list.

226 (minimum sum segment) Given a list of integers, possibly including negatives, write a program to find

(a)√ the minimum sum of any nonempty segment (sublist of consecutive items).

(b) the nonempty segment whose sum is minimum.

(c) the minimum sum of any segment, including empty segments.

227 (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 whose product is maximum.

228 (segment sum count)
(a) Write a program to find, in a given list of naturals, the number of segments (sublists of consecutive items) 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.

229 (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.

230 (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 (sublist of consecutive items) of a given list.

231 (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.

232 (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.

233 (longest balanced segment) Given a list of binary values, write a program to find a longest segment (sublist of consecutive items) having an equal number of $\top$ and $\bot$ items.

234 (longest palindrome) A palindrome is a list that equals its reverse. Write a program to find a longest palindromic segment (sublist of consecutive items) in a given list.

235 (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.)

236 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.

237 (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

$$\Sigma\, L\,[0;..i] \;=\; \Sigma\, L\,[j;..\#L]$$

238 (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.

239 Let L and M be sorted lists of numbers. Write a program to find the number of pairs of indexes i: $\square L$ and j: $\square M$ such that $L\,i \le M\,j$.

240 (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.

241 (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.

242 (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.

243 (bit sum) Write a program to find the number of ones in the binary representation of a given natural number.

244 (digit sum) Write a program to find the sum of the digits in the decimal representation of a given natural number.

245 (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.

246 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.

247 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.

248 (transitive closure) A relation R: $(0,..n)\to(0,..n)\to bin$ can be represented by a square binary array of size n. Given a relation in the form of a square binary array, write a program to find

(a) its transitive closure (the strongest transitive relation implied by the given relation).

(b) its reflexive transitive closure (the strongest reflexive and transitive relation that is implied by the given relation).

249 (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.

250 (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.

251 (inversion count) Given a list, write a program to find how many pairs of items (not necessarily consecutive) are out of order, with the larger item before the smaller item.

252 (McCarthy's 91 problem) Let i be an integer variable. Let

M = **if** i>100 **then** i:= i–10 **else** i:= 91 **fi**

(a) Prove $M \Leftarrow$ **if** i>100 **then** i:= i–10 **else** i:= i+11. M. M **fi** .

(b) Find the execution time of M as refined in part (a).

253 (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.

254 (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)
$f\ 0\ n = n+1$
$f\ 1\ 0 = 2$
$f\ 2\ 0 = 0$
$f\ (m+3)\ 0 = 1$
$f\ (m+1)\ (n+1) = f\ m\ (f\ (m+1)\ n)$

255√ (Collatz) 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×n + 1. n'=1 **fi fi**

The problem is to find the execution time. Warning: this problem has never been solved.

256√ (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 383.

257 (Fibolucci) Let a and b be integers. Then the Fibolucci numbers for a and b are

$flc\ 0 = 0$
$flc\ 1 = 1$
$flc\ (n+2) = a \times flc\ n + b \times flc\ (n+1)$

(The Fibonacci numbers are Fibolucci numbers for 1 and 1 .) Given natural k, without using any list variables, write a program to compute

Σn: 0,..k· $flc\ n \times flc\ (k–n)$

258 Let n be a natural variable. Add time according to the recursive measure, and find a finite upper bound on the execution time of

$$P \Longleftarrow \textbf{if } n \geq 2 \textbf{ then } n := n-2.\ P.\ n := n+1.\ P.\ n := n+1 \textbf{ else } ok \textbf{ fi}$$

259 (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 naturals, find the list representing their sum.

(c) Given a base and two lists representing naturals, 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 naturals, find the list representing their product.

(e) Given a base and two lists representing natural numbers, find the lists representing their quotient and remainder.

260 (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.

261 (machine division) Given two natural numbers, write a program to find their quotient using only addition, subtraction, doubling, halving, test for even, and comparisons.

262 (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.

263 Given a list of roots of a polynomial, write a program to find the list of coefficients.

264 (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.

265 (ultimately periodic sequence) You are given function f: $int \rightarrow int$ such that the sequence

$$x_0 = 0$$
$$x_{n+1} = f(x_n)$$

generated by f starting at 0 is ultimately periodic:

$$\exists p\colon nat+1 \cdot \exists n\colon nat \cdot x_n = x_{n+p}$$

The smallest positive p such that $\exists n\colon nat \cdot x_n = x_{n+p}$ 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.

266 (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 $[[1; 1; 1]; [1; 2]; [3]]$.

267 (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.

268 (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?

269 (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?

270 (greatest common divisor) Write a program to find the greatest common divisor of

(a) two positive integers.

(b) two integers (not necessarily positive ones) that are not both zero.

(c) three positive integers. 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.

271 (least common multiple) Given two positive integers, write a program to find their least common multiple.

272 (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.

273 (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.

274 (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.

275 (longest common prefix) A natural number can be written as a string of decimal digits with a single leading 0. 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.

276 (museum) You are given natural n, rationals s and f (start and finish), and lists $A, D{:}\ [n{*}rat]$ (arrive and depart) such that

$$\forall i{\cdot}\ s \le A\ i \le D\ i \le f$$

They represent a museum that opens at time s, is visited by n people with person i arriving at time $A\ i$ and departing at time $D\ i$ 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.

277 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.

278 (shift test) You are given two infinitely long lists A and B. The items can be compared for order. Both lists have period n: nat+1.

$$\forall k\text{: } nat\cdot\ A\ k = A\ (k+n)\ \wedge\ B\ k = B\ (k+n)$$

Write a program to determine if A and B are the same except for a shift of indexes.

279 (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.

280 (smallest rotation) Given a text variable, write a program to reassign it to its alphabetically (lexicographically) smallest rotation. You may use character comparisons, but not text comparisons.

281 You are given a list variable L assigned a nonempty list. All changes to L must be via procedure $swap$, defined as

$$swap\ =\ \langle i, j\text{: } \square L\cdot\ L := i \rightarrow L\ j\ |\ j \rightarrow L\ i\ |\ L\rangle$$

(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.

282 Let n and p be natural variables. Write a program to solve

$$n\geq 2\ \Rightarrow\ p'\text{: } 2^{2^{nat}}\ \wedge\ n\leq p'<n^2$$

Include a finite upper bound on the execution time, but it doesn't matter how big or small.

283 (largest true square) Write a program to find, within a two-dimensional binary array, a largest square subarray consisting entirely of items with value $\top$.

284 (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.

285 (Dutch national flag) Given a variable

$$flag\text{: } [*(red, white, blue)]$$

sort it so that all red values are first, all $white$ values are in the middle, and all $blue$ values are last. The only way allowed to change $flag$ is to use

$$swap\ =\ \langle i, j\text{: } \square flag\cdot\ flag := i \rightarrow flag\ j\ |\ j \rightarrow flag\ i\ |\ flag\rangle$$

286 (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 lists of short texts. (The long text represented is the join 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 lists of short texts.

287 (fast string searching)

(a) Given list P, find list L such that for every index n of list P, $L\,n$ 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** $C.\ L{:=}\,j{\rightarrow}i \mid L.\ j{:=}\,j{+}1.\ B$ **fi**
$C \Leftarrow$ **if** $P\,i = P\,j$ **then** $i{:=}\,i{+}1$
else if $i{=}0$ **then** ok
else $i{:=}\,L\,(i{-}1).\ C$ **fi fi**

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 **fi**
$F \Leftarrow$ **if** $n{=}\#S$ **then** $h{:=}\,\infty$
else if $P\,m = S\,n$ **then** $m{:=}\,m{+}1.\ n{:=}\,n{+}1.\ E$
else G **fi fi**
$G \Leftarrow$ **if** $m{=}0$ **then** $n{:=}\,n{+}1.\ F$ **else** $m{:=}\,L\,(m{-}1).\ G$ **fi**

Find specifications D, E, F, and G so that D is the problem and the four refinements are theorems.

288 (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.

289 Let x be a *nat* variable. In the refinement

$P \Leftarrow$ **if** $x{=}1$ **then** ok **else** $x{:=}\,div\ x\ 2.\ P.\ x{:=}\,x{\times}2$ **fi**

each call pushes a return address onto a stack, and each return pops an address from the stack. Add a space variable s and a maximum space variable m, with appropriate assignments to them in the program. Find and prove an upper bound on the maximum space used.

290 (factorial space) We can compute $x{:=}\,n!$ (factorial) as follows.

$x{:=}\,n! \Leftarrow$ **if** $n{=}0$ **then** $x{:=}\,1$ **else** $n{:=}\,n{-}1.\ x{:=}\,n!.\ n{:=}\,n{+}1.\ x{:=}\,x{\times}n$ **fi**

Each call $x{:=}\,n!$ pushes a return address onto a stack, and each return pops an address from the stack. Add a space variable s and a maximum space variable m, with appropriate assignments to them in the program. Find and prove an upper bound on the maximum space used.

291 Let k be a natural constant, and let x and n be natural variables. Suppose one unit of space is allocated before each recursive call (for the return address), and freed after the call. Find and prove a maximum space bound for the refinement

$P \Leftarrow$ **if** $n{=}0$ **then** $x{:=}\,0$ **else** $n{:=}\,n{-}1.\ P.\ x{:=}\,x{+}k$ **fi**

292 (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.

293 (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, proceeding 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 which disk is 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.

294 Let P and Q be specifications. Let A be an assertion and let A' be the same as A but with primes on all the variables. Prove the assertion law

$$P.\,Q \Longleftarrow P \wedge A'.\ A \Rightarrow Q$$

295 Which three of the six assertion laws (Subsection 11.3.12) can be turned around, switching the problem and the solution?

296 Let S be a specification. Let A be an assertion and let A' be the same as A but with primes on all the variables. How does the exact precondition for A' to be refined by S differ from $(S.\ A)$? Hint: consider prestates in which S is unsatisfiable, then deterministic, then nondeterministic.

297 Let x, y, and z be real variables. In what circumstances is the execution of

$$x := y+z.\ \ z := x+y$$

guaranteed to result in $z' = x+y$?

298√ 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).

299 Prove

(a) the Precondition Law: Assertion A is a sufficient precondition for specification P to be refined by specification S if and only if $A \Rightarrow P$ is refined by S.

(b) the Postcondition Law: Assertion A' is a sufficient postcondition for specification P to be refined by specification S if and only if $A' \Rightarrow P$ is refined by S.

300 Let all variables be integer except L is a list of integers. What is the exact precondition
(a) for $x'+y' > 8$ to be refined by $x := 1$
(b) for $x'=1$ to be refined by $x := 1$
(c) for $x'=2$ to be refined by $x := 1$
(d) for $x'=y$ to be refined by $y := 1$
(e) for $x' \geq y'$ to be refined by $x := y+z$
(f) for $y'+z' \geq 0$ to be refined by $x := y+z$
(g) for $x' \leq 1 \vee x' \geq 5$ to be refined by $x := x+1$
(h) for $x' < y' \wedge \exists x \cdot L\, x < y'$ to be refined by $x := 1$
(i) for $\exists y \cdot L\, y < x'$ to be refined by $x := y+1$
(j) for $L'\, 3 = 4$ to be refined by $L := i \rightarrow 4 \mid L$
(k) for $x'=a$ to be refined by **if** $a > b$ **then** $x := a$ **else** ok **fi**
(l) for $x'=y \wedge y'=x$ to be refined by $(z := x.\ x := y.\ y := z)$
(m) for $a \times x'^{\,2} + b \times x' + c = 0$ to be refined by $(x := a \times x + b.\ x := -x/a)$
(n) for $f' = n'!$ to be refined by $(n := n+1.\ f := f \times n)$ where n is natural and $!$ is factorial.
(o) for $7 \leq c' < 28 \wedge odd\ c'$ to be refined by $(a := b-1.\ b := a+3.\ c := a+b)$
(p) for $s' = \Sigma\, L\, [0;..i']$ to be refined by $(s := s + L\, i.\ i := i+1)$
(q) for $x' > 5$ to be refined by $x' : x+(1,2)$
(r) for $x' > 0$ to be refined by $x' : x+(-1, 1)$

301 For what exact precondition and exact postcondition does the following assignment move integer variable x farther from zero?
(a) $x := x+1$
(b) $x := abs\ (x+1)$
(c)√ $x := x^2$

302 For what exact precondition and exact 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) $x := x^2$

303 Let x be an integer state variable, and there are no other state variables.
(a) For what exact precondition does $x := x^2$ make x be even?
(b) What does it mean to say that your answer to part (a) is the exact precondition for $x := x^2$ to make x be even?
(c) For what exact postcondition does $x := x^2$ make x be even?
(d) What does it mean to say that your answer to part (c) is the exact postcondition for $x := x^2$ to make x be even?

304 Prove
(a) $\top$ is an invariant for specification S.
(b) $\bot$ is an invariant for specification S.
(c) Assertion A is an invariant for ok.
(d) $x+y = 5$ is an invariant for $(x := x+1.\ y := y-1)$ where the variables are x and y.
(e) $x \geq 0$ is an invariant for $x := x+1$ where the variable is x.
(f)√ $y=x^2$ is an invariant for $(x := x+1.\ y := y + 2 \times x - 1)$ where the variables are x and y.
(g) $a \times x^2 + b \times x + c = 0$ is an invariant for $(x := a \times x + b.\ x := -x/a)$ where the variable is x.
(h) $f = n!$ is an invariant for $(n := n+1.\ f := f \times n)$ where f and n are natural variables and $!$ is factorial.

305 (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) $Q.S$

End of Program Theory

10.5 Programming Language

306 (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 binary notation for this form of assignment. Show what happens to the Substitution Law.

307 Suppose variable declaration is defined as

new 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

new x: *int*$\cdot$ *ok*

308 Suppose variable declaration with initialization is defined as

new x: $T := e\cdot\ P \quad = \quad$ **new** x: $T\cdot\ x := e.\ P$

In what way does this differ from the definition given in Subsection 5.0.0?

309 Here are two different definitions of variable declaration with initialization.

new x: $T := e\cdot\ P \quad = \quad \exists x, x'$: $T\cdot\ x{=}e \wedge P$

new x: $T := e\cdot\ P \quad = \quad \exists x'$: $T\cdot$ (substitute e for x in P)

Show how they differ with an example.

310 What is wrong with defining local variable declaration as follows:

new x: $T\cdot P \quad = \quad \forall x$: $T\cdot\ \exists x'$: $T\cdot P$

311 The specification

new 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.)

312 (frame problem) Suppose there is one nonlocal variable x, and we define $P \ = \ x'{=}0$. Can we prove

$P \ \Longleftarrow$ **new** 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 sequential composition. Is it being used properly?

313 Let the state variables be x, y, and z. Rewrite **frame** $x\cdot\ \top$ without using **frame**. Say in words what the final value of x is.

314 Let x, y, and n be natural variables. Let f: $nat \to nat$ be a function. Simplify

frame $x\cdot$ **new** y, m: *nat*$\cdot\ \ m := n.\ \ x' = f\,m \ \wedge\ y' = f\,(m{+}1)$

315 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).

316 In a language with array element assignment, what is the exact precondition for $A'i' = 1$ to be refined by $(A(A\ i) := 0.\ \ A\ i := 1.\ \ i := 2)$?

317 Let A be a nonempty array-of-naturals variable

A: $[(nat+1)^* nat]$

and there are no other variables.

(a) Express $A(A(A\ 0)) := A\ 0$ without using $:=$.

(b) Suppose initially $A = [2; 0; 1]$. What is the final value after execution of the array element assignment in part (a)?

318 Let n be a natural constant, and let f and i be natural state variables. Define

$n! \ = \ \Pi i: 0,..n \cdot\ i+1 \ = \ 1 \times 2 \times 3 \times ... \times n$

Prove

$f' = n! \ \Leftarrow \ f := 1.\ \ i := 0.\ \ \mathbf{while}\ i<n\ \mathbf{do}\ i := i+1.\ \ f := f \times i\ \mathbf{od}$

319 Let p and n be natural variables. Prove

$p' = 2^{20} \ \Leftarrow \ p := 1.\ \ n := 0.\ \ \mathbf{while}\ n \neq 20\ \mathbf{do}\ p := p \times 2.\ \ n := n+1\ \mathbf{od}$

320√ (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 $x := x - 1.$ **new** n: $nat \cdot\ y := n$ **fi**
od

321 Let $W \Leftarrow$ **while** b **do** P **od** be an alternate notation for $W \Leftarrow$ **if** b **then** $P.\ W$ **else** ok **fi**.
Let $R \Leftarrow$ **do** P **until** b **od** be an alternate notation for $R \Leftarrow P.$ **if** b **then** ok **else** R **fi**.
Now prove

$(R \Leftarrow$ **do** P **until** b **od**$) \wedge (W \Leftarrow$ **while** $\neg b$ **do** P **od**$)$
$\Leftarrow (R \Leftarrow P.\ W) \wedge (W \Leftarrow$ **if** b **then** ok **else** R **fi**$)$

322 The notation **do** P **while** b **od** has been used as a loop construct that is executed as follows. First, P is executed; then b is evaluated, and if its value is $\top$ then execution is repeated, and if its value is $\bot$ then execution is finished.

(a) Let x be an integer variable. Prove

$mod\ x'\ 2 \ = \ mod\ x\ 2 \ \Leftarrow$ **do** $x := x - 2$ **while** $x \geq 2$ **od**

(b) Let m and n be integer variables. Prove

$m := m+n-10.\ \ n := 10 \ \Leftarrow$ **do** $m := m-1.\ \ n := n+1$ **while** $n \neq 10$ **od**

(c) In parts (a) and (b), add a time variable, and charge time 1 for each loop iteration. Notice that for this loop, recursive time is not quite the same as charging time 1 for each iteration. Choose a time specification, and prove it.

323 Let P: $nat \to bin$.

(a) Define quantifier $\Downarrow$ so that $\Downarrow m$: $nat \cdot P\ m$ is the smallest natural m such that $P\ m$, and ∞ if there is none.

(b) Prove n:= $\Downarrow m$: $nat \cdot P\ m \Leftarrow n$:= 0. **while** $\neg\ P\ n$ **do** n:= n+1 **od** .

324 Given natural list variable L , index variable i , and time variable t , increase each list item by 1 until you have created item 100 . The time is bounded by $\#L$. The program is

```
i:= 0.
do exit when i=#L.
   L i := L i + 1.
   exit when L i = 100.
   i:= i+1
od
```

Write a formal specification, and prove it is refined by the program.

325 Here is a nest of loops. All **exit**s are shown. What refinements need to be proved in order to prove that this nest of loops refines specification S ?

(a)√

```
do A.
   do B.
      exit 2 when c.
      D. od.
   E od
```

(b)√

```
do A.
   exit 1 when b.
   C.
   do D.
      exit 2 when e.
      F.
      exit 1 when g.
      H od.
   I od
```

(c)

```
do A.
   do B.
      do C.
         exit 1 when u.
         exit 2 when v.
         exit 3 when w.
         D od.
      E od.
   F od
```

(d)

```
do A.
   do B.
      exit 2 when u.
      C.
      exit 1 when v.
      D od.
   E.
   exit 1 when w.
   F.
   do G.
      do H.
         exit 1 when x.
         I.
         exit 2 when y.
         J od.
      K.
      exit 1 when z.
      L od.
   M od
```

(e)

```
do A.
   exit 1 when u.
   do B.
      exit 2 when v.
      exit 1 when ⊤ od od
```

326√ Using a **for**-loop, write a program to add 1 to every item of a list.

327 (square) Let n be natural and let s be a natural variable. Using a **for**-loop, without using multiplication or exponentiation, write a program for $s'=n^2$.

328 Let L be a variable, L: [*int] . Here is a program to change all the negative items of L to 0 , and otherwise leave L unchanged.

for n:= 0;..#L **do if** $L\,n < 0$ **then** L:= $n \rightarrow 0 \mid L$ **else** ok **fi od**

Write all the specifications and refinements needed to prove that execution of this program does as intended. You do not need to prove the refinements.

329 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 **od** = ok

for i:= j **do** P **od** = (substitute j for i in P)

for i:= (n;..k); (k;..m) **do** P **od** = **for** i:= n;..k **do** P **od**. **for** i:= k;..m **do** P **od**

(a) From this definition, what can we prove about **for** i:= 0;..n **do** n:= n+1 **od** where n is an integer variable?

(b) What kinds of **for**-loop are in the programming languages you know?

330 (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)
```
new e: nat := 0·
for i:= 0;..#L
do  if m = L i then e:= e+1
    else  if i = 2×e then m:= L i.  e:= e+1
          else ok fi fi od
```

(b)
```
new s: nat := 0·
for i:= 0;..#L
do  if m = L i then ok
    else  if i = 2×s then m:= L i
          else s:= s+1 fi fi od
```

331 Let a and b be binary expressions with unprimed variables, and let A , B , C , P , Q , R , S , T , and U be implementable specifications such that the refinements

A ⟸ **if** a **then** ok **else if** b **then** P. B **else** Q. C **fi fi**

B ⟸ **if** a **then** ok **else if** b **then** R. C **else** S. A **fi fi**

C ⟸ **if** a **then** ok **else if** b **then** T. A **else** U. B **fi fi**

are all theorems. Then A can be executed as follows (using ⁸ for labeling):

A⁸ **if** a **then go to** D **else if** b **then** P. **go to** B **else** Q. **go to** C **fi fi**.

B⁸ **if** a **then go to** D **else if** b **then** R. **go to** C **else** S. **go to** A **fi fi**.

C⁸ **if** a **then go to** D **else if** b **then** T. **go to** A **else** U. **go to** B **fi fi**.

D⁸ ok

We have replaced refinement and call with labeling and **go to**s.

(a) Show that it is not possible to replace refinement and call (in this example) with **while** loops without introducing any new variables.

(b) Show that it is possible to replace refinement and call (in this example) with **while** loops if you introduce new variables.

332 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).

333 We defined **wait until** w $=$ $t:= t\uparrow w$ where t is an extended natural time variable, and w is an extended natural expression.

(a)√ Prove **wait until** w $\Leftarrow$ **if** $t\geq w$ **then** ok **else** $t:= t+1$. **wait until** w **fi**

(b) Now suppose that t is a nonnegative extended real time variable, and w is a nonnegative extended real expression. Redefine **wait until** w appropriately, and refine it using the real time measure (assume any positive operation time you need).

334 Could we define the expression P **value** e with the axiom

(a) $x'=(P \textbf{ value } e) = P.\ x'=e$

(b) $x'=(P \textbf{ value } e) \Longrightarrow P.\ x'=e$

(c) $P \Rightarrow (P \textbf{ value } e)=e'$

(d) $x'=(P \textbf{ value } e) \wedge P \Rightarrow x'=e'$

335 Let a and b be rational variables. Define procedure P as

$P = \langle x, y: rat\cdot \textbf{ if } x=0 \textbf{ then } a:= x \textbf{ else } a:= x\times y.\ a:= a\times y \textbf{ fi}\rangle$

(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.

336 "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 must be a variable. Define "call-by-value-result" formally. Discuss its merits and demerits.

337 (call-by-name) Here is a procedure applied to an argument.

$\langle x: int\cdot a:= x.\ b:= x\rangle\ (a+1)$

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".)

338 (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 binary 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 as a specification using the notations of this book.

(b) Refine this specification as a program using the notations of this book.

339 (Boole's binaries) If $\top=1$ and $\bot=0$, express

(a) $\neg a$

(b) $a\wedge b$

(c) $a\vee b$

(d) $a\Longrightarrow b$

(e) $a\Longleftarrow b$

(f) $a=b$

(g) $a\neq b$

using only the following symbols (in any quantity)

(i) 0 1 a b ( ) + − ×

(ii) 0 1 a b ( ) − ↑ ↓

That's $7\times 2 = 14$ questions.

340 Let n be a number, and let P, Q, and R be probabilistic specifications. Prove
(a) $n{\times}P.\,Q \;=\; n{\times}(P.\,Q) \;=\; P.\,n{\times}Q$
(b) $P{+}Q.\,R \;=\; (P.\,R) + (Q.\,R)$
(c) $P.\,Q{+}R \;=\; (P.\,Q) + (P.\,R)$
(d) $x{:=}\,e.\;P \;=\; \langle x{\cdot}\;P\rangle e$

341 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.

342 (two children) I have two children. At least one of them is a girl. What is the probability that the other one is also a girl?

343 (three cards) There are three cards. One is red on both sides; one is white on both sides; one is red on one side and white on the other side. You are looking at one side of one card, and it is red. What is the probability that its other side is also red?

344 (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.

345 (drunk) A drunkard is trying to walk home. At each time unit, the drunkard may go forward one distance unit, or 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.

346 (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.

347 (one 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

$$R \;\Longleftarrow\; t{:=}\,t{+}1.\;\textbf{if}\;\mathit{rand}\;2\;\textbf{then}\;\mathit{ok}\;\textbf{else}\;R\;\textbf{fi}$$

348 A coin is flipped repeatedly. At each head, natural variable x is decreased by 1; at each tail, x is left unchanged. How many flips are there until $x{=}0$?

349 (two coins) We flip a pair of coins repeatedly. We ignore the flip when both coins show tail, and we count the flip when at least one coin shows head. We continue flipping until we have counted 100 flips. Of these 100 flips, what fraction show two heads?

350 (flipping switch) A two-position switch is flipped some number of times. At each time (including initially, before the first flip) there is probability 1/2 of continuing to flip, and probability 1/2 of stopping. The probability that the switch ends in its initial state is 2/3 , and the probability that it ends flipped is 1/3 .

(a) Express the final state as a probability distribution.

(b) Equate the distribution with a program describing the flips.

(c) Prove the equation.

351 (dice) If you repeatedly throw a pair of six-sided dice until they are equal,

(a)√ prove the distribution of final times is $(t' \geq t) \times (5/6)^{t'-t} \times 1/6$.

(b) prove the average final time is $t+5$.

352 (building 1/2) Suppose we can flip a coin, but we suspect that the coin may be biased. Let us say that the probability of landing on its head is p . What we want is a coin with probability 1/2 of landing on its head. Here's one way to create what we want. Flip the coin twice. If the outcomes differ, use the first outcome. If the outcomes are the same, repeat the experiment, until the two outcomes differ, and then use the first outcome of the first pair that differed. Prove that this procedure works, and find out how long it takes.

353 (disease) The incidence of a certain disease is one person in a thousand. There is a test for the disease that is 99% accurate. What is the probability that the person has the disease if they test

(a) positive (the test says they have the disease)?

(b) negative (the test says they don't have the disease)?

354 (Monty Hall) Monty Hall is a game show host, and in this game there are three doors. A prize is hidden behind one of the doors. The contestant chooses a door. Monty then opens one of the doors, but not the door with the prize behind it, and not the door the contestant has chosen. Monty asks the contestant whether they (the contestant) would like to change their choice of door, or stay with their original choice. What should the contestant do?

355 (biased Monty Hall) There are three doors numbered 0 , 1 , and 2 . Monty rolls a 6-sided die, and if it shows 1 , 2 , or 3 Monty places the prize behind door 0 ; if it shows 4 or 5 Monty places the prize behind door 1 , and if it shows 6 Monty places the prize behind door 2 . The contestant chooses a door. Monty then opens one of the doors, but not the door with the prize behind it, and not the door the contestant has chosen. Monty asks the contestant whether they (the contestant) would like to change their choice of door, or stay with their original choice. What should the contestant do?

356 (Bob asks Alice for a date) Bob wants to ask Alice out on a date. He knows his success rate (the fraction of all the women he's asked out so far who have said yes). And he knows Alice's agreement rate (the fraction of all the guys who have asked Alice out so far that she has said yes to). What is the probability that Alice will say yes to Bob if

(a) Bob's success rate is 2/3 and Alice's agreement rate is 1/2 ?

(b) Bob's success rate is 2/3 and Alice's agreement rate is 2/3 ?

357 (chameleon) There are c chameleons, of which r are red and the remainder are blue. At each tick of the clock, a chameleon is chosen at random, and

(a) it changes color. How long will it be before all chameleons have the same color?

(b) one of the chameleons of the other color changes color. How long will it be before all chameleons have the same color?

358 (amazing average) Consider the following innocent-looking program, where p is a positive natural variable.

$$loop \;=\; \textbf{if } rand\ 2 \textbf{ then } p := 2{\times}p.\ \ t := t{+}1.\ \ loop \textbf{ else } ok \textbf{ fi}$$

We repeatedly flip a coin; each time we see a head, we double p, stopping the first time we see a tail.

(a) What is the $loop$ distribution?

(b) What is the average final value of t?

(c) What is the average final value of p?

359 (dice room) There are infinitely many people, and a madman. The madman kidnaps a person and puts the person in a room. The madman then throws a pair of six-sided dice. If the dice land as a pair of 1s, the madman murders the person. If the dice do not land as a pair of 1s, the madman releases the person, then kidnaps 10 new people. Again the madman throws the dice. If the dice land as a pair of 1s, the madman murders everyone in the room. If the dice do not land as a pair of 1s, the madman releases everyone, then kidnaps 100 new people. The madman keeps doing this, increasing the number of kidnapped people by 10 times each round, until the dice land as a pair of 1s, at which point he murders everyone in the room, and is finished.

(a) Each round, what is the probability that the people in the room are murdered?

(b) What is the average number of rounds?

(c) What is the average number of people kidnapped?

(d) What is the average number of people murdered?

(e) What is the average fraction of people kidnapped who are murdered?

360 (Newcomb's paradox) There are two boxes, one transparent and one opaque. Inside the transparent box there is a visible \$1,000. The player can choose to take the contents of both boxes, or just the opaque box. The amount in the opaque box was determined by a predictor, who predicted, with probability p of being correct, whether the player will take both boxes or just the opaque box. If the predictor predicted that the player will take both boxes, then the opaque box contains nothing. If the predictor predicted that the player will take just the opaque box, then the opaque box contains \$1,000,000. The player knows, when making the choice, what the two possible amounts are, and that the amount was determined by the predictor with probability p of being correct. What should the player do?

361 (Tokieda's paradox)

(a) Alice tosses a coin until she sees a head followed by a tail. How many times does she toss a coin on average?

(b) Bob tosses a coin until he sees two heads in a row. How many times does he toss a coin on average?

(c) Since the probability of a head is equal to the probability of a tail, the probability of a head followed by a tail is equal to the probability of two heads in a row; each is $1/4$. So why do the answers to (a) and (b) differ?

362 (conditional probability) Bayes defined conditional probability, using his own notation, as follows:

$$P\,(A \mid B) \;=\; \frac{P\,(A \wedge B)}{P\,(B)}$$

This is to be read "the probability that A is true given that B is true is equal to the probability that both are true divided by the probability that B is true". How is conditional probability expressed in our notation?

363 (standard deviation) Let P be any distribution of final states (primed variables), and let e be any number expression or binary expression over initial states (unprimed variables). The standard deviation of e according to distribution P is

$$((P.\,e^2) - (P.\,e)^2)^{1/2}$$

Standard deviation measures the margin of error.

(a) Prove standard deviation is equivalent to

$$(P.\,(e - (P.\,e))^2)^{1/2}$$

(b) Let n vary over $nat{+}1$. What is the standard deviation of 2 according to distribution $2^{-n'}$?

(c) Let n vary over $nat{+}1$. What is the standard deviation of 2^{-n} according to distribution $2^{-n'}$?

364 What features of programming languages interfere with the use of programming theory? In what way do they interfere?

365 We propose to define a new programming connective $P \blacklozenge Q$. What properties of $\blacklozenge$ are essential? Why?

End of Programming Language

10.6 Recursive Definition

366 Prove $\neg\, {-1}\colon nat$. Hint: You will need induction.

367 Prove $\forall n\colon nat\cdot P\,n \;=\; \forall n\colon nat\cdot \forall m\colon 0\,..\,n\cdot P\,m$.

368√ Prove that the square of an odd natural number is $8{\times}m + 1$ for some natural m.

369 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.

370 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, 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 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.

371 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 of *nat*?

372 Subsection 6.0.0 gives four predicate versions of *nat* induction. Prove that they are equivalent.

373 Prove the law $nat = 0,..\infty$ from the other laws.

374 Define the even integers $int{\times}2$ by construction and induction, without using multiplication or division.

375 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

$bad \ = \ \S n: nat\cdot \neg\ n: bad$

376 Prove the following; quantifications are over *nat* unless stated otherwise.

(a) $\neg\exists i, j\cdot\ j{\neq}0 \ \wedge\ 2^{1/2} = i/j$ The square roots of 2 are 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) $\forall n\cdot (\Sigma i: 0,..n\cdot i^2) \ = \ n{\times}(n{-}1){\times}(2{\times}n - 1)/6$

(e) $\forall n\cdot (\Sigma i: 0,..n\cdot i^3) \ = \ (\Sigma i: 0,..n\cdot i)^2$

(f) $\forall n\cdot (\Sigma i: 0,..n\cdot 2^i) \ = \ 2^n - 1$

(g) $\forall n\cdot (\Sigma i: 0,..n\cdot i{\times}2^i) \ = \ (n{-}2){\times}2^n + 2$

(h) $\forall n\cdot (\Sigma i: 0,..n\cdot (-2)^i) \ = \ (1 - (-2)^n) \,/\, 3$

(i) $\forall n\cdot n{\geq}3 \implies 2{\times}n^3 > 3{\times}n{\times}(n{+}1)$

(j) $\forall n\cdot n{\geq}4 \implies 3^n > n^3$

(k) $\forall n\cdot n{\geq}4 \implies n! > 2^n$ where ! is factorial

(l) $\forall n\cdot n{\geq}10 \implies 2^n > n^3$

(m) $\forall a, d\cdot \exists q, r\cdot d{\neq}0 \implies r{<}d \ \wedge\ a = q{\times}d + r$

(n) $\forall a, b\cdot a{\leq}b \implies (\Sigma i: a,..b\cdot 3^i) = (3^b{-}3^a)/2$

(o) $\forall n\cdot \ (n{+}1)^{nat} : \ nat{\times}n + 1$

(p) $\forall n\cdot (\Sigma i: 0,..n\cdot i{\times}(i{+}1)) \ = \ (n{-}1){\times}n{\times}(n{+}1)/3$

(q) $\forall n\cdot (\Sigma i: 0,..n\cdot (-1)^i \times i^2) \ = \ -\,(-1)^n \times (n{-}1) \times n \,/\, 2$

(r) $\forall n\cdot (\Sigma i: 0,..n\cdot 1/((i{+}1){\times}(i{+}2))) \ = \ n/(n{+}1)$

377 Show that we can define *nat* by fixed-point construction together with

(a) $\forall n: nat\cdot \ 0 \leq n < n{+}1$

(b) $\exists m: nat\cdot \forall n: nat\cdot \ m \leq n < n{+}1$

378 Let *R* be a relation of naturals $R: nat{\rightarrow}nat{\rightarrow}bin$ that is monotonic in its second parameter

$\forall i, j\cdot R\ i\ j \Rightarrow R\ i\ (j{+}1)$

Prove

$\exists i\cdot \forall j\cdot R\ i\ j \ = \ \forall j\cdot \exists i\cdot R\ i\ j$.

379√ 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.

380 (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.

381 (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?

382 Function f is called monotonic if $\forall i, j : \square f \cdot\ i \le j \Rightarrow f\,i \le f\,j$.

(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 $\forall n \cdot f\,n \le n$.

383 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) $fib\ (2{\times}n{+}2) = 2 \times fib\ n \times fib\ (n{+}1) + (fib\ (n{+}1))^2$

(g) $fib\ (k{\times}n) : nat \times fib\ n$

384 What elements can be proved in P from the axiom $P = 1, x, -P, P{+}P, P{\times}P$? Prove

$2{\times}x^2{-}1 : P$

385 Express 2^{int} without using exponentiation. You may introduce auxiliary names.

386 What is the smallest bunch satisfying

(a) $B = 0, 2{\times}B + 1$

(b) $B = 2, B{\times}B$

387 Bunch *this* is defined by the construction and induction axioms

$2, 2{\times}this\colon\ this$

$2, 2{\times}B\colon\ B \Rightarrow this\colon B$

Bunch *that* is defined by the construction and induction axioms

$2, that{\times}that\colon\ that$

$2, B{\times}B\colon\ B \Rightarrow that\colon B$

Prove $this = that$.

388 For each of the following construction axioms, use recursive construction to find the first few elements, and then guess what *what* is.

(a) $1, what{+}what, what{\times}what\colon what$

(b) $2, what{+}what, what{\times}what\colon what$

389 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 in English, and formally (no proof needed).

(b) What is ply_∞ ?

(c) Is ply_∞ a solution? If so, is it the only solution?

390 Let $A \dot{-} B$ be bunch removal: from bunch A remove any elements in bunch B . The operator $\dot{-}$ has precedence level 4, and is defined by the axiom

$x\colon A \dot{-} B \ =\!=\ x\colon A \wedge \neg\, x\colon B$

For each of the following fixed-point equations, what does recursive construction yield? Does it satisfy the fixed-point equation?

(a) $Q = nat \dot{-} (Q{+}3)$

(b) $D = 0, (D{+}1) \dot{-} (D{-}1)$

(c) $E = nat \dot{-} (E{+}1)$

(d) $F = 0, (nat \dot{-} F){+}1$

391 For each of the following fixed-point equations, what does recursive construction yield? Does it satisfy the fixed-point equation?

(a) $P = \S n\colon nat\cdot\ n{=}0 \wedge P{=}null \ \vee\ n\colon P{+}1$

(b) $Q = \S x\colon xnat\cdot\ x{=}0 \wedge Q{=}null \ \vee\ x\colon Q{+}1$

392 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?

393(a) Considering E as the unknown, find three solutions of $E, E{+}1 = nat$.

(b) Now add the induction axiom $B, B{+}1 = nat \Rightarrow E\colon B$. What is E ?

394 Let n be a natural number. From the construction axiom $0, n{-}few\colon few$

(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.

395 Let *truer* be a bunch of strings of binary values defined by the construction and induction axioms

nil , ⊥;*truer* , *truer*;⊤ : *truer*

nil , ⊥;*B* , *B*;⊤ : *truer* ⇒ *truer*: *B*

Given a string of binary values, write a program to determine if the string is in *truer* .

396 Investigate the fixed-point equation

strange = §*n*: *nat*· ∀*m*: *strange*· ¬ *m*+1: *n*×*nat*

397 (strings) If *S* is a bunch of strings, then **S* is the bunch of all strings formed by joining together any number of any strings in *S* in any order.

(a) Define **S* by construction and induction.

(b) Prove ***S* = **S* (don't just use this law).

398 (pure sets) A pure set is a set all of whose elements are pure sets. For example

{{*null*}, {{*null*}}}

First, {*null*} is a pure set because all zero of its elements are pure sets. So {{*null*}} is a pure set because its one element {*null*} is a pure set. So {{*null*}, {{*null*}}} is a pure set because both its elements are pure sets. All of mathematics can be implemented as pure sets. Define the bunch of texts representing pure sets.

399 (Backus-Naur Form) Backus-Naur Form is a grammatical formalism in which grammatical rules are written as in the following example.

⟨*exp*⟩::= ⟨*exp*⟩ + ⟨*exp*⟩ | ⟨*exp*⟩ × ⟨*exp*⟩ | 0 | 1

In our formalism, it would be written

exp = *exp*; "+"; *exp*, *exp*; "×"; *exp*, "0", "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.

400 Define language *lang* by the fixed-point construction axiom

lang = *nil*, "("; *lang*; ")", *lang*;*lang*

and associated fixed-point induction axiom.

(a) Informally, what is the language described?

(b) Write an equivalent, nonrecursive definition of the language. Hint: start with § and use a predicate that counts occurrences of characters in a text.

401 (bit strings) Let *a*, *b*, *c*: *(0, 1) . Define operator ⊕ (precedence 4) as follows.

nil ⊕ *nil* = 0

a ⊕ 0 = *a*

(*a*; 0) ⊕ 1 = *a*; 1

(*a*; 1) ⊕ 1 = *a* ⊕ 1; 0

a ⊕ *b* = *b* ⊕ *a*

(*a* ⊕ *b*) ⊕ *c* = *a* ⊕ (*b* ⊕ *c*)

Prove

(a) *a* ⊕ *a* = *a*; 0

(b) *a*; 0 = *a* ⊕ *a* ⊕ 0

(c) *a*; 1 = *a* ⊕ *a* ⊕ 1

402 (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.

403 Section 6.1 defines program *zap* by the fixed-point equation

$$zap \;=\; \textbf{if } x{=}0 \textbf{ then } y{:=}\,0 \textbf{ else } x{:=}\,x{-}1.\;\; t{:=}\,t{+}1.\;\; zap \textbf{ fi}$$

(a)√ What axiom is needed to make *zap* the weakest fixed-point?
(b) What axiom is needed to make *zap* the strongest fixed-point?
(c) Section 6.1 gives six solutions to this equation. Find more solutions. Hint: strange things can happen at time ∞.

404 Let n be a natural variable. You are given the refinement

$$P \;\Longleftarrow\; \textbf{if } n{=}0 \textbf{ then } n'{>}0.\;\; P \textbf{ else } ok \textbf{ fi}$$

Using recursive construction, find P. You may ignore time.

405 Let x and y be rational variables. Define program *zot* by the fixed-point equation

$$zot \;=\; \textbf{if } x{=}y \textbf{ then } y{:=}\,0 \textbf{ else } x{:=}\,(x{+}y)/2.\;\; zot \textbf{ fi}$$

(a) Add recursive time.
(b) Give two solutions to this equation (with recursive time added) (considering *zot* as the unknown). (No proof needed.)
(c) The definition of *zot* makes it a solution (fixed-point) of an equation. What axiom is needed to make *zot* the weakest solution (weakest fixed-point)?

406 Let i be an integer variable, and let P be a specification such that

$$P \;=\; i{:=}\,i{-}1.\;\; \textbf{if } i{=}0 \textbf{ then } i{:=}\,3 \textbf{ else } P.\;\; i{:=}\,3 \textbf{ fi}$$

(a) Add recursive time.
(b) Is $i'{=}3 \wedge (0{<}i \Rightarrow t'{=}t{+}i{-}1)$ a fixed-point (solution for P)? Prove or disprove.

407 Let all variables be integer. Add recursive time. Using recursive construction, find a fixed-point of

(a) $skip \;=\; \textbf{if } i{\geq}0 \textbf{ then } i{:=}\,i{-}1.\;\; skip.\;\; i{:=}\,i{+}1 \textbf{ else } ok \textbf{ fi}$
(b) $inc \;=\; ok \vee (i{:=}\,i{+}1.\;\; inc)$
(c) $sqr \;=\; \textbf{if } i{=}0 \textbf{ then } ok \textbf{ else } s{:=}\,s + 2{\times}i - 1.\;\; i{:=}\,i{-}1.\;\; sqr \textbf{ fi}$
(d) $fac \;=\; \textbf{if } i{=}0 \textbf{ then } f{:=}\,1 \textbf{ else } i{:=}\,i{-}1.\;\; fac.\;\; i{:=}\,i{+}1.\;\; f{:=}\,f{\times}i \textbf{ fi}$
(e) $chs \;=\; \textbf{if } a{=}b \textbf{ then } c{:=}\,1 \textbf{ else } a{:=}\,a{-}1.\;\; chs.\;\; a{:=}\,a{+}1.\;\; c{:=}\,c{\times}a/(a{-}b) \textbf{ fi}$
(f) $foo \;=\; \textbf{if } i{=}0 \textbf{ then } i{:=}\,3 \textbf{ else } foo \textbf{ fi}$
(g) $bar \;=\; i{:=}\,i{-}1.\;\; \textbf{if } i{=}0 \textbf{ then } i{:=}\,3 \textbf{ else } bar.\;\; i{:=}\,3 \textbf{ fi}$

408 Let all variables be integer. Add recursive time. Any way you can, find a fixed-point of

(a) $walk \;=\; \textbf{if } i{\geq}0 \textbf{ then } i{:=}\,i{-}2.\;\; walk.\;\; i{:=}\,i{+}1.\;\; walk.\;\; i{:=}\,i{+}1 \textbf{ else } ok \textbf{ fi}$
(b) $crawl \;=\; \textbf{if } i{\geq}0 \textbf{ then } i{:=}\,i{-}1.\;\; crawl.\;\; i{:=}\,i{+}2.\;\; crawl.\;\; i{:=}\,i{-}1 \textbf{ else } ok \textbf{ fi}$
(c) $run \;=\; \textbf{if } even\ i \textbf{ then } i{:=}\,i/2 \textbf{ else } i{:=}\,i{+}1 \textbf{ fi}.\;\; \textbf{if } i{=}1 \textbf{ then } ok \textbf{ else } run \textbf{ fi}$

409 Let i be an extended integer variable in the refinement

$$P \;\Longleftarrow\; \textbf{if } i{=}0 \textbf{ then } ok \textbf{ else } i{:=}\,i{-}1.\;\; t{:=}\,t{+}1.\;\; P \textbf{ fi}$$

Investigate how recursive construction is affected when we start with

(a) $t' = \infty$
(b) $t{:=}\,\infty$
(c) ok

410 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

$P \Leftarrow x' \geq 0.\ Q$

$Q \Leftarrow$ **if** $x{=}0$ **then** ok **else** $x{:=}\ x{-}1.\ Q$ **fi**

Assume that $x' \geq 0$ takes no time.

411 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

$S \Leftarrow$ **if** $x{=}0$ **then** ok
else if $x{>}0$ **then** $x{:=}\ x{-}1.\ S$
else $x' \geq 0.\ S$ **fi fi**

Assume that $x' \geq 0$ takes no time.

(b) What do we get from recursive construction starting with $t' \geq t$?

412 Prove that the following three ways of defining R are equivalent.

$R = ok \vee (R.\ S)$

$R = ok \vee (S.\ R)$

$R = ok \vee S \vee (R.\ R)$

413 Prove the laws of Refinement by Steps and Refinement by Parts for **while**-loops.

414 Prove that

$\forall \sigma, \sigma' \cdot\ t' \geq t \wedge$ **if** b **then** $P.\ t{:=}\ t{+}1.\ W$ **else** ok **fi** $\Leftarrow W$

$\Leftarrow\ \forall \sigma, \sigma' \cdot$ **while** b **do** P **od** $\Leftarrow W$

is equivalent to the **while** construction axioms, and hence that construction and induction can be expressed together as

$\forall \sigma, \sigma' \cdot\ t' \geq t \wedge$ **if** b **then** $P.\ t{:=}\ t{+}1.\ W$ **else** ok **fi** $\Leftarrow W$

$=\ \forall \sigma, \sigma' \cdot$ **while** b **do** P **od** $\Leftarrow W$

415 The notation **do** P **while** b **od** has been used as a loop construct that is executed as follows. First P is executed; then b is evaluated, and if b is $\top$, execution is repeated, and if b is $\bot$, execution is finished. Define **do** P **while** b **od** by construction and induction axioms.

416 Using the definition of Exercise 415, but ignoring time, prove

(a) **do** P **while** b **od** $= P.$ **while** b **do** P **od**

(b) **while** b **do** P **od** $=$ **if** b **then do** P **while** b **od else** ok **fi**

(c) $(\forall \sigma, \sigma' \cdot D =$ **do** P **while** b **od**$) \wedge (\forall \sigma, \sigma' \cdot W =$ **while** b **do** P **od**$)$

$= (\forall \sigma, \sigma' \cdot (D = P.\ W)) \wedge (\forall \sigma, \sigma' \cdot W =$ **if** b **then** D **else** ok **fi**$)$

417 Let the state consist of binary variables b and c . Let

$W =$ **if** b **then** $P.\ W$ **else** ok **fi**

$X =$ **if** $b \vee c$ **then** $P.\ X$ **else** ok **fi**

(a) Find a counterexample to $W.\ X = X$.

(b) Now let W and X be the weakest solutions of those equations, and prove $W.\ X = X$.

418 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 $t' \geq t$? Is it a strongest implementable solution?

(d) Now let x be an integer variable, and redo the question.

419 In natural variable n, ignoring time, find three specifications P satisfying

$$P \;=\; P.\ n = 2 \times n'$$

420 Suppose we define **while** b **do** P **od** by ordinary construction and induction, ignoring time.

$$\textbf{if } b \textbf{ then } P.\ \textbf{while } b \textbf{ do } P \textbf{ od else } ok \textbf{ fi} \;\Leftarrow\; \textbf{while } b \textbf{ do } P \textbf{ od}$$

$$\forall \sigma, \sigma' \cdot \textbf{if } b \textbf{ then } P.\ W \textbf{ else } ok \textbf{ fi} \;\Leftarrow\; W \;\Longrightarrow\; \forall \sigma, \sigma' \cdot \textbf{while } b \textbf{ do } P \textbf{ od} \;\Leftarrow\; W$$

Prove that fixed-point construction and induction

$$\textbf{while } b \textbf{ do } P \textbf{ od} \;=\; \textbf{if } b \textbf{ then } P.\ \textbf{while } b \textbf{ do } P \textbf{ od else } ok \textbf{ fi}$$

$$\forall \sigma, \sigma' \cdot W \;=\; \textbf{if } b \textbf{ then } P.\ W \textbf{ else } ok \textbf{ fi} \;\Longrightarrow\; \forall \sigma, \sigma' \cdot \textbf{while } b \textbf{ do } P \textbf{ od} \;\Leftarrow\; W$$

are theorems.

421 Suppose we define **while** b **do** P **od** by fixed-point construction and induction, ignoring time.

$$\textbf{while } b \textbf{ do } P \textbf{ od} \;=\; \textbf{if } b \textbf{ then } P.\ \textbf{while } b \textbf{ do } P \textbf{ od else } ok \textbf{ fi}$$

$$\forall \sigma, \sigma' \cdot W \;=\; \textbf{if } b \textbf{ then } P.\ W \textbf{ else } ok \textbf{ fi} \;\Longrightarrow\; \forall \sigma, \sigma' \cdot \textbf{while } b \textbf{ do } P \textbf{ od} \;\Leftarrow\; W$$

Prove that ordinary construction and induction

$$\textbf{if } b \textbf{ then } P.\ \textbf{while } b \textbf{ do } P \textbf{ od else } ok \textbf{ fi} \;\Leftarrow\; \textbf{while } b \textbf{ do } P \textbf{ od}$$

$$\forall \sigma, \sigma' \cdot \textbf{if } b \textbf{ then } P.\ W \textbf{ else } ok \textbf{ fi} \;\Leftarrow\; W \;\Longrightarrow\; \forall \sigma, \sigma' \cdot \textbf{while } b \textbf{ do } P \textbf{ od} \;\Leftarrow\; W$$

are theorems. Warning: this is hard, and requires the use of limits.

End of Recursive Definition

10.7 Theory Design and Implementation

422 (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?

423√ Implement data-stack theory to make the two binary expressions

$$pop\ empty = empty$$

$$top\ empty = 0$$

antitheorems.

424 Prove the following definitions implement simple data-stack theory (Subsection 7.0.2).

$$stack \;=\; [nil], [stack; X]$$

$$push \;=\; \langle s{:}\ stack \cdot \langle x{:}\ X \cdot [s; x] \rangle \rangle$$

$$pop \;=\; \langle s{:}\ stack \cdot s\ 0 \rangle$$

$$top \;=\; \langle s{:}\ stack \cdot s\ 1 \rangle$$

425 (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.

426 (data-queue implementation) Implement the data-queue theory presented in Section 7.0.

427 In data-queue theory (Subsection 7.0.3), prove that if you start with an empty queue, and join two items, the first item joined is the front of the queue.

428 (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?

429 Prove that the program-stack implementation given in Subsection 7.1.1 satisfies the program-stack axioms of Subsection 7.1.0.

430 From the axioms of simple program stack theory (Subsection 7.1.0), prove

$$top'=3 \Leftarrow push\ 3.\ push\ 4.\ push\ 5.\ pop.\ pop$$

which says that when we push something onto the stack, we find it there later at the appropriate time.

431 Implement weak program-stack theory (Subsection 7.1.3) as follows: the implementer's variable is a list that grows and never shrinks. A popped item must be marked as garbage.

432 From the axioms of program queue theory (Subsection 7.1.4), prove

(a) $front'=3 \Leftarrow mkempty.\ join\ 3$

(b) $front'=4 \Leftarrow mkempty.\ join\ 3.\ join\ 4.\ leave$

433 (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.

434 (general trees) In general, each node of a tree may have any number of subtrees.

(a) Design a data theory for general trees.

(b) Implement your theory.

(c) Prove your implementation.

435 (leaf count) Write a program to count the number of leaves in a binary tree.

436 (leafy tree) A leafy tree is a tree with information residing only at the leaves. Design appropriate axioms for a binary leafy data-tree.

437 (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.

438 (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 ?

439 (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 ?

440 (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.

441 (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.

442 A particular program-list has the following operations:
- the operation *mkempty* makes the list empty
- the operation *extend x* joins 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.

443 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.

444 (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).

445 (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.

446 (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) Write a program to find an item in a binary search tree.

(d) Write a program to add an item to a binary search tree as a new leaf.

(e) Write a program to make a list of the items in a binary search tree in order.

(f) Write a program to determine whether two binary search trees have the same items.

447 (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 conviviality is maximized.

448 (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.

449 (program list) A program list is a list with an associated index, and the following operations: *item* gives the value of the indexed item; *set x* changes the value of the indexed item to *x*; *goLeft* moves the index one item to the left; *goRight* moves the index one item to the right.

(a) Design axioms for a doubly infinite program list.

(b) Using your theory from part (a), prove

$$goLeft.\ set\ 3.\ goRight.\ set\ 4.\ goLeft \implies item'=3$$

450 Implement the program-tree theory of Subsection 7.1.5 in which the tree is infinite in all directions. At any time, only the part of the tree that has been visited needs representation.

451√ (parsing) Define E as a bunch of strings of lists of characters satisfying
E = [“x”], [“if”]; E; [“then”]; E; [“else”]; E; [“fi”]
Given a string of lists of characters, write a program to determine if the string is in the bunch E.

452 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?

453 The user's variable is binary b. The implementer's variables are natural x and y. The operations are:
done = $b := x=y=0$
step = **if** $y>0$ **then** $y := y-1$ **else** $x := x-1$. **new** n: *nat*· $y := n$ **fi**
Replace the two implementer's variables x and y with a single new implementer's variable: natural z.

454 A theory provides three names: *zero*, *increase*, and *inquire*. It is presented by an implementation. Let u: *bin* 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: *bin* 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)

455 A theory provides three names: *set*, *flip*, and *ask*. It is presented by an implementation. Let u: *bin* be the user's variable, and let v: *bin* 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?

456 Let a, b, and x be natural variables. Variables a and b are implementer's variables, and x is a user's variable for the operations
start = $a := 0$. $b := 0$
step = $a := a+1$. $b := b+2$
ask = $x := a+b$
Reimplement this theory replacing the two old implementer's variables a and b with one new natural implementer's variable c.
(a) What is the data transformer?
(b) Using your data transformer, transform *step*.

457 (co-ordinates) In a graphical program, a pixel might be identified by its Cartesian co-ordinates x and y, or by its polar co-ordinates r (radius, or distance from the origin) and a (angle in radians counter-clockwise from the x axis). An operation written using one kind of co-ordinates may need to be transformed into the other kind of co-ordinates.

(a) What is the data transformer to transform from Cartesian to polar co-ordinates?

(b) In Cartesian co-ordinates, one of the operations on a pixel is *translate*, which moves a pixel from position x and y to position $x+u$ and $y+v$.

$$translate = x := x+u.\ y := y+v$$

Use the data transformer from (a) to transform operation *translate* from Cartesian to polar co-ordinates.

(c) What is the data transformer to transform from polar to Cartesian co-ordinates?

(d) In polar co-ordinates, one of the operations on a pixel is *rotate* by d radians.

$$rotate = a := a+d$$

Use the data transformer from (c) to transform operation *rotate*.

458 Let n be a natural constant. Let S: $n*nat$ be an implementer's variable. It is being reimplemented by R: $n*nat$ representing the same n naturals but in the reverse order.

(a) What is the data transformer?

(b) A user has variable i: nat and the operation

$$get = i := S_i$$

Use your transformer from part (a) to transform *get*.

459 (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) $A := (i;j) \rightarrow x \mid A$

460√ (security switch) A security switch has three binary 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.

461 Let p be a user's binary 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.

462√ (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.

463 Find a data transformer to transform the program of Exercise 330(a) into the program of Exercise 330(b).

464√ A limited queue is a queue with a limited number of places for items. Let the limit be n: $nat+1$, and let Q: $[n*X]$ and p: nat be implementer's variables. Here is an implementation.

$mkemptyq = p := 0$
$isemptyq = p=0$
$isfullq = p=n$
$join\ x = Q\ p := x.\ p := p+1$
$leave = \mathbf{for}\ i := 1;..p\ \mathbf{do}\ Q\ (i-1) := Q\ i\ \mathbf{od}.\ p := p-1$
$front = Q\ 0$

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.

465 (transformation incompleteness) The user's variable is i and the implementer's variable is j , both of type $0, 1, 2$. The operations are:

$initialize = i'=0$
$step = \mathbf{if}\ j>0\ \mathbf{then}\ i := i+1.\ j := j-1\ \mathbf{else}\ ok\ \mathbf{fi}$

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 binary variable b so that

initialize	is transformed to	$i'=0$
step	is transformed to	$\mathbf{if}\ b \wedge i<2\ \mathbf{then}\ i' = i+1\ \mathbf{else}\ ok\ \mathbf{fi}$

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 the increased value of i will not exceed 2 . 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 is a constant, k: $0, 1, 2$.

466 Subsection 7.1.5 presented program-tree theory.
(a) Implement the theory, storing all node values in a data structure.
(b) Transform the implementation so that an assignment to *node* does not update the main data structure used for storing the node values; updates occur when you *go* from a node.

467 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

$$\begin{array}{lcl} goHome & = & p := 1 \\ goLeft & = & p := 2\times p \\ goRight & = & p := 2\times p + 1 \\ goUp & = & p := div\ p\ 2 \\ put & = & s := p \rightarrow x \mid s \\ get & = & x := s\ p \end{array}$$

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.

468 (weak limited program bunches) Given natural number n, a theory maintains a subbunch of $0,..n$. The operations are: *mkempty*, which makes the bunch empty; *insert* x, which inserts x into the bunch; *remove* x, which removes x if it was there, and *check* x which tells whether x is there by assigning to a user's binary variable u.

(a) Design axioms that are weak enough to allow other operations to be added to the theory.
(b) Implement your theory of part (a) as a list of binary values.
(c) Transform your implementation of part (b) to one that maintains a list of natural numbers.

469 (row major) The usual way to represent a 2-dimensional array in a computer's memory is in row major order, stringing the rows together. For example, the 3×4 array

[[5; 2; 7; 3] ;
  [8; 4; 2; 0] ;
  [9; 2; 7; 7]]

is represented as 5; 2; 7; 3; 8; 4; 2; 0; 9; 2; 7; 7.

(a) Given naturals n and m, find a data transformer that transforms an $n\times m$ array A to its row major representation B.
(b) Using your transformer, transform $x := A\ y\ z$ where x, y, and z are user's variables.

470 Let u be a binary user's variable. Let a and b be old binary implementer's variables. We replace a and b by new integer implementer's variables x and y using the convention (from the C language) that 0 stands for $\bot$ and non-zero integers stand for $\top$.

(a) What is the transformer?
(b) Transform $a := \neg a$.
(c) Transform $u := a \wedge b$.

471 The user's variable is natural n. The implementer's variable is set S. The operations are

$$\begin{array}{lcl} start & = & S := \{null\} \\ insert & = & S := S \cup \{\{S\}\} \\ ask & = & n := \$S \end{array}$$

Operation *start* starts variable S at the empty set. Then repeated use of operation *insert* increases the size of the set. Operation *ask* asks how large the set is. Reimplement this theory replacing the old implementer's variable S with natural variable m.

(a) What is the data transformer?
(b) Transform operation *ask*.

472 The user's variables are binary b and natural x. Using implementer's variables L: [*nat] and i: nat, we implement the following operations.

$init = L := [nil]$
$start = i := 0$
$insert = L := L[0;..i];;[x];;L[i;..\#L]$
$delete = L := L[(0;..i); (i+1;..\#L)]$
$next = i := i+1$
$end = b := i=\#L$
$value = x := L\ i$
$set = L := i \rightarrow x \mid L$

Transform the operations to provide a heap implementation. Inserted nodes come from a free list, and deleted nodes are returned to the free list.

473 Implementer's variables p, q: $real$ represent two points along a line. Each number tells the distance of one point from the origin (a standard point). They must be reimplemented by one implementer's variable r: $real$ which tells the distance from p to q. For examples, if $p=3$ and $q=5$, then $r=2$; if $p=5$ and $q=3$, then $r=-2$.

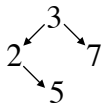
(a) What is the data transformer?

(b) A user has binary variable b and operation

$compare = b := q \geq p$

Use your transformer from part (a) to transform operation $compare$.

474 An implementer's variable A holds a binary tree representation as follows. If the tree is empty, $A = [nil]$. If the tree has left subtree L and right subtree R and root value n, then $A = [L; n; R]$. The tree



is represented as $A = [[[nil]; 2; [[nil]; 5; [nil]]]; 3; [[nil]; 7; [nil]]]$. The tree must be reimplemented using implementer's variable B as follows. If the tree is empty, $B = 0$. If the tree has left subtree L and right subtree R and root value n, then

$B =$ "left" $\rightarrow L \mid$ "root" $\rightarrow n \mid$ "right" $\rightarrow R$

The same example tree is represented as

```
B =   "left" →  ("left" → 0
                 | "root" → 2
                 | "right" →  ("left" → 0
                               | "root" → 5
                               | "right" → 0 ) )
      | "root" → 3
      | "right" → ("left" → 0
                   | "root" → 7
                   | "right" → 0 )
```

(a) What is the data transformer?

(b) A user has natural variable n and the operation

$root = n := A\ 1$

which assigns to n the root value. Use your transformer from part (a) to transform $root$.

475 An old implementer's variable c: $-1, 0, 1$ is being replaced by new implementer's variables a, b: *bin* such that $c{=}{-1}$ is replaced by a and b both being $\bot$, $c{=}1$ is replaced by a and b both being $\top$, and $c{=}0$ is replaced by a and b being unequal.

(a) What is the transformer?

(b) Use your transformer to transform $c := 0$.

476 Let b: *bin* be the user's variable, and let n: *nat* be the implementer's variable, and let the operations be

step = **if** $n{>}0$ **then** $n := n{-}1$ **else** *ok* **fi**

done = $b := n{=}0$

Show that there is no transformer to get rid of n so that

step is transformed to *ok*

done is transformed to $b := \bot$

even though the user cannot detect the difference.

End of Theory Design and Implementation

10.8 Concurrency

477 Invent an example of concurrent composition for which there are two reasonable ways to partition the variables that give different meanings to the composition. Hint: the operands of the concurrent composition don't have to be programs.

478 Let a, b, and c be integer variables. Simplify

$a := a{+}b.\ (b := a{-}b \parallel a := a{-}b)$

479 Let x and y be natural variables. Ignoring time, rewrite the following program as a program that does not use $\parallel$.

(a) $x := x{+}1 \parallel$ **if** $x{=}0$ **then** $y := 1$ **else** *ok* **fi**

(b) **if** $x{>}0$ **then** $y := x{-}1$ **else** *ok* **fi** $\parallel$ **if** $x{=}0$ **then** $x := y{+}1$ **else** *ok* **fi**

480 Let a and b be integer variables. Refine $a' = a{+}b \ \wedge\ b' = a{-}b$ by replacing the question marks in the following. Prove that your answer is a refinement.

(a) $a := ?.\ b := ?$

(b) $b := ?.\ a := ?$

(c) $a := ? \parallel b := ?$

481 (research) If we ignore time, then

$x := 3.\ y := 4 \ =\ x := 3 \parallel y := 4$

Some sequential compositions could be executed concurrently 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 concurrent 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 concurrent.

$t := t{+}1 \parallel t := t{+}2 \ =\ t := t{+}2$

Invent another form of composition, intermediate between sequential and concurrent composition, whose execution is sequential to the extent necessary, and concurrent to the extent possible.

482 Let a, b, and c be number variables. Using concurrency, write a program to sort the values of these variables so that $a' \le b' \le c'$, and $(a'; b'; c')$ is a permutation of $(a; b; c)$.

483 (disjoint composition) Concurrent 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.

484 Extend the definition of disjoint composition $P \,|v|w|\, Q$ (Exercise 483) from variables to list items.

485 (merged composition) Concurrent 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 merged composition $P \| \| Q$. Like sequential composition, it requires P and Q to have the same state variables. Like concurrent composition, it can be executed by executing the processes concurrently, 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 merged composition, including time.

(b) What laws apply to merged 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

$$x{:=}\,y \;\|\|\; x{:=}\,z \;=\; \textbf{if } x{=}y \textbf{ then } x{:=}\,z \textbf{ else if } x{=}z \textbf{ then } x{:=}\,y \textbf{ else } x{:=}\,y \;\|\|\; x{:=}\,z \textbf{ fi fi}$$

(h) In binary 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

$$\begin{array}{ll} & w{:=}\,2 \times (\mathit{div}\ w\ 2 \uparrow z) \;+\; \mathit{mod}\ w\ 2 \uparrow (1{-}z) \\ \|\| & w{:=}\,2 \times (\mathit{div}\ w\ 2 \uparrow (1{-}z)) \;+\; \mathit{mod}\ w\ 2 \uparrow z \end{array}$$

486 Extend the definition of merged composition $P \| \| Q$ (Exercise 485) from variables to list items.

487 Redefine merged composition $P|||Q$ (Exercise 485) 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.

488 (sieve) Given variable p: $[n*bin] := [\bot; \bot; (n–2)*\top]$, 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:= $(i\times j)\rightarrow\bot \mid p$ **od**
else *ok* **fi od**

(a) Show how the program can be transformed for concurrency.

(b) What is the execution time, as a function of n, with maximum concurrency?

489 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 concurrent transformation gives the desired computation.

490 Exercise 161 asks for a program to compute cumulative sums (running total). Write a program that can be transformed from sequential to concurrent execution with $log\ n$ time where n is the length of the list.

491√ (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.

492 (simultaneous equations) We are given string variable X whose n items are rational, and a string of n functions f_i, each of which takes n rational arguments and produces a rational result. Assign to X a value satisfying

$$\forall i: 0,..n \cdot X_i = f_i@X$$

or, spreading it out,

$$\begin{aligned} X_0 &= f_0\ X_0\ X_1 \cdots X_{n-1} \\ X_1 &= f_1\ X_0\ X_1 \cdots X_{n-1} \\ &\vdots \\ X_{n-1} &= f_{n-1}\ X_0\ X_1 \cdots X_{n-1} \end{aligned}$$

In other words, find n simultaneous fixed-points. Assume that a repetition of assignments of the form $X_i := f_i\ X_0\ X_1 \cdots X_{n-1}$ will result in an improving sequence of approximations until the value of X is "close enough", within some tolerance. Function evaluation is the time-consuming part of the computation, so as much as possible, function evaluations should be done concurrently.

493 In a language with array element assignment and list concurrency, what is the exact precondition for

if $\#A \le 1$ **then** *ok*
else if $A\,0 \le A\,(\#A-1)$ **then** *ok*
else $A\,0 := A\,(\#A-1) \parallel A\,(\#A-1) := A\,0$ **fi fi**

to refine

$\forall i, j: \Box A \cdot\ i \le j \Rightarrow A'i \le A'j$

End of Concurrency

10.9 Interaction

494 Express the program

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

without using assignment, sequential composition, or concurrent composition, where t is time, and x and y are

(a) boundary variables and assignment takes time 0.
(b) interactive variables and assignment takes time 1.

495√ Suppose a and b are integer boundary variables, x and y are integer interactive variables, and t is an extended natural time variable. Suppose that each assignment takes time 1. Express the following without using assignment, sequential composition, or concurrent composition.

$(x := 2.\ x := x+y.\ x := x+y) \parallel (y := 3.\ y := x+y)$

496√ (grow slow) Suppose *alloc* allocates 1 unit of memory space and takes time 1 to do so. Then the following computation slowly allocates memory.

$GrowSlow \Leftarrow$ **if** $t = 2 \times x$ **then** *alloc* $\parallel x := t$ **else** $t := t+1$ **fi**. $GrowSlow$

If the time is equal to $2 \times x$, then one space is allocated, and concurrently 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.

497 Let a and b be boundary variables; assignment to them takes time 0. Let assignment to interactive variables take time 1. Let t be extended natural time. Define

wait w $=$ $t := t+w$

Simplify

new x: $time \to nat \cdot$ $(x := 1.\ x := 2) \parallel (a := 3.$ **wait** $1.\ b := a+x.$ **wait** $1.\ a := b+x)$

498 Let a and b be binary interactive variables. Define

$loop$ $=$ **if** b **then** $loop$ **else** *ok* **fi**

Add a time variable according to any reasonable measure, and then express

$b := \bot \parallel loop$

as an equivalent program but without using $\parallel$.

499 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.

500√ (thermostat) Specify a thermostat for a gas burner. The thermostat operates concurrently 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.
- binary *flame* , which comes from a flame sensor in the burner and indicates whether there is a flame.

The outputs of the thermostat are:

- binary *gas* ; assigning it $\top$ turns the gas on and $\bot$ turns the gas off.
- binary *spark* ; assigning it $\top$ causes sparks for the purpose of igniting the gas.

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. 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. The gas burner must respond to its inputs within 1 second.

501 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.

502 (interactive data transformation; research) Section 7.2 presented data transformation for boundary variables. How do we do data transformation when there are interactive variables?

503 (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, there is no output. If the phone is put down and then picked up again within 2 seconds, this is considered to be an accident, and does not affect the output.

504 (consensus) Some concurrent 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.

505 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?

506 Write a program to print the sequence of natural numbers, one per time unit.

507 Write a program to repeatedly print the current time, up until some given time.

508 The equation

$$printTime \;=\; screen!\, t.\;\; t := t+1.\;\; printTime$$

outputs the time t onto the *screen* channel once each time unit forever. Considering *printTime* as the unknown,

(a) what is the weakest solution to this equation? (No proof required.)

(b) what is the strongest solution to this equation? (No proof required.)

509 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.

510 (generate) Write a program to print all texts consisting of only the characters "a", "b", and "c" in this order: shorter texts come before longer texts; texts of equal length are in alphabetical order.

511 (T-strings) Let us call a string S: *("a", "b", "c") a T-string if no two adjacent nonempty segments are identical:

$$\neg \exists i, j, k \cdot\; 0 \le i < j < k \le \leftrightarrow S \;\wedge\; S_{i;..j} = S_{j;..k}$$

Write a program to output all T-strings in alphabetical order. (The mathematician Axel Thue proved that there are infinitely many T-strings.)

512 According to the definition of **value** expression given in Subsection 5.5.0, what happens to any input or output?

513 (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.

514 (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.

515 (input implementation) Let W be "wait for input on channel c and then read it".

(a)√ $$W \;=\; t := t \uparrow (\mathcal{T}_r + 1).\;\; c?$$

Prove $W \;\Longleftarrow\;$ **if** $\sqrt{c}$ **then** $c?$ **else** $t := t+1.\;\; W$ **fi** where time is an extended natural.

(b) Now let time be a nonnegative extended real, redefine W, and reprove the refinement.

516 (input with timeout) As in Exercise 515, 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.

$$W \Longleftarrow \textbf{if } t \le \mathit{deadline} \textbf{ then if } \surd c \textbf{ then } c? \textbf{ else } t := t+1.\ W \textbf{ fi else } \mathit{alarm} \textbf{ fi}$$

Define W appropriately, and prove the refinement.

517 Here are two definitions.

$$\begin{aligned}
A \ = \ & \textbf{if } \surd c \wedge \surd d \textbf{ then } c? \vee d? \\
& \textbf{else if } \surd c \textbf{ then } c? \\
& \quad \textbf{else if } \surd d \textbf{ then } d? \\
& \quad\quad \textbf{else if } \mathcal{T}c_{\mathbf{r}c} < \mathcal{T}d_{\mathbf{r}d} \textbf{ then } t := \mathcal{T}c_{\mathbf{r}c} + 1.\ c? \\
& \quad\quad\quad \textbf{else if } \mathcal{T}d_{\mathbf{r}d} < \mathcal{T}c_{\mathbf{r}c} \textbf{ then } t := \mathcal{T}d_{\mathbf{r}d} + 1.\ d? \\
& \quad\quad\quad\quad \textbf{else } t := \mathcal{T}c_{\mathbf{r}c} + 1.\ c? \vee d? \textbf{ fi fi fi fi fi} \\
B \ = \ & \textbf{if } \surd c \wedge \surd d \textbf{ then } c? \vee d? \\
& \textbf{else if } \surd c \textbf{ then } c? \\
& \quad \textbf{else if } \surd d \textbf{ then } d? \\
& \quad\quad \textbf{else } t := t+1.\ B \textbf{ fi fi fi}
\end{aligned}$$

Letting time be an extended natural, prove $A = B$.

518 (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.

519 Define relation $\mathit{partmerge}$: $\mathit{nat} \to \mathit{nat} \to \mathit{bin}$ as follows:

$$\begin{aligned}
& \mathit{partmerge}\ 0\ 0 \\
& \mathit{partmerge}\ (m+1)\ 0 \ = \ \mathit{partmerge}\ m\ 0 \ \wedge \ \mathcal{M}c_{\mathbf{w}c+m} = \mathcal{M}a_{\mathbf{r}a+m} \\
& \mathit{partmerge}\ 0\ (n+1) \ = \ \mathit{partmerge}\ 0\ n \ \wedge \ \mathcal{M}c_{\mathbf{w}c+n} = \mathcal{M}b_{\mathbf{r}b+n} \\
& \mathit{partmerge}\ (m+1)\ (n+1) \ = \ \mathit{partmerge}\ m\ (n+1) \ \wedge \ \mathcal{M}c_{\mathbf{w}c+m+n+1} = \mathcal{M}a_{\mathbf{r}a+m} \\
& \qquad\qquad\qquad\qquad \vee \ \mathit{partmerge}\ (m+1)\ n \ \wedge \ \mathcal{M}c_{\mathbf{w}c+m+n+1} = \mathcal{M}b_{\mathbf{r}b+n}
\end{aligned}$$

Now $\mathit{partmerge}\ m\ n$ 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

$$\mathit{merge} \ = \ (a?.\ c!\ a) \vee (b?.\ c!\ b).\ \mathit{merge}$$

Prove $\mathit{merge} \ = \ (\forall m \cdot \exists n \cdot \mathit{partmerge}\ m\ n) \vee (\forall n \cdot \exists m \cdot \mathit{partmerge}\ m\ n)$

520 (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

$$\begin{aligned}
& \mathcal{T}e_{\mathbf{w}e} \ = \ t \uparrow (\mathcal{T}c_{\mathbf{r}c} \downarrow \mathcal{T}d_{\mathbf{r}d} + 1) \\
& \forall m, n \cdot \mathcal{T}e_{\mathbf{w}e+m+n+1} \le \mathcal{T}c_{\mathbf{r}c+m} \uparrow \mathcal{T}d_{\mathbf{r}d+n} \uparrow \mathcal{T}e_{\mathbf{w}e+m+n} + 1
\end{aligned}$$

521 (fairer time merge) This question is the same as the time merge (Exercise 520), 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.

522 Let t be an extended natural time variable. Is the following specification implementable?

(a) $\forall n\colon nat\cdot\ \mathcal{M}_n{=}n \wedge \mathcal{T}_n{=}t$

(b) $\forall n\colon nat\cdot\ \mathcal{M}_{w+n}{=}n{-}t \wedge \mathcal{T}_{w+n}{=}t{-}n$

(c) $\forall n\colon nat\cdot\ \mathcal{M}_{r+n}{=}n \wedge \mathcal{T}_{r+n}{=}t$

(d) $\mathcal{M}_w = \mathcal{T}_w = t{-}1$

523 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.

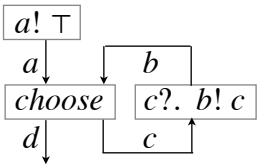
524 In the program

new $c?$ *int*$\cdot$ $c?$

(a) add the time spent waiting for input according to the transit time measure.

(b) including the time (from part (a)), rewrite the program without using $?$, and simplify as much as possible.

525 (choose) The following picture shows a network of communicating processes.



The formal description of this network is

new $a, b, c?$ *bin*$\cdot$ $a!\ \top \parallel$ *choose* $\parallel (c?.\ b!\ c)$

Formally define *choose*, add transit time, and state the output message and time if

(a) *choose* either reads from a and then outputs $\top$ on c and d, or reads from b and then outputs $\bot$ on c and d. The choice is made freely.

(b) as in part (a), *choose* either reads from a and then outputs $\top$ on c and d, or reads from b and then outputs $\bot$ 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).

526√ (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 concurrently 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 concurrently 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.

527 (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.

528√ Prove that execution of the following program deadlocks.

(a) **new** $c?\ int\cdot\ c?.\ c!\ 5$

(b) **new** $c, d?\ int\cdot\ (c?.\ d!\ 6) \parallel (d?.\ c!\ 7)$

529 (repetition) Write a program to read an infinite sequence, and after every even number of inputs, to output a binary value saying whether the second half of the input so far is a repetition of the first half.

530 (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 concurrently $(P \parallel Q)$. 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.

531 (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.

532 Section 5.3 defined and implemented the program **wait until** w where w is a time. Define and implement the program **wait until** b where b is a binary expression. For example, **wait until** $x{=}y$ should delay execution until variables x and y are equal. At least one variable in the expression should be an interactive variable belonging to another process.

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

In the first edition, I used λ notation for functions. 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 brackets, 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 brackets 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 brackets. 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 brackets. One can immediately see generalizations of this convention to all symbols and many sizes (a slippery slope). I considered big brackets $( )$ $\{ \}$ $[ ]$ $\langle \rangle$, but decided against.

End of Notation

11.0.1 Basic Theories

Binary Theory is sometimes called Boolean Algebra, Propositional Calculus, or 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 binary expressions and treat them on a par with number expressions and expressions of other types. I consider that predicates are just binary functions. I use the same equal sign for binary expressions 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 Subsection 1.0.0, 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 metalanguage. 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

$\langle v\cdot b\rangle\, a \;=\; b[a/v]$

but then I would have to explain that $b[a/v]$ means “substitute a for v in b”. I may as well say directly

$\langle v\cdot b\rangle\, a \;=\; (\text{substitute } a \text{ for } v \text{ in } b)$

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 joins (denoted here by adjacency) 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 collection (aggregation); Set Theory tells us about packaging (nesting). 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 (collection), 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". The problem was already in mathematics before programming. For example, 1 has two square roots, therefore $1^{1/2} = 1, -1$. And $(1^{1/2})^2 = (1, -1)^2 = 1^2, (-1)^2 = 1, 1 = 1$.

Type theory duplicates all the operators of its value space: for each operation on values, there is a corresponding operation on type spaces. By using bunches as types, this duplication is eliminated.

Many mathematicians consider that curly brackets and commas are just syntax. 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 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 joining lists will be abused by sometimes joining a list and an item. Or perhaps there are three join notations: one to join 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.)

Let x be real, and let p be positive real. Then $0 < 0;x < p$. Hence $0;x$ is an infinitesimal, larger than 0 but smaller than any positive real. And $0 < 0;0;x < 0;p < p$, so $0;0;x$ is an infinitesimal smaller than $0;p$. And so on. Similarly $x < \infty < \infty;x < \infty;\infty < \infty;\infty;x$ and so on. More infinitesimals and infinities can be created by using sets and lists, and they can be related to the infinities we already have by strengthening the axioms $A \neq \{A\}$ and $S \neq [S]$ to $A < \{A\}$ and $S < [S]$ (see Exercise 67). But this book is not about infinitesimals and infinities.

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 $\langle\cdot\rangle$ is said to "bind" variables, and any variable that is not bound remains "free". For example, $\langle x{:}\ int\cdot\ x{+}y\rangle$ 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. "Universal quantification" would be better named "conjunction quantification", and "existential quantification" would be better named "disjunction quantification". I call the quantifier $\Uparrow$ "maximum" even though its result may not be any result of the function it is applied to; the name "least upper bound" is traditional. Similarly I call $\Downarrow$ "minimum"; it 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 does tell us a lower and upper bound. The domains could be rat or $real$ or com rather than nat.

End of Function Theory

11.0.4 Program Theory

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 binary expression: a specification (of anything) really is a binary 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 binary 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.

Assignment could have been defined as weakly as

$$x{:=}\,e \;=\; \mathit{defined}\ \text{“}e\text{”} \wedge e{:}\ T \Rightarrow x'{=}e \wedge y'{=}y \wedge \ldots$$

or as strongly as

$x := e \;=\; \mathit{defined}\ \text{“}e\text{”} \wedge e{:}\ T \wedge x'{=}e \wedge y'{=}y \wedge \ldots$

where *defined* rules out expressions like $1/0$, and T is the type of variable x. I left out *defined* because its definition is as complicated as all of program theory, and it serves no purpose. It is questionable whether $e{:}\ T$ is useful, since an out-of-type value already makes assignment unimplementable. I prefer to keep assignment, and the Substitution Law, simple.

Since the design of ALGOL in 1958, 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 join. Sequential 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 sequential composition and a period for string join, but the latter did not work well.

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).

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 variable local 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 three problems with these notations. The first is an abuse of notation. *wp* uses functional syntax, but its second argument appears to be of type binary, which is not what is intended. The next problem is that these notations 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 last 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.

End of Program Theory

11.0.5 Programming Language

The form of variable declaration given in Subsection 5.0.0 assigns the new local variable an arbitrary value of its type. Thus, for example, if y and z are integer variables, then

new x: nat· y:= x = y': nat ∧ z'=z

For ease of implementation and speed of execution, this is better than initialization with "the undefined value". For error detection, it is no worse, if we prove all our refinements. Furthermore, there are circumstances in which arbitrary initialization is exactly what's wanted (see Exercise 330 (majority vote)). If we do not prove all our refinements, initialization with *undefined* provides a measure of protection. If we allow the generic operators (= , ≠ , **if then else fi**) 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

new x: T = x': T ∧ ok

which starts the scope of x , and

old 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 Rule of Invariants and Variants. Let I be an assertion (the "invariant") and let I' be the corresponding expression with primed variables. Let v be an integer expression (the "variant" or "bound function") and let v' be the corresponding expression with primed variables. The Rule of Invariants and Variants says:

$I \Rightarrow I' \wedge \neg b'$ ⇐ **while** b **do** $I \wedge b \Rightarrow I' \wedge 0 \le v' < v$ **od**

The rule says, 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

$s' = s + \Sigma\, L\,[n;..\#L]$ ⇐ **while** $n \neq \#L$ **do** $s := s + L\,n.\ \ n := n+1$ **od**

we must invent an invariant $s + \Sigma\, L\,[n;..\#L] = \Sigma L$ and a variant $\#L - n$ and prove both

$s' = s + \Sigma\, L\,[n;..\#L]$

⇐ $s + \Sigma\, L\,[n;..\#L] = \Sigma L \Rightarrow s' + \Sigma\, L\,[n';..\#L] = \Sigma L \wedge n' = \#L$

and

$s + \Sigma\, L\,[n;..\#L] = \Sigma L \wedge n \neq \#L \Rightarrow s' + \Sigma\, L\,[n';..\#L] = \Sigma L \wedge 0 \le \#L - n' < \#L - n$

⇐ $s := s + L\,n.\ \ n := n+1$

The proof method given in Section 5.2 is easier and more information (time) is obtained. Sometimes the Rule of Invariants and Variants requires the introduction of extra constants (mathematical variables) not required by the proof method in Section 5.2. 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 ⇐ **while** b **do** P **od** is an alternate notation, 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 says that probabilities are the reals in the closed interval from 0 to 1 . It would be simpler if all extended real numbers were probabilities, 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. Other people have approached probabilistic programming by reinterpreting 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 is 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 rather than recursive refinement, 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 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 (Exercise 465) 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 various 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 concurrent composition of sequential compositions, but reluctant to have a sequential composition of concurrent compositions. So in currently popular languages, a concurrent 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 concurrent and sequential composition. This second pattern occurs in the buffer program (Subsection 8.1.0).

In the first edition of this book, concurrent composition was defined for processes having the same state space (merged composition). That definition was more complicated than the present one (see Exercise 485), 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 (concurrent composition) is used.

End of Concurrency

11.0.9 Interaction

The notation for interactive variables has a problem. The declaration

new x: $time \to T \cdot S \ = \ \exists x$: $time \to T \cdot S$

conflicts with the declaration of a boundary variable with type $time \to T$

new x: $time \to T \cdot S \ = \ \exists x, x' \cdot$ $time \to T \cdot S$

and the abbreviation x for $x\,t$ and x' for $x\,t'$ creates ambiguity. But I chose to rely on context to disambiguate, rather than introduce any new notations.

If x is an interactive variable, then $(t'{=}\infty.\ x{:=}\,2.\ x{:=}\,3)$ is unfortunately $\bot$. To eliminate this unwanted result we would have to weaken assignment to an interactive variable with the antecedent $t{<}\infty$. If execution of a program causes an infinite amount of output, and then execution of a sequentially following program causes more output, we have a similar problem. As a simple example, $(\boldsymbol{w}c'{=}\infty.\ c!\,2.\ c!\,3)$ is unfortunately $\bot$. To eliminate this unwanted result we would have to weaken output with the antecedent $\boldsymbol{w}c{<}\infty$. But I think these pathological cases are not worth complicating the theory.

In the formula for implementability, there is no conjunct $\boldsymbol{r}' \le \boldsymbol{w}'$ saying that the read cursor must not get ahead of the write cursor. In Subsection 9.1.8 on deadlock we see that mathematically it can happen, but it takes infinite time to do so (it never happens).

End of Interaction

We could talk about a structure of channels, and about indexed processes. We could talk about a concurrent **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 from 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. (See the Bibliography, which follows.)

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). 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 binary 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 my 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. Robert Will and Lev Naiman persuaded me to add the closing **fi** and **od** brackets. The word "conflation" was suggested by Doug McIlroy. Exercise 29 (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. José Ortiz noticed that two bunch distributivity laws were needed as axioms. 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 backtracking implementation of unimplementable specifications is an adaptation of a technique due to Greg Nelson (1989) for implementing angelic nondeterminism. Lev Naiman found an inconsistency with an earlier version (now Exercise 334(a)) of the **value** axiom. Carroll Morgan and Annabelle McIver (1996) suggested probabilities as observable quantities, and Exercise 346 (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 Theo Norvell and I each checked it for equivalence with those in Wei Chen and Jan Tijmen Udding (1989). Theo 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 concurrent (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 485 (merged composition), and is used in work by Hoare and He (1998); for the current edition of this book I was persuaded by Leslie Lamport to return to my earlier (1984, 1990) version: simple conjunction. Section 8.1 on sequential to concurrent transformation is joint work with Chris Lengauer (1981); he has since made great advances in the automatic production of highly concurrent, systolic computations from ordinary sequential, imperative programs. Phil Clayton found a problem with the definition of implementability for interactive variables. 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). Jianjun Zhao found an error in my unnecessary attempt to strengthen channel declaration. 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).

The theory in this book has been applied to compiler correctness and the generation of machine language programs by Theo Norvell in his PhD thesis (1993). It has been applied to object-oriented programming (including modularity, encapsulation, inheritance, reuse, polymorphism, and unrestricted pointers) by Yannis Kassios in his PhD thesis (2006). It has been applied to quantum programming (including quantum communications and non-locality) by Anya Tafliovich in her PhD thesis (2010). It has been applied to lazy execution by Albert Lai in his PhD thesis (2013); a simplified version of lazy execution timing can be found in my paper (2018). Albert Lai's thesis also proved the consistency and soundness of the program theory in this book.

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, 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, Part 1 and Part 2”, *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

E.C.R.Hehner: “a Theory of Lazy Imperative Timing”, Workshop on Refinement, Oxford U.K., 2018 July

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 International, 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

I.T.Kassios: *a Theory of Object-Oriented Refinement*, PhD thesis, University of Toronto, 2006

A.Y.C.Lai: *Eager, Lazy, and Other Executions for Predicative Programming*, PhD thesis, University of Toronto, 2013

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*, v.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”, *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*, v.18 n.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*, v.6 n.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: *a Predicative Theory of Machine Language and its Application to Compiler Correctness*, PhD thesis, University of Toronto, 1993

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 Science, 1990 March

M.Rem: “Small Programming Exercises”, *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 v.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.Tafliovich: *Predicative Quantum Programming*, PhD thesis, University of Toronto, 2010

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 Laws

11.3.0 Generic

The operators $=\ \neq$ **if then else fi** apply to every type of expression (but the first operand of **if then else fi** must be binary), with the laws

$x{=}x$	Reflexivity	**if** $\top$ **then** x **else** y **fi** $= x$	Case Base
$x{=}y \ = \ y{=}x$	Symmetry	**if** $\bot$ **then** x **else** y **fi** $= y$	Case Base
$x{=}y \wedge y{=}z \Rightarrow x{=}z$	Transitivity	**if** a **then** x **else** x **fi** $= x$	Case Idempotent
$x{=}y \Rightarrow f\,x = f\,y$	Transparency	**if** a **then** x **else** y **fi** $=$ **if** $\neg a$ **then** y **else** x **fi**	
$x{\neq}y \ = \ \neg(x{=}y)$	Unequality		Case Reversal

The operators $\uparrow\ \downarrow\ <\ \leq\ >\ \geq$ apply to numbers, characters, strings, and lists, with the laws

$x{\leq}x$	Reflexivity	$\neg\ x{<}x$	Irreflexivity
$\neg(x{<}y \wedge x{=}y)$	Exclusivity	$\neg(x{>}y \wedge x{=}y)$	Exclusivity
$\neg(x{<}y \wedge x{>}y)$	Exclusivity	$x{\leq}y \ = \ x{<}y \vee x{=}y$	Inclusivity
$x{\leq}y \wedge y{\leq}z \Rightarrow x{\leq}z$	Transitivity	$x{<}y \wedge y{\leq}z \Rightarrow x{<}z$	Transitivity
$x{<}y \wedge y{<}z \Rightarrow x{<}z$	Transitivity	$x{\leq}y \wedge y{<}z \Rightarrow x{<}z$	Transitivity
$x{>}y \ = \ y{<}x$	Mirror	$x{\geq}y \ = \ y{\leq}x$	Mirror
$\neg\ x{<}y \ = \ x{\geq}y$	Totality	$\neg\ x{\leq}y \ = \ x{>}y$	Totality
$x{\leq}y \wedge y{\leq}x \ = \ x{=}y$	Antisymmetry	$x{<}y \vee x{=}y \vee x{>}y$	Totality, Trichotomy
$x\uparrow x = x$	Idempotence	$x\downarrow x = x$	Idempotence
$x\uparrow y = y\uparrow x$	Symmetry	$x\downarrow y = y\downarrow x$	Symmetry
$x\uparrow(y\uparrow z) = (x\uparrow y)\uparrow z$	Associativity	$x\downarrow(y\downarrow z) = (x\downarrow y)\downarrow z$	Associativity
$x\uparrow(y\downarrow z) = (x\uparrow y)\downarrow(x\uparrow z)$	Distributivity	$x\downarrow(y\uparrow z) = (x\downarrow y)\uparrow(x\downarrow z)$	Distributivity
$x\uparrow y \leq z \ = \ x{\leq}z \wedge y{\leq}z$	Connection	$x\downarrow y \leq z \ = \ x{\leq}z \vee y{\leq}z$	Connection
$x \leq y\uparrow z \ = \ x{\leq}y \vee x{\leq}z$	Connection	$x \leq y\downarrow z \ = \ x{\leq}y \wedge x{\leq}z$	Connection
$x\uparrow y \ =$ **if** $x{\geq}y$ **then** x **else** y **fi**		$x\downarrow y \ =$ **if** $x{\leq}y$ **then** x **else** y **fi**	
$x\downarrow y \leq x \leq x\uparrow y$		$x{\leq}y \ = \ x = x\downarrow y$	$x{\leq}y \ = \ y = x\uparrow y$

End of Generic

11.3.1 Binary

Let a , b , c , d , and e be binary.

Binary

$\top$

$\neg\bot$

$\top{\neq}\bot$

Excluded Middle

$a \vee \neg a$

Noncontradiction

$\neg(a \wedge \neg a)$

Base

$\neg(a \wedge \bot)$

$a \vee \top$

$a \Rightarrow \top$

$\bot \Rightarrow a$

Mirror

$a \Leftarrow b \ = \ b \Rightarrow a$

Double Negation

$\neg\neg a \ = \ a$

Duality

$\neg(a \wedge b) \ = \ \neg a \vee \neg b$

$\neg(a \vee b) \ = \ \neg a \wedge \neg b$

Exclusion

$a \Rightarrow \neg b \ = \ b \Rightarrow \neg a$ (Contrapositive)

$a = \neg b \ = \ a \neq b \ = \ \neg a = b$

Inclusion

$a \Rightarrow b \ = \ \neg a \vee b$ (Material Implication)

$a \Rightarrow b \ = \ (a \wedge b \ = \ a)$

$a \Rightarrow b \ = \ (a \vee b \ = \ b)$

Identity

$$\top \wedge a = a$$
$$\bot \vee a = a$$
$$\top \Rightarrow a = a$$
$$\top = a = a$$

Idempotent

$$a \wedge a = a$$
$$a \vee a = a$$

Reflexive

$$a \Rightarrow a$$
$$a = a$$

Indirect Proof

$$\neg a \Rightarrow \bot = a$$
$$\neg a \Rightarrow a = a$$

Specialization

$$a \wedge b \Rightarrow a$$

Associative

$$a \wedge (b \wedge c) = (a \wedge b) \wedge c$$
$$a \vee (b \vee c) = (a \vee b) \vee c$$
$$a = (b = c) = (a = b) = c$$
$$a \neq (b \neq c) = (a \neq b) \neq c$$
$$a = (b \neq c) = (a = b) \neq c$$

Symmetry (Commutative)

$$a \wedge b = b \wedge a$$
$$a \vee b = b \vee a$$
$$a = b = b = a$$
$$a \neq b = b \neq a$$

Antisymmetry (Double Implication)

$$(a \Rightarrow b) \wedge (b \Rightarrow a) = a = b$$

Discharge

$$a \wedge (a \Rightarrow b) = a \wedge b$$
$$a \Rightarrow (a \wedge b) = a \Rightarrow b$$

Antimonotonic

$$a \Rightarrow b = \neg a \Leftarrow \neg b \text{ (Contrapositive)}$$
$$a \Rightarrow b \Longrightarrow (a \Rightarrow c) \Leftarrow (b \Rightarrow c)$$

Monotonic

$$a \Rightarrow b \Longrightarrow a \wedge c \Rightarrow b \wedge c$$
$$a \Rightarrow b \Longrightarrow a \vee c \Rightarrow b \vee c$$
$$a \Rightarrow b \Longrightarrow (c \Rightarrow a) \Rightarrow (c \Rightarrow b)$$

Absorption

$$a \wedge (a \vee b) = a$$
$$a \vee (a \wedge b) = a$$

Direct Proof

$$(a \Rightarrow b) \wedge a \Rightarrow b$$
$$(a \Rightarrow b) \wedge \neg b \Rightarrow \neg a$$
$$(a \vee b) \wedge \neg a \Rightarrow b$$

Transitive

$$(a \wedge b) \wedge (b \wedge c) \Rightarrow (a \wedge c)$$
$$(a \Rightarrow b) \wedge (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$$
$$(a = b) \wedge (b = c) \Rightarrow (a = c)$$
$$(a \Rightarrow b) \wedge (b = c) \Rightarrow (a \Rightarrow c)$$
$$(a = b) \wedge (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$$

Distributive (Factoring)

$$a \wedge (b \wedge c) = (a \wedge b) \wedge (a \wedge c)$$
$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$$
$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$
$$a \vee (b \vee c) = (a \vee b) \vee (a \vee c)$$
$$a \vee (b \Rightarrow c) = (a \vee b) \Rightarrow (a \vee c)$$
$$a \vee (b = c) = (a \vee b) = (a \vee c)$$
$$a \Rightarrow (b \wedge c) = (a \Rightarrow b) \wedge (a \Rightarrow c)$$
$$a \Rightarrow (b \vee c) = (a \Rightarrow b) \vee (a \Rightarrow c)$$
$$a \Rightarrow (b \Rightarrow c) = (a \Rightarrow b) \Rightarrow (a \Rightarrow c)$$
$$a \Rightarrow (b = c) = (a \Rightarrow b) = (a \Rightarrow c)$$

Generalization

$$a \Rightarrow a \vee b$$

Antidistributive

$$a \wedge b \Rightarrow c = (a \Rightarrow c) \vee (b \Rightarrow c)$$
$$a \vee b \Rightarrow c = (a \Rightarrow c) \wedge (b \Rightarrow c)$$

Portation

$$a \wedge b \Rightarrow c = a \Rightarrow (b \Rightarrow c)$$
$$a \wedge b \Rightarrow c = a \Rightarrow \neg b \vee c$$

Conflation

$$(a \Rightarrow b) \wedge (c \Rightarrow d) \Longrightarrow a \wedge c \Rightarrow b \wedge d$$
$$(a \Rightarrow b) \wedge (c \Rightarrow d) \Longrightarrow a \vee c \Rightarrow b \vee d$$

Equality and Difference

$$a=b = (a \wedge b) \vee (\neg a \wedge \neg b)$$
$$a \neq b = (a \wedge \neg b) \vee (\neg a \wedge b)$$

Resolution

$a \wedge c \implies (a \vee b) \wedge (\neg b \vee c) = (a \wedge \neg b) \vee (b \wedge c) \implies a \vee c$

Case Creation

a = **if** b **then** $b \Rightarrow a$ **else** $\neg b \Rightarrow a$ **fi**
a = **if** b **then** $b \wedge a$ **else** $\neg b \wedge a$ **fi**
a = **if** b **then** $b = a$ **else** $b \neq a$ **fi**

Case Analysis

if a **then** b **else** c **fi** $= (a \wedge b) \vee (\neg a \wedge c)$
if a **then** b **else** c **fi** $= (a \Rightarrow b) \wedge (\neg a \Rightarrow c)$

Case Absorption

if a **then** b **else** c **fi** = **if** a **then** $a \wedge b$ **else** c **fi**
if a **then** b **else** c **fi** = **if** a **then** $a \Rightarrow b$ **else** c **fi**
if a **then** b **else** c **fi** = **if** a **then** $a = b$ **else** c **fi**
if a **then** b **else** c **fi** = **if** a **then** b **else** $\neg a \wedge c$ **fi**
if a **then** b **else** c **fi** = **if** a **then** b **else** $a \vee c$ **fi**
if a **then** b **else** c **fi** = **if** a **then** b **else** $a \neq c$ **fi**

One Case

if a **then** $\top$ **else** b **fi** $= a \vee b$
if a **then** $\bot$ **else** b **fi** $= \neg a \wedge b$
if a **then** b **else** $\top$ **fi** $= a \Rightarrow b$
if a **then** b **else** $\bot$ **fi** $= a \wedge b$
if a **then** b **else** $\neg b$ **fi** $= a = b$
if a **then** $\neg b$ **else** b **fi** $= a \neq b$

Case Distributive (Case Factoring)

$\neg$ **if** a **then** b **else** c **fi** = **if** a **then** $\neg b$ **else** $\neg c$ **fi**
if a **then** b **else** c **fi** $\wedge\ d$ = **if** a **then** $b \wedge d$ **else** $c \wedge d$ **fi**
and similarly replacing $\wedge$ by any of $\vee\ =\ \neq\ \Rightarrow\ \Leftarrow$
if a **then** $b \wedge c$ **else** $d \wedge e$ **fi** = **if** a **then** b **else** d **fi** $\wedge$ **if** a **then** c **else** e **fi**
and similarly replacing $\wedge$ by any of $\vee\ =\ \neq\ \Rightarrow\ \Leftarrow$

End of Binary

11.3.2 Numbers

Let d be a sequence of (zero or more) digits, and let x, y, and z be numbers.

$d0+1 = d1$	$d5+1 = d6$	Counting (see Exercise 32)
$d1+1 = d2$	$d6+1 = d7$	Counting
$d2+1 = d3$	$d7+1 = d8$	Counting
$d3+1 = d4$	$d8+1 = d9$	Counting
$d4+1 = d5$	$9+1 = 10$	Counting
for nonempty d	$d9+1 = (d+1)0$	Counting
$x+0 = x$		Identity
$x+y = y+x$		Symmetry
$x+(y+z) = (x+y)+z$		Associativity
$-\infty<x<\infty \implies (x+y = x+z = y=z)$		Cancellation
$-\infty<x \implies \infty+x = \infty$		Absorption
$x<\infty \implies -\infty + x = -\infty$		Absorption
$-x = 0-x$		Negation
$--x = x$		Self-inverse
$-(x+y) = -x + -y$		Distributivity
$-(x-y) = y-x$		Antisymmetry
$-x \times y = -(x \times y) = x \times -y$		Semi-distributivity
$-x / y = -(x/y) = x / -y$		Semi-distributivity
$x-0 = x$		Identity
$x-y = x + -y$		Subtraction
$x+(y-z) = (x+y)-z$		Associativity
$-\infty<x<\infty \implies (x-y = x-z = y=z)$		Cancellation
$-\infty<x<\infty \implies x-x = 0$		Inverse
$x<\infty \implies \infty-x = \infty$		Absorption
$-\infty<x \implies -\infty - x = -\infty$		Absorption
$-\infty<x<\infty \implies x \times 0 = 0$		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
$-\infty<x<\infty \wedge x\neq 0 \Rightarrow (x{\times}y = x{\times}z = y{=}z)$	Cancellation
$0<x \Rightarrow x{\times}\infty = \infty$	Absorption
$0<x \Rightarrow x \times -\infty = -\infty$	Absorption
$x/1 = x$	Identity
$x\neq 0 \Rightarrow 0/x = 0$	Base
$-\infty<x<\infty \wedge x\neq 0 \Rightarrow x/x = 1$	Base
$x{\times}(y/z) = (x{\times}y)/z = (x/z){\times}y = x/(z/y)$	Multiplication-Division
$(x/y)/z = x/(y{\times}z)$	Multiplication-Division
$-\infty<y<\infty \wedge y\neq 0 \Rightarrow (x/y){\times}y = x$	Multiplication-Division
$-\infty<x<\infty \Rightarrow x/\infty = 0 = x/-\infty$	Annihilation
$-\infty<x<\infty \Rightarrow x^0 = 1$	Base
$x^1 = x$	Identity
$-\infty<0<1<\infty$	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
$-\infty \leq x \leq \infty$	Extremes

$x\uparrow\infty = \infty$	$x\downarrow-\infty = -\infty$	Base
$x\uparrow-\infty = x$	$x\downarrow\infty = x$	Identity
$-(x\uparrow y) = -x\downarrow-y$	$-(x\downarrow y) = -x\uparrow-y$	Duality
$x + y\uparrow z = (x{+}y)\uparrow(x{+}z)$	$x - y\uparrow z = (x{-}y)\downarrow(x{-}z)$	Distributivity
$x\geq 0 \Rightarrow x{\times}(y\uparrow z) = (x{\times}y)\uparrow(x{\times}z)$	$x\geq 0 \Rightarrow x{\times}(y\downarrow z) = (x{\times}y)\downarrow(x{\times}z)$	Distributivity
$x\leq 0 \Rightarrow x{\times}(y\uparrow z) = (x{\times}y)\downarrow(x{\times}z)$	$x\leq 0 \Rightarrow x{\times}(y\downarrow z) = (x{\times}y)\uparrow(x{\times}z)$	Distributivity

End of Numbers

11.3.3 Bunches

Let x and y be elements (binaries, numbers, characters, sets, strings and lists of elements).

$x{:}\ y = x{=}y$	Elementary
$x{:}\ A, B = x{:}\ A \vee x{:}\ B$	Compound
$A, A = A$	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
$A{::}\ B = B{:}\ A$	Mirror
$\not\in null = 0$	Size
$\not\in x = 1$	Size

$¢\ nat\ =\ \infty$	Size
$¢(A, B) + ¢(A`B)\ =\ ¢A + ¢B$	Size
$\neg\ x{:}\ A\ =\!=\ ¢(A`x) = 0$	Size
$A{:}\ B\ \Rightarrow\ ¢A \le ¢B$	Size
$A, (A`B)\ =\ A\ =\ A`(A, B)$	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
$A, null\ =\ A\ =\ null, A$	Identity
$A`null\ =\ null\ =\ null`A$	Base
$¢A = 0\ =\!=\ A = null$	Size
$x, y{:}\ xint\ \wedge\ x{\le}y\ \Rightarrow\ (i{:}\ x{,..}y\ =\!=\ i{:}\ xint\ \wedge\ x{\le}i{<}y)$	Interval
$x, y{:}\ xint\ \wedge\ x{\le}y\ \Rightarrow\ ¢(x{,..}y)\ =\ y{-}x$	Interval
$nat = 0{,..}\infty$	Interval
$\infty, -\infty{:}\ \ x/0$	Division by 0
$xreal{:}\ 0/0$	Division by 0
$x^{y+z}{:}\ \ x^y \times x^z$	Adding Exponents
$x^{y\times z}{:}\ \ (x^y)^z$	Multiplying Exponents
$-null\ =\ null$	Distribution
$-(A, B)\ =\ -A, -B$	Distribution
$A{+}null\ =\ null\ =\ null{+}A$	Distribution
$(A, B){+}(C, D)\ =\ A{+}C, A{+}D, B{+}C, B{+}D$	Distribution

and similarly for many other operators (see Section 11.7)

End of Bunches

11.3.4 Sets

Let S be a set.

$\{{\sim}S\}\ =\ S$	$\{A\}{:}\ ♯B\ =\!=\ A{:}\ B$
${\sim}\{A\}\ =\ A$	$\$\{A\} = ¢A$
$\{A\}\ ♯\ A$	$\{A\} \cup \{B\}\ =\ \{A, B\}$
$A \in \{B\}\ =\!=\ A{:}\ B$	$\{A\} \cap \{B\}\ =\ \{A\ `\ B\}$
$\{A\} \subseteq \{B\}\ =\!=\ A{:}\ B$	$\{A\} = \{B\}\ =\!=\ A = B$
	$\{A\} ♯ \{B\}\ =\!=\ A ♯ B$

End of Sets

11.3.5 Strings

Let S, T, and U be strings; let i and j be items (binary values, numbers, characters, sets, lists, functions); let n and m be extended natural; let x, y, and z be extended integers such that $x{\le}y{\le}z$.

$S; nil\ =\ S\ =\ nil; S$	$S_{(T_U)}\ =\ (S_T)_U$
$S; (T; U)\ =\ (S; T); U$	$S_{nil}\ =\ nil$
$\leftrightarrow nil\ =\ 0$	$S_{T;\,U}\ =\ S_T; S_U$
$\leftrightarrow i\ =\ 1$	$S_{\{A\}}\ =\ \{S_A\}$
$\leftrightarrow(S; T)\ =\ \leftrightarrow S + \leftrightarrow T$	$\leftrightarrow S{<}\infty\ \Rightarrow\ nil\ \le\ S\ <\ S; i; T$

$\not\in nil = 1$
$\not\in(A; B) \le \not\in A \times \not\in B$
$\leftrightarrow S{<}\infty \Rightarrow (S; i; T)_{\leftrightarrow S} = i$
$\leftrightarrow S{<}\infty \Rightarrow S;i;T \triangleleft \leftrightarrow S \triangleright j = S;j;T$
$0{*}S = nil$
$(n{+}1){*}S = n{*}S; S$
$*S = **S = nat{*}S$

$\leftrightarrow S{<}\infty \wedge i{<}j \Rightarrow S; i; T < S; j; U$
$\leftrightarrow S{<}\infty \Rightarrow (S;A;T : S;B;T = A : B)$
$\leftrightarrow S{<}\infty \Rightarrow (i{=}j = S; i; T = S; j; T)$
$(S \triangleleft n \triangleright i)_m =$ **if** $n{=}m$ **then** i **else** S_m **fi**
$-\infty{<}x{<}\infty \Rightarrow x;..x = nil$
$-\infty{<}x{<}\infty \Rightarrow x;..x{+}1 = x$
$(x;..y) ; (y;..z) = x;..z$
$\leftrightarrow(x;..y) = y–x$

End of Strings

11.3.6 Lists

Let S and T be strings; let i be an item (binary value, number, character, set, list, function); let L, M, and N be lists; let n and m be extended natural.

$[S] \neq S = \sim[S]$
$[\sim L] = L$
$[S];;[T] = [S; T]$
$[S] = [T] = S = T$
$[S] < [T] = S < T$
$[A]: [B] = A: B$
$\#[S] = \leftrightarrow S$
$nil{\rightarrow}i \mid L = i$
$n{\rightarrow}i \mid [S] = [S \triangleleft n \triangleright i]$
$(n{\rightarrow}i \mid L)\, m =$ **if** $n{=}m$ **then** i **else** $L\, m$ **fi**
$(S; T){\rightarrow}i \mid L = S{\rightarrow}(T{\rightarrow}i \mid L@S) \mid L$

$\square L = 0,..\#L$
$[S]\, T = S_T$
$S_{[T]} = [S_T]$
$[S]\, [T] = [S_T]$
$L\, \{A\} = \{L\, A\}$
$L\, [S] = [L\, S]$
$(L\, M)\, N = L\, (M\, N)$
$\#L = \not\in\square L$
$L@nil = L$
$L@i = L\, i$
$L@(S; T) = L@S@T$

End of Lists

11.3.7 Functions

Renaming — if v and w do not appear in D and w does not appear in b
$\langle v: D{\cdot}\ b\rangle = \langle w: D{\cdot}\ \langle v: D{\cdot}\ b\rangle\, w\rangle$

Function Composition — if $\neg f: \square g$
$\square(g\, f) = \S x: \square f{\cdot}\ f\, x: \square g$
$(g\, f)\, x = g\, (f\, x)$

Domain
$\square\langle v: D{\cdot}\ b\rangle = D$

Application — if element $x: D$
$\langle v: D{\cdot}\ b\rangle\, x =$ (substitute x for v in b)

Distributive
$f\, null = null$
$f\, (A, B) = f\, A, f\, B$
$f\, (\S g) = \S y: f\, (\square g){\cdot}\ \exists x: \square g{\cdot}\ f\, x = y \wedge g\, x$
f **if** b **then** x **else** y **fi** = **if** b **then** $f\, x$ **else** $f\, y$ **fi**
if b **then** f **else** g **fi** $x =$ **if** b **then** $f\, x$ **else** $g\, x$ **fi**

Function Union
$\square(f, g) = \square f \text{ ‘ } \square g$
$(f, g)\, x = f\, x, g\, x$

Function Intersection
$\square(f \text{ ‘ } g) = \square f, \square g$
$(f \text{ ‘ } g)x = (f \mid g)x \text{ ‘ } (g \mid f)x$

Selective Union
$\square(f \mid g) = \square f, \square g$
$(f \mid g)x =$ **if** $x: \square f$ **then** $f\, x$ **else** $g\, x$ **fi**
$f \mid f = f$
$f \mid (g \mid h) = (f \mid g) \mid h$
$(g \mid h)\, f = g\, f \mid h\, f$

Function Inclusion and Equality
$f: g = \square f:: \square g \wedge \forall x: \square g{\cdot}\ f\, x: g\, x$
$f = g = \square f = \square g \wedge \forall x: \square f{\cdot}\ f\, x = g\, x$

Arrow

$f\colon null \to A$
$A \to B\colon\ (A\,`\,C) \to (B, D)$
$(A, B) \to C\ =\ A \to C \mid B \to C$
$f\colon A \to B\ =\ \Box f\colon\colon A\ \wedge\ \forall a\colon A \cdot f\,a\colon B$

Size

$\#f\ =\ ¢\Box f$

Extension

$f\ =\ \langle v\colon \Box f \cdot f\,v \rangle$

End of Functions

11.3.8 Quantifiers

Let x be an element, let a, b and c be binary, let n and m be numeric, let f and g be functions, and let p be a predicate.

$\forall v\colon null \cdot b\ =\ \top$
$\forall v\colon x \cdot b\ =\ \langle v\colon x \cdot b \rangle\, x$

$\forall v\colon A, B \cdot b\ =\ (\forall v\colon A \cdot b) \wedge (\forall v\colon B \cdot b)$
$\forall v\colon (\S v\colon D \cdot b) \cdot c\ =\ \forall v\colon D \cdot b \Rightarrow c$

$\exists v\colon null \cdot b\ =\ \bot$
$\exists v\colon x \cdot b\ =\ \langle v\colon x \cdot b \rangle\, x$

$\exists v\colon A, B \cdot b\ =\ (\exists v\colon A \cdot b) \vee (\exists v\colon B \cdot b)$
$\exists v\colon (\S v\colon D \cdot b) \cdot c\ =\ \exists v\colon D \cdot b \wedge c$

$\Sigma v\colon null \cdot n\ =\ 0$
$\Sigma v\colon x \cdot n\ =\ \langle v\colon x \cdot n \rangle\, x$

$(\Sigma v\colon A, B \cdot n) + (\Sigma v\colon A\,`\,B \cdot n)\ =\ (\Sigma v\colon A \cdot n) + (\Sigma v\colon B \cdot n)$
$\Sigma v\colon (\S v\colon D \cdot b) \cdot n\ =\ \Sigma v\colon D \cdot$ **if** b **then** n **else** 0 **fi**

$\Pi v\colon null \cdot n\ =\ 1$
$\Pi v\colon x \cdot n\ =\ \langle v\colon x \cdot n \rangle\, x$

$(\Pi v\colon A, B \cdot n) \times (\Pi v\colon A\,`\,B \cdot n)\ =\ (\Pi v\colon A \cdot n) \times (\Pi v\colon B \cdot n)$
$\Pi v\colon (\S v\colon D \cdot b) \cdot n\ =\ \Pi v\colon D \cdot$ **if** b **then** n **else** 1 **fi**

$\Downarrow v\colon null \cdot n\ =\ \infty$
$\Downarrow v\colon x \cdot n\ =\ \langle v\colon x \cdot n \rangle\, x$

$\Downarrow v\colon A, B \cdot n\ =\ (\Downarrow v\colon A \cdot n) \downarrow (\Downarrow v\colon B \cdot n)$
$\Downarrow v\colon (\S v\colon D \cdot b) \cdot n\ =\ \Downarrow v\colon D \cdot$ **if** b **then** n **else** ∞ **fi**

$\Uparrow v\colon null \cdot n\ =\ -\infty$
$\Uparrow v\colon x \cdot n\ =\ \langle v\colon x \cdot n \rangle\, x$

$\Uparrow v\colon A, B \cdot n\ =\ (\Uparrow v\colon A \cdot n) \uparrow (\Uparrow v\colon B \cdot n)$
$\Uparrow v\colon (\S v\colon D \cdot b) \cdot n\ =\ \Uparrow v\colon D \cdot$ **if** b **then** n **else** $-\infty$ **fi**

$\S v\colon null \cdot b\ =\ null$
$\S v\colon x \cdot b\ =$ **if** $\langle v\colon x \cdot b \rangle\, x$ **then** x **else** $null$ **fi**
$\S v\colon A, B \cdot b\ =\ (\S v\colon A \cdot b), (\S v\colon B \cdot b)$
$\S v\colon A\,`\,B \cdot b\ =\ (\S v\colon A \cdot b)\ `\ (\S v\colon B \cdot b)$
$\S v\colon (\S v\colon D \cdot b) \cdot c\ =\ \S v\colon D \cdot b \wedge c$

Inclusion

$A\colon B\ =\ \forall x\colon A \cdot x\colon B$

Cardinality

$¢A\ =\ \Sigma\,(A \to 1)$

Change of Variable — if d does not appear in b

$\forall r\colon f\,D \cdot b\ =\ \forall d\colon D \cdot \langle r\colon f\,D \cdot b \rangle\,(f\,d)$
$\exists r\colon f\,D \cdot b\ =\ \exists d\colon D \cdot \langle r\colon f\,D \cdot b \rangle\,(f\,d)$
$\Downarrow r\colon f\,D \cdot n\ =\ \Downarrow d\colon D \cdot \langle r\colon f\,D \cdot n \rangle\,(f\,d)$
$\Uparrow r\colon f\,D \cdot n\ =\ \Uparrow d\colon D \cdot \langle r\colon f\,D \cdot n \rangle\,(f\,d)$

Identity

$\forall v \cdot \top$
$\neg \exists v \cdot \bot$

Specialize and Generalize — if element $x\colon \Box f$

$\Downarrow f \le f\,x \le \Uparrow f$

Bunch-Element Conversion

$A\colon B\ =\ \forall a\colon A \cdot \exists b\colon B \cdot a = b$
$f\,A\colon g\,B\ =\ \forall a\colon A \cdot \exists b\colon B \cdot f\,a = g\,b$

Distributive — if $D \neq null$ and v does not appear in a

$a \wedge \forall v\colon D \cdot b\ =\ \forall v\colon D \cdot a \wedge b$
$a \wedge \exists v\colon D \cdot b\ =\ \exists v\colon D \cdot a \wedge b$
$a \vee \forall v\colon D \cdot b\ =\ \forall v\colon D \cdot a \vee b$
$a \vee \exists v\colon D \cdot b\ =\ \exists v\colon D \cdot a \vee b$
$a \Rightarrow \forall v\colon D \cdot b\ =\ \forall v\colon D \cdot a \Rightarrow b$
$a \Rightarrow \exists v\colon D \cdot b\ =\ \exists v\colon D \cdot a \Rightarrow b$

Idempotent — if $D \neq null$ and v does not appear in b

$\forall v\colon D \cdot b\ =\ b$
$\exists v\colon D \cdot b\ =\ b$

Absorption — if x: D

$\langle v\colon D\cdot b\rangle\, x \wedge \exists v\colon D\cdot b = \langle v\colon D\cdot b\rangle\, x$
$\langle v\colon D\cdot b\rangle\, x \vee \forall v\colon D\cdot b = \langle v\colon D\cdot b\rangle\, x$
$\langle v\colon D\cdot b\rangle\, x \wedge \forall v\colon D\cdot b = \forall v\colon D\cdot b$
$\langle v\colon D\cdot b\rangle\, x \vee \exists v\colon D\cdot b = \exists v\colon D\cdot b$

Specialization — if element x: $\square p$

$\forall p \Rightarrow p\, x$

One-Point — if x: D

and v does not appear in x
$\forall v\colon D\cdot v{=}x \Rightarrow b = \langle v\colon D\cdot b\rangle\, x$
$\exists v\colon D\cdot v{=}x \wedge b = \langle v\colon D\cdot b\rangle\, x$

Duality

$\neg\forall v\cdot b = \exists v\cdot \neg b$
$\neg\exists v\cdot b = \forall v\cdot \neg b$
$-\Uparrow v\cdot n = \Downarrow v\cdot -n$
$-\Downarrow v\cdot n = \Uparrow v\cdot -n$

Solution

$\S v\colon D\cdot \top = D$
$(\S v\colon D\cdot b)\colon D$
$\S v\colon D\cdot \bot = null$
$(\S v\cdot b)\colon (\S v\cdot c) = \forall v\cdot b{\Rightarrow}c$
$(\S v\cdot b), (\S v\cdot c) = \S v\cdot b \vee c$
$(\S v\cdot b)\ \text{`}\ (\S v\cdot c) = \S v\cdot b \wedge c$
$x\colon \S p = x\colon \square p \wedge p\, x$
$\forall f = (\S f){=}(\square f)$
$\exists f = (\S f){\neq}null$

Bounding — if $D{\neq}null$

and v does not appear in n
$n > (\Uparrow v\colon D\cdot m) \Rightarrow (\forall v\colon D\cdot n{>}m)$
$n < (\Downarrow v\colon D\cdot m) \Rightarrow (\forall v\colon D\cdot n{<}m)$
$n \geq (\Uparrow v\colon D\cdot m) = (\forall v\colon D\cdot n{\geq}m)$
$n \leq (\Downarrow v\colon D\cdot m) = (\forall v\colon D\cdot n{\leq}m)$
$n \geq (\Downarrow v\colon D\cdot m) \Leftarrow (\exists v\colon D\cdot n{\geq}m)$
$n \leq (\Uparrow v\colon D\cdot m) \Leftarrow (\exists v\colon D\cdot n{\leq}m)$
$n > (\Downarrow v\colon D\cdot m) = (\exists v\colon D\cdot n{>}m)$
$n < (\Uparrow v\colon D\cdot m) = (\exists v\colon D\cdot n{<}m)$

Antidistributive — if $D{\neq}null$

and v does not appear in a
$a \Leftarrow \exists v\colon D\cdot b = \forall v\colon D\cdot a \Leftarrow b$
$a \Leftarrow \forall v\colon D\cdot b = \exists v\colon D\cdot a \Leftarrow b$

Generalization — if element x: $\square p$

$p\, x \Rightarrow \exists p$

Splitting — 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

$\forall v\cdot \forall w\cdot b = \forall w\cdot \forall v\cdot b$
$\exists v\cdot \exists w\cdot b = \exists w\cdot \exists v\cdot b$

Semicommutative

$\exists v\cdot \forall w\cdot b \Rightarrow \forall w\cdot \exists v\cdot b$
$\forall x\cdot \exists y\cdot p\, x\, y = \exists f\cdot \forall x\cdot p\, x\, (f\, x)$

Domain Change

$A\colon B \Rightarrow (\forall v\colon A\cdot b) \Leftarrow (\forall v\colon B\cdot b)$
$A\colon B \Rightarrow (\exists v\colon A\cdot b) \Rightarrow (\exists v\colon B\cdot b)$
$\forall v\colon A\cdot v\colon B \Rightarrow p = \forall v\colon A\text{`}B\cdot p$
$\exists v\colon A\cdot v\colon B \wedge p = \exists v\colon A\text{`}B\cdot p$

Extreme

$(\Downarrow n\colon int\cdot n) = (\Downarrow n\colon real\cdot n) = -\infty$
$(\Uparrow n\colon int\cdot n) = (\Uparrow n\colon real\cdot n) = \infty$

Connection

$n{\leq}m = \forall k\cdot k{\leq}n \Rightarrow k{\leq}m$
$n{\leq}m = \forall k\cdot k{<}n \Rightarrow k{<}m$
$n{\leq}m = \forall k\cdot m{\leq}k \Rightarrow n{\leq}k$
$n{\leq}m = \forall k\cdot m{<}k \Rightarrow n{<}k$

Distributive — if $D{\neq}null$ and v does not appear in n

$n{\uparrow}(\Uparrow v\colon D\cdot m) = (\Uparrow v\colon D\cdot n{\uparrow}m)$
$n{\uparrow}(\Downarrow v\colon D\cdot m) = (\Downarrow v\colon D\cdot n{\uparrow}m)$
$n + (\Uparrow v\colon D\cdot m) = (\Uparrow v\colon D\cdot n{+}m)$
$n - (\Uparrow v\colon D\cdot m) = (\Downarrow v\colon D\cdot n{-}m)$
$(\Uparrow v\colon D\cdot m) - n = (\Uparrow v\colon D\cdot m{-}n)$
$n{\geq}0 \Rightarrow n{\times}(\Uparrow v\colon D\cdot m) = (\Uparrow v\colon D\cdot n{\times}m)$
$n{\leq}0 \Rightarrow n{\times}(\Uparrow v\colon D\cdot m) = (\Downarrow v\colon D\cdot n{\times}m)$
$n{\times}(\Sigma v\colon D\cdot m) = (\Sigma v\colon D\cdot n{\times}m)$

$n{\downarrow}(\Downarrow v\colon D\cdot m) = (\Downarrow v\colon D\cdot n{\downarrow}m)$
$n{\downarrow}(\Uparrow v\colon D\cdot m) = (\Uparrow v\colon D\cdot n{\downarrow}m)$
$n + (\Downarrow v\colon D\cdot m) = (\Downarrow v\colon D\cdot n{+}m)$
$n - (\Downarrow v\colon D\cdot m) = (\Uparrow v\colon D\cdot n{-}m)$
$(\Downarrow v\colon D\cdot m) - n = (\Downarrow v\colon D\cdot m{-}n)$
$n{\geq}0 \Rightarrow n{\times}(\Downarrow v\colon D\cdot m) = (\Downarrow v\colon D\cdot n{\times}m)$
$n{\leq}0 \Rightarrow n{\times}(\Downarrow v\colon D\cdot m) = (\Uparrow v\colon D\cdot n{\times}m)$
$(\Pi v\colon D\cdot m)^n = (\Pi v\colon D\cdot m^n)$

End of Quantifiers

11.3.9 Limits

Let all domains be nat .

$(\Uparrow m\cdot \Downarrow n\cdot f(m{+}n)) \le \Updownarrow f \le (\Downarrow m\cdot \Uparrow n\cdot f(m{+}n))$

$\exists m\cdot \forall n\cdot p(m{+}n) \implies \Updownarrow p \implies \forall m\cdot \exists n\cdot p(m{+}n)$

$\Updownarrow n\cdot n = \infty$

End of Limits

11.3.10 Specifications and Programs

For specifications P , Q , R , and S , and binary b ,

$ok = x'{=}x \wedge y'{=}y \wedge \ldots$

$x{:=}\,e = x'{=}e \wedge y'{=}y \wedge \ldots$

$P.Q = \exists x'', y'', \ldots\cdot \langle x', y', \ldots\cdot P\rangle\, x''\, y''\, \ldots \wedge \langle x, y, \ldots\cdot Q\rangle\, x''\, y''\, \ldots$

$P \| Q = \exists t_P, t_Q\cdot \langle t'\cdot P\rangle\, t_P \wedge \langle t'\cdot Q\rangle\, t_Q \wedge t' = t_P \uparrow t_Q$

if b **then** P **else** Q **fi** $= b \wedge P \vee \neg b \wedge Q = (b \Rightarrow P) \wedge (\neg b \Rightarrow Q)$

new x: $T\cdot P = \exists x, x'\colon T\cdot P$

frame $x\cdot P = P \wedge y'{=}y \wedge \ldots$

while b **do** P **od** $= t'{\ge}t \wedge$ **if** b **then** $P.\ t{:=}\,t{+}1.$ **while** b **do** P **od else** ok **fi**

$\forall \sigma, \sigma'\cdot$ **if** b **then** $P.\ W$ **else** ok **fi** $\Leftarrow W \implies \forall \sigma, \sigma'\cdot$ **while** b **do** P **od** $\Leftarrow W$

To prove $F\,m \Leftarrow$ **for** $i{:=}\,m;..n$ **do** P **od**

prove $F\,i \Leftarrow i{:}\,m,..n \wedge (P.\ F(i{+}1))$

and $F\,n \Leftarrow ok$

$A\,m \Rightarrow A'n \Leftarrow$ **for** $i{:=}\,m;..n$ **do** $i{:}\,m,..n \wedge A\,i \Rightarrow A'(i{+}1)$ **od**

wait until $w = t{:=}\,t\uparrow w$

assert $b = $ **if** b **then** ok **else** $screen!$ “error”. **wait until** ∞ **fi**

ensure $b = b \wedge ok$

$P.\ (P$ **value** $e){=}e$ but do not double-prime or substitute in $(P$ **value** $e)$

Data transformer D satisfies $\forall new\cdot \exists old\cdot D$ and transforms specification S to

$\forall old\cdot D \Rightarrow \exists old'\cdot D' \wedge S$

$c? = t{:=}\,t\uparrow(\mathcal{T}c_{\boldsymbol{rc}} + (\text{transit time}).\ \boldsymbol{rc}{:=}\,\boldsymbol{rc}{+}1$

$c = \mathcal{M}c_{\boldsymbol{rc}-1}$

$c!\,e = \mathcal{M}c_{\boldsymbol{wc}} = e \wedge \mathcal{T}c_{\boldsymbol{wc}} = t \wedge (\boldsymbol{wc}{:=}\,\boldsymbol{wc}{+}1)$

$\surd c = \mathcal{T}c_{\boldsymbol{rc}} + (\text{transit time}) \le t$

new x: $time{\to}T\cdot S = \exists x\colon time{\to}T\cdot S$

new $c?\ T\cdot S = \exists \mathcal{M}c\colon \infty{*}T\cdot \exists \mathcal{T}c\colon \infty{*}xnat\cdot \exists \boldsymbol{rc}, \boldsymbol{rc}', \boldsymbol{wc}, \boldsymbol{wc}'\colon xnat\cdot \boldsymbol{rc}{=}\boldsymbol{wc}{=}0 \wedge S$

$P.ok = P = ok.P$	identity
$P.(Q.R) = (P.Q).R$	associativity
$P{\vee}Q.R{\vee}S = (P.R) \vee (P.S) \vee (Q.R) \vee (Q.\mathrm{S})$	distributivity
if b **then** P **else** Q **fi**$.R =$ **if** b **then** $P.R$ **else** $Q.R$ **fi**	distributivity (unprimed b)
$P.$ **if** b **then** Q **else** R **fi** $=$ **if** $P.b$ **then** $P.Q$ **else** $P.R$ **fi**	distributivity (unprimed b)
$P \parallel Q = Q \parallel P$	symmetry
$P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$	associativity
$P \parallel Q{\vee}R = (P \parallel Q) \vee (P \parallel R)$	distributivity
$P \parallel$ **if** b **then** Q **else** R **fi** $=$ **if** b **then** $P \parallel Q$ **else** $P \parallel R$ **fi**	distributivity
if b **then** $P{\|}Q$ **else** $R{\|}S$ **fi** $=$ **if** b **then** P **else** R **fi** $\parallel$ **if** b **then** Q **else** S **fi**	distributivity
$x{:=}$ **if** b **then** e **else** f **fi** $=$ **if** b **then** $x{:=}\,e$ **else** $x{:=}\,f$ **fi**	functional-imperative

End of Specifications and Programs

11.3.11 Substitution

Let x and y be different boundary state variables, let e and f be expressions of the prestate, and let S be a specification.

$x{:=}\,e\,.\,S \;=\;$ (for x substitute e in S)

$(x{:=}\,e \parallel y{:=}\,f).\;S \;=\;$ (for x substitute e and concurrently for y substitute f in S)

End of Substitution

11.3.12 Assertions

Let P and Q be specifications. Let A be an assertion and let A' be the same as A but with primes on all the variables.

$A \wedge (P.\,Q) \;\Leftarrow\; A \wedge P.\,Q$

$A \Rightarrow (P.Q) \;\Leftarrow\; A \Rightarrow P.\,Q$

$(P.Q) \wedge A' \;\Leftarrow\; P.\,Q \wedge A'$

$(P.Q) \Leftarrow A' \;\Leftarrow\; P.\,Q \Leftarrow A'$

$P.\,A \wedge Q \;\Leftarrow\; P \wedge A'.\,Q$

$P.\,Q \;\Leftarrow\; P \wedge A'.\;A \Rightarrow Q$

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

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

End of Assertions

11.3.13 Refinement

Refinement by Steps (Stepwise Refinement) (monotonicity, transitivity)

If $A \Leftarrow$ **if** b **then** C **else** D **fi** and $C \Leftarrow E$ and $D \Leftarrow F$ are theorems,
then $A \Leftarrow$ **if** b **then** E **else** F **fi** 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 **fi** and $E \Leftarrow$ **if** b **then** F **else** G **fi** are theorems,
then $A \wedge E \Leftarrow$ **if** b **then** $C \wedge F$ **else** $D \wedge G$ **fi** 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 $A \wedge D \Leftarrow B \wedge E \parallel C \wedge F$ 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 **fi** 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

11.4 Names

abs: *xreal*→(§*r*: *xreal*· *r*≥0)	*abs r* = **if** *r*≥0 **then** *r* **else** –*r* **fi**
bin (the binary values)	*bin* = ⊤, ⊥
ceil: *real*→*int*	*r* ≤ *ceil r* < *r*+1
char (the characters)	*char* = ..., "a", "A", ...
div: *real*→(§*r*: *real*· *r*>0)→*int*	*div x y* = *floor* (*x*/*y*)
divides: (*nat*+1)→*int*→*bin*	*divides n i* = *i*/*n*: *int*
entro: *prob*→(§*r*: *xreal*· *r*≥0)	*entro p* = *p* × *info p* + (1–*p*) × *info* (1–*p*)
even: *int*→*bin*	*even i* = *i*/2: *int*
	even = *divides* 2
floor: *real*→*int*	*floor r* ≤ *r* < *floor r* + 1
info: *prob*→(§*r*: *xreal*· *r*≥0)	*info p* = – *log p*
int (the integers)	*int* = *nat*, –*nat*
log: (§*r*: *xreal*· *r*≥0)→*xreal*	*log* (2^x) = *x*
	log (*x*×*y*) = *log x* + *log y*
mod: *real*→(§*r*: *real*· *r*>0)→*real*	0 ≤ *mod a d* < *d*
	a = *div a d* × *d* + *mod a d*
nat (the naturals)	0, *nat*+1: *nat*
	0, *B*+1: *B* ⇒ *nat*: *B*
nil (the empty string)	↔*nil* = 0
	nil; *S* = *S* = *S*; *nil*
	nil ≤ *S*
null (the empty bunch)	¢*null* = 0
	null, *A* = *A* = *A*, *null*
	null: *A*
odd: *int*→*bin*	*odd i* = ¬ *i*/2: *int*
	odd = ¬*even*
ok (the empty program)	*ok* = σ′=σ
	ok.*P* = *P* = *P*.*ok*
prob (probability)	*prob* = §*r*: *real*· 0≤*r*≤1
rand (random number)	*rand n*: 0,..*n*
rat (the rationals)	*rat* = *int*/(*nat*+1)
real (the reals)	*real* = §*r*: *xreal*· –∞ < *r* <∞
suc: *nat*→(*nat*+1)	*suc n* = *n*+1
time (time)	either *time*=*xnat* or *time*=§*r*: *xreal*· *r*≥0
xint (the extended integers)	*xint* = –∞, *int*, ∞
xnat (the extended naturals)	*xnat* = *nat*, ∞
xrat (the extended rationals)	*xrat* = –∞, *rat*, ∞
xreal (the extended reals)	*x*: *xreal* = ∃*f*: *nat*→*rat*· *x* = ⇳*f*

End of Names

11.5 Symbols

symbol	page	pronunciation
⊤	3	top, true
⊥	3	bottom, false
¬	3	not
∧	3	and
∨	3	or
⇒	3	implies
⟹	4	implies
⇐	3	follows from, is implied by
⟸	4	follows from, is implied by
=	3	equals, if and only if
⩵	4	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
−	12	minus
×	12	times, multiplication
/	12	divided by
↑	12	maximum
↓	12	minimum
,	14	bunch union
,..	16	union from (including) to (excluding)
‘	14	bunch intersection
;	17	string join
;;	20	list join
;..	19	join from (including) to (excluding)
:	14	is in, are in, bunch inclusion
::	14	includes
:=	36	assignment
∞	78	label, target of **go to**
.	36	sequential composition
·	23	function and quantifier
!	136	output
?	136	input

symbol	page	pronunciation
√	136	input check
( )	4	precedence brackets
{ }	17	set brackets
[ ]	20	list brackets
⟨ ⟩	23	function brackets
↯	17	power
¢	14	bunch size, cardinality
$	17	set size, cardinality
↔	17	string size (length)
#	20,23	list size (length), function size
|	20,24	otherwise, selective union
‖	121	concurrent (parallel) composition
~	17,20	contents of a set or list
*	18	repetition of a string
□	20,23	domain of a list or function
→	25	function arrow
∈	17	element of a set
⊆	17	subset
∪	17	set union
∩	17	set intersection
@	22	index with a pointer
∀	26	for all, universal quantifier
∃	26	there exists, existential quantifier
Σ	26	sum of, summation quantifier
Π	26	product of, product quantifier
⇑	26	maximum (least upper bound) quantifier
⇓	26	minimum (greatest lower bound) quantifier
⇳	33	limit quantifier
§	28	those, solution quantifier
′	34	x' is final value of state variable x
“ ”	13,19	“hi” is a text or string of characters
a^b	12	exponentiation
a_b	18	string indexing
$a\ b$	20,24,31	indexing, application, composition
◅ ▻	18	string modification
∞	12	infinity

assert	79	**if then else fi**	4
do od	73	**new**	68
ensure	80	**or**	80
exit when	73	**value**	81
for do od	76	**wait until**	79
frame	69	**while do od**	71
go to	78		

End of Symbols

11.6 Precedence

0	⊤ ⊥ ( ) { } [ ] ⟨ ⟩ **if fi** **do od** number text name superscript subscript
1	@ adjacency
2	prefix– ¢ $ ↔ # * ~ ↯ □ → √ ∀ ∃ Σ Π ⇑ ⇓ ⇳ §
3	× / ∩ ↑ ↓
4	+ infix– ∪
5	; ;.. ;; ‘
6	, ,.. | ◅ ▻
7	= ≠ < > ≤ ≥ : :: ∈ ⊆
8	¬
9	∧
10	∨
11	⇒ ⇐
12	:= ! ?
13	**exit when** **go to** **wait until** **assert** **ensure** **or**
14	. ‖ **value**
15	∀· ∃· Σ· Π· ⇑· ⇓· ⇳· §· **new·** **frame·**
16	⩵ ⟹ ⟸

Superscripting and subscripting associate from right to left, and bracket what is in them.

Adjacency 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 × ∩ ↑ ↓ + ∪ ; ;; ‘ , | ∧ ∨ . ‖ are associative (they associate in both directions).

Quantifiers as prefix operators are on level 2, but in the abbreviated quantifier notation they are on level 15.

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 $a \le b \wedge b < c$.

The operators ⩵ ⟹ ⟸ are identical to = ⇒ ⇐ except for precedence.

End of Precedence

11.7 Distribution

The operators in the following expressions distribute over bunch union in any operand:

$\neg A$ $A \wedge B$ $A \vee B$ $A \Rightarrow B$ $A \Leftarrow B$ $-A$ $A+B$ $A-B$ $A \times B$ A/B A^B $A \uparrow B$ $A \downarrow B$
A,B $A‘B$ $\$A$ $A \cup B$ $A \cap B$ $\sim A$ $A;B$ $\leftrightarrow A$ A_B $[A]$ $A;;B$ $A\ B$ $\#A$ $A@B$

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